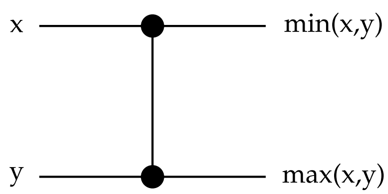
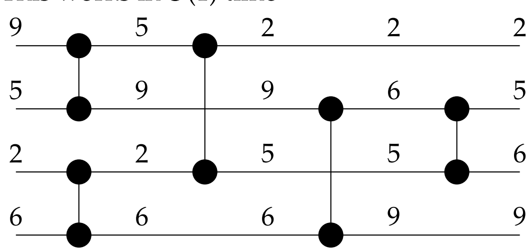
Sorting Part 3

1 Networks

1.1 Comparator



This works in $O(1)$ time



Wires go straight, left to right
Each comparator has inputs/outputs on some pairs of wires
Data marches left to right, in goose step (synchronised)

The depth is the longest chain of comparators that could be encountered, so the above example has depth of 3

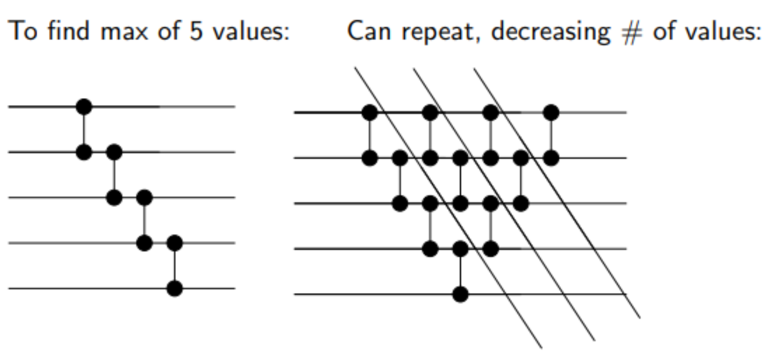
1.2 Claim

This comparison network will sort any set of 4 input values

1.3 Proof

- after leftmost comparators, min is on either wire 1 or 3, max is on either 2 or 4
- after next two comparators, min is on wire 1, max on 4
- last comparator gets correct values onto 2 and 3

1.4 Selection sorter



Depth: $D(n) = D(n-1) + 2, D(2) = 1 \Rightarrow D(n) = 2n - 3 = \Theta(n)$
If view depth as "time" parallelism gets us faster method than any sequential comparison based sort

1.5 Can we do better

We can do better than $\Theta(n)$, the AKS network has depth $O(\log n)$, with the caveats:

- Huge constant
- Really hard to construct
- Highly impractical - of theoretical interest only

1.6 Simplification

- Consider the line (1D array) of length n
- Each node 1,. . . ,n stores one of the items to be sorted
- in even steps even-numbered nodes i compare/exchange with their neighbour i+1
- in odd steps odd-numbered nodes i compare/exchange with their neighbour i+1
- called **odd-even transposition sort** (OETS)

For example:

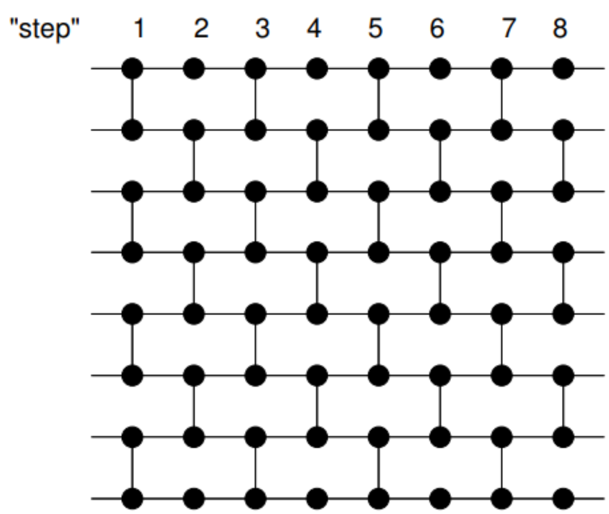
6		3		1		7		8		2		4		5	
3	-	6		1	-	7		2	-	8		4	-	5	step 1, odd
3		1	-	6		2	-	7		4	-	8		5	step 2, even
1	-	3		2	-	6		4	-	7		5	-	8	step 3, odd
1		2	-	3		4	-	6		5	-	7		8	step 4, even
1	-	2		3	-	4		5	-	6		7	-	8	step 5, odd

This is very similar to **bubble sort**

The questions are now:

- does it always work? yes
- how long does it take? no more than n steps

It can be viewed as a sorting network of depth n:



We will prove this using the 0/1 principle and induction

Lemma (0-1 principle)
Let CE be an **oblivious compare-exchange algorithm** or network. CE correctly sort all sequences of integer numbers iff CE correctly sorts all 0-1 sequences

Oblivious - does the same steps no matter what the values of the input is
Lemma(OETS)
An OETS network of depth n sorts any input of length n

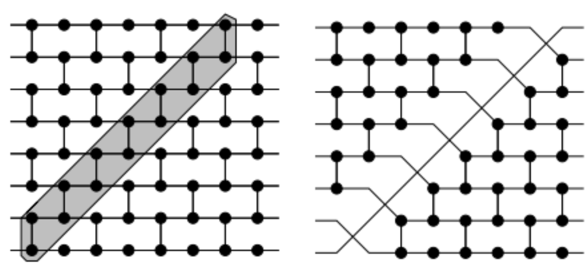
Base: $n \leqslant 2$ clear (comparators work by definition)
Step $n-1 \rightarrow n$ Let N be OETS network for n elements and let $a = (a_0, ..., a_{n-1})$ be 0/1 sequence

Case 1 if a_{n-1} (bottom row) then:

- bottom row of comparators obsolete
- we've got OETS network for n-1 elements plus superfluous row and column
- by hypothesis $(a_0, ..., a_{n-2})$ get sorted
- a_{n-1} already in proper position so we're done

Case 2 if $a_{n-1} = 0$ then

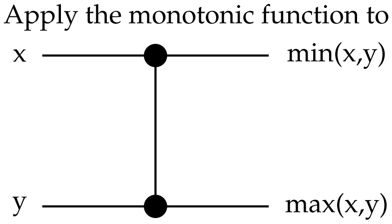
- Any comparator seeing this 0 performs swap (if other element is also 0 then jump horses); might as well replace them with fixed crossing lines



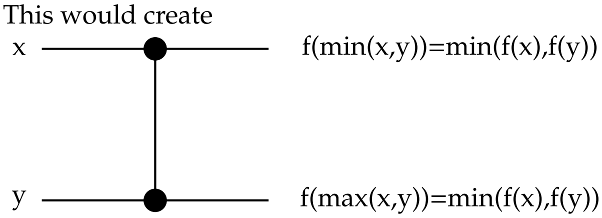
- What remains is OETS network for input size n-1 so we're done

1.7 Proving the 0-1 Principle

Assume you have an input $< a_1, a_2, ..., a_n > \xrightarrow{C} < b_1, b_2, ..., b_n >$
You have a monotonic function $f : \mathbb{Z} \rightarrow \mathbb{Z}$
Where monotone is for every $x \leqslant y \Rightarrow f(x) \leqslant f(y)$
Apply the monotonic function to a comparator



This would create



As this is true for a comparator, it will be true for any network of comparators

Applying this to the sequence:
$< (f(a_1), f(a_2), ..., f(a_n)) > \xrightarrow{C} < f(b_1), f(b_2), ..., f(b_n) >$
The winning elements of the first sequence($< a_1, a_2, ..., a_n > \xrightarrow{C} < b_1, b_2, ..., b_n >$) will be the same as the winning elements of this sequence

Assume you have some input $a_1, a_2, ..., a_i, ..., a_j, ...$ where $a_i > a_j$ and their order is maintained after going through the function, then the function is wrong.

$$f(x) = \{0\ if\ x < a_i \quad 1\ if\ x \geqslant a_i\}$$

Out of this, you get a sequence of 0s and 1s sorted incorrectly.

Want to prove C sorts 0-1 input ⇒ sorts any input

Proved it sorts some input incorrectly ⇒ sorts some 0-1 input incorrectly (contrapositive proof)

This is because in logic:

$$A \Rightarrow B \equiv \neg B \Rightarrow \neg A$$

1.8 Bitonic sequences

Formal definition: A sequence $(a_0, ..., a_{n-1})$ is called **bitonic** if

1. There is an index j, $0 \leqslant j < n$ such that $(a_0, \ldots, a_j)$ is monotonically increasing , and $(a_j, \ldots, a_{n-1})$ is monotonically decreasing
2. if (1) is not fulfilled, then there is an index i, $0 \leqslant i < n$ such that $(a_i, ..., a_{n-1}, a_0, ..., a_{i-1})$ does fulfill (1). i.e. you can just loop the numbers round to the front (see example below)

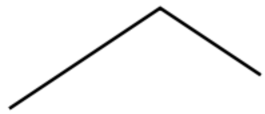
Examples:

- (0,2,3,5,6,7,3,1) is bitonic by (1), j=5
- (6,7,5,3,0,1,4,5) is bitonic by (2), i=4, j=5, after shift: (0,1,4,5,6,7,5,3)
- An example of a non bitonic sequence is 1,3,2,4

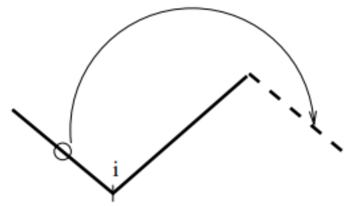
All bitonic sequences of 0s and 1s are of the form:

- $0^i 1^j 0^k$
- $1^i 0^j 1^k$

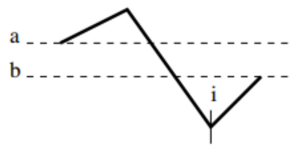
1.8.1 "Shapes" of bitonic sequences



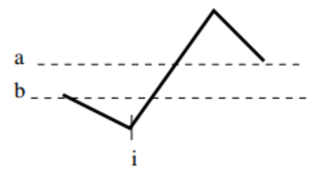
by (1)



by (2)



by (2)
b not higher than a



by (2)
b not higher than a

1.8.2 Properties of bitonic sequences

- Property "bitonic" is closed under cyclic shift (remains bitonic under any cyclic shift)
- Every sub-sequence of a bitonic sequence is bitonic itself
- If

  $$A = (a_0, \ldots, a_i)$$

  monotonically increasing and

  $$B = (b_{i+1}, \ldots, b_{n-1})$$

  is monotonically decreasing, then

  $$AB = (a_0, \ldots, a_i, b_{i+1}, b_{n-1})$$

  is bitonic

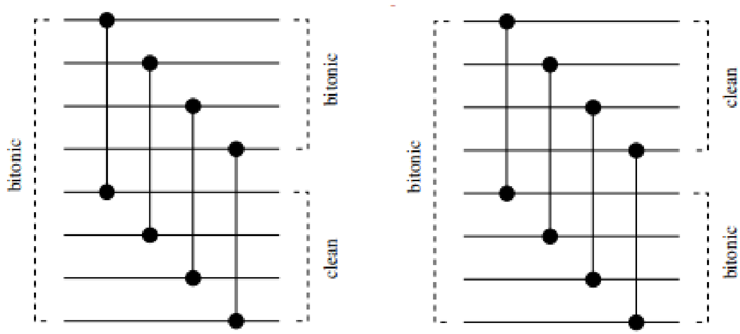
1.8.3 Bitonic sorting network

Step 1: construct a "bitonic sorter"; it sorts any bitonic sequence

For 0-1 sequences- which we can focus on - bitonic sequences have the form

$$0^i 1^j 0^k \quad \text{or} \quad 1^i 0^j 1^k$$

1.8.3.1 Step 0: Half cleaner



(note: depth=1)

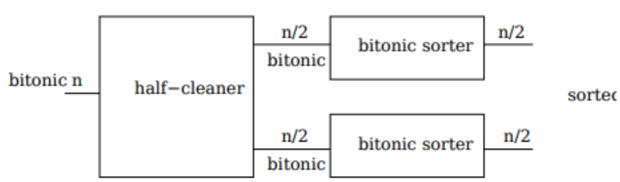
If clean in lower part: all 1s
If clean in upper part: all 0s

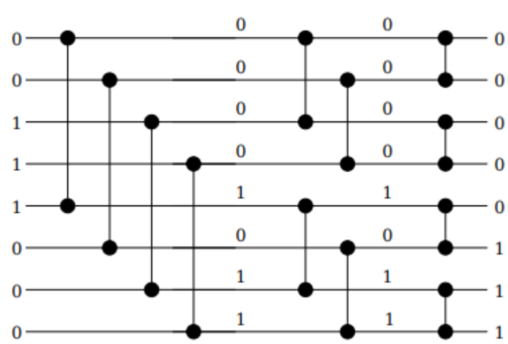
Lemma:
If the imput to a half-cleaner is a bitonic 0-1 sequence, then for output:

- Both top and bottom half are bitonic
- every element in top half is $\leqslant$ any element in bottom half
- at least one of the halves is clean - all 0's or all 1's

Proof:
Sorter is comparing nth element in 1st half with nth element in 2nd half repeatedly for all elements

1.8.3.2 Step 1: Bitonic Sorter





Depth: $D(n) = D(n/2) + 1,\ D(2) = 1 \implies D(n) = \log n$

Using the bitonic sorters on the clean version, while not actually doing anything, keeps the data in the correct place. Supposing the converse where the bottom half was clean with 1s, they would all be in the correct place and concatenating it with the output from the bitonic sorter gives a list

1.8.3.3 Step 2: A merging network

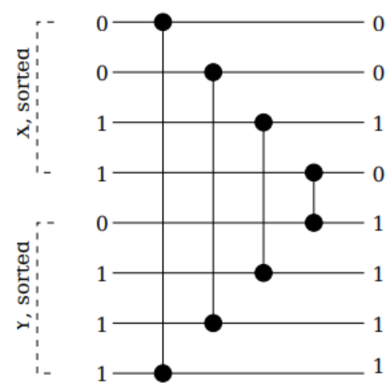
Merges 2 sorted sequences. Adapt a half-cleaner.

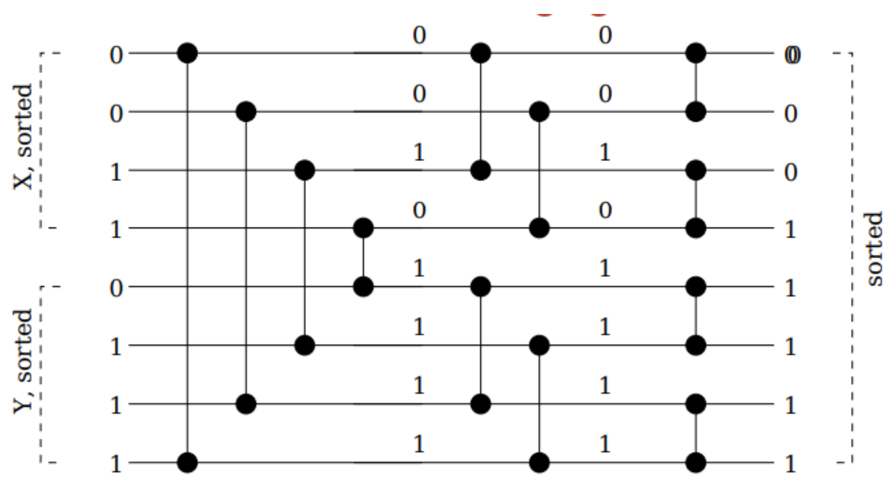
Idea: given 2 sorted sequences, reverse second one, concatenate with the first ⇒ bitonic

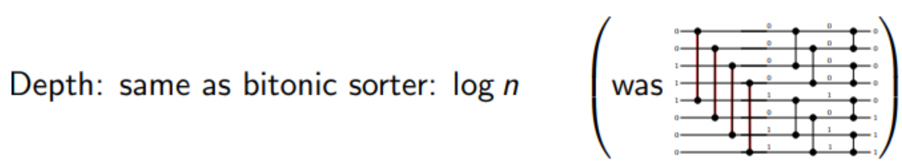
Example:

$$X = 0011 \quad Y = 0111 \quad Y^R = 1110 \quad XY^R = 00111110$$

- So can merge X and Y by doing bitonic sort on X and Y^R
- Don't explicitly reverse Y, instead reverse the bottom half of the connections of the first half-cleaner

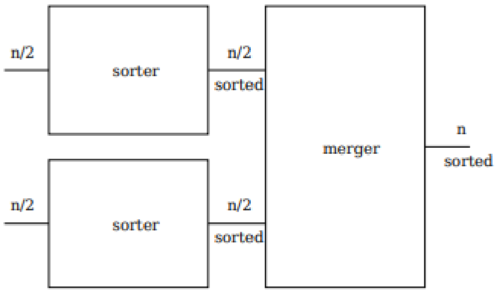


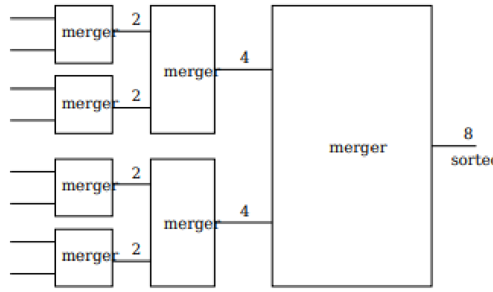


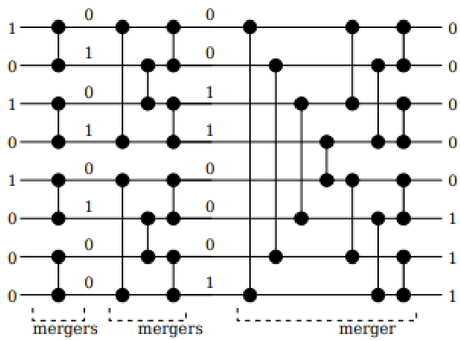


1.8.3.4 Step 3: Asorting network

Recursive merging - like merge sort, bottom up:







You can see that this just keeps recursing down the size of the mergers

Depth:

$$D(n) = D(n/2) + \log n, D(2) = 2 \Rightarrow D(n) = \Theta(\log^2 n)$$

Use 0-1 principle to prove that this sorts all inputs

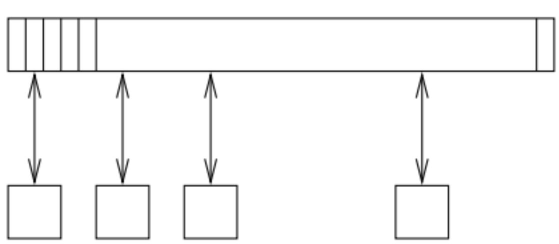
Prove by induction as it has recursive construction.
Base case: comparators sort two numbers.
Need to prove mergers do what they are supposed to do. - recursive construction so use induction again
Need to prove half cleaners do what they are supposed to do - proof already shown via lemma.

2 Shared Memory - NOT EXAMINABLE WOOOOOO

2.1 Parallel Random Access Machines (PRAMs)

Basically, that's what you get when you dream von Neumann's dream in parallel.

Bunch of synchronous processors (with little local memory), shared memory; in each step, each processor can access a memory cell in unit time (read or write), or perform local computation



Various models:

- EREW (exclusive read, exclusive write)
- CREW (concurrent read, exclusive write)

- CRCW (concurrent read, concurrent write)

Here: problems with concurrent writes

- **common**: all the same
- **priority**: processors are sorted by priority; if more than one want to write, highest priority wins
- **arbitrary**: arbitrary one wins
- **majority**: many sub models

Obviously: $EREW \leqslant CREW \leqslant CRCW$

PRAMs in general, and CRCWs in particular, are theoretical models. No real success with building a large one

They are, however , extremely nice when one wants to develop parallel algorithms without having to care about, say, network structure

The developer can concentrate on algorithmic aspects, rather than messy technical details

There are a few ways to (more or less) automatically transform PRAM algorithms into ”real” ones

Before we can see out first ”real” PRAM algorithm, here’s the basic extra programming construct

for $P_i, 1 \leqslant i \leqslant n$ **in parallel do**
    processor P_i does something
end for

Example
Suppose we have processors $P_1 \dots P_n$, and memory cells, say $A(1), A(2), \dots$

for $P_i, 1 \leqslant i \leqslant n$ **in parallel do**
    A(i)← i
end for

2.2 Parallel Mergesort

Suppose you have an input $S = (s_1 \dots s_n)$, n power of two (just for presentation)

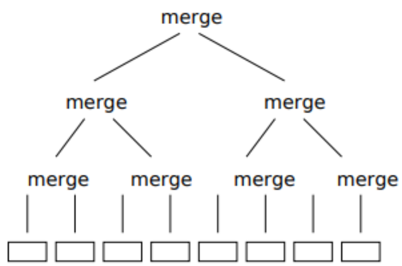
Suppose S pairwise distinct (also no real restriction as can replace s_i by (s_i, p_i) with p_i ID of ith processor)

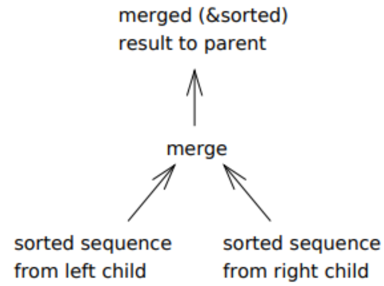
Sequence MergeSort (Sequence S)

if |S|⩽ 1 **then**
    return S
end if
Split S into two sequences S_1, S_2 of equal size
$R_1 \leftarrow MergeSort(S_1)$
$R_2 \leftarrow MergeSort(S_2)$
return Merge(R_1, R_2)

As so often, can visualise idea of algorithm using complete binary tree of height $\log n$ (i.e., n leaves)

- ith leaf stores ith input element s_i
- each internal node (non leaf) receives sorted sequence from each of its two children and merges them
- this, each internal node v is responsible for sorting the leaves in the sub-tree underneath, i.e., in sub tree with root v





Let $M(\ell)$ demote time needed to merge two sorted sequences of length ℓ each

An algorithm that does **all the mergers on one level in parallel** would then need time

$$\sum_{i=0}^{\log(n)-1} M\left(2^i\right) = O(M(n)\log n)$$

Now, what $M(n)$ can we come up with **Definition**

- Let $R_1\&R_2$ denote result of MERGE(R_1, R_2)
- **Rank:** Let S be a sequence and x be an arbitrary element, not necessarily $\in$ S. Then Rank(x,S) is the number of elements from S that are strictly smaller than x
- **Cross rank:** Let S,T be sorted sequences, $S = (s_1 \dots s_n)$. Let

$$R[S,T] : S \to \mathbb{N} \text{ with } R[S,T]\,(s_i) = \text{Rank}\,(s_i, T)\,, 1 \leq i \leq n$$

  We use $R[S,T]$ as vector $(\text{Rank}\,(s_1,T)\,, \dots, \text{Rank}\,(s_n,T))$

Example:

- S=(1,3,7,9,11), then Rank(8,S)=3 and Rank (3,S)=1

2.2.1 Lemma

Let S,T be two sorted sequences with $S \cap T = \emptyset$. Then we can compute S&T on an $O(|S|+|T|)$-processor CREW-PRAM in time $O(\log(|S|+|T|))$

2.2.2 Corollary

With —S—=—T—=n we get time $M(n) = O(\log(n))$

2.2.3 Proof of lemma

Let $S = (s_1 \dots s_n)$. s_i must appear in S&T at position $Rank(s_i, S\&T)$. Since $S \cap T = \emptyset$ we have

$$\text{Rank}\,(s_i, S\&T) = \text{Rank}\,(s_i, S) + \text{Rank}\,(s_i, T) = (i-1) + \text{Rank}\,(s_i, T)$$

T is sorted, this can find $Rank(s_i, T)$ in time $O(\log|T|)$ using binary search. If we've got one processor for each element of S when know $Rank(s_i, S\&T)$ for every $s \in S$ in time $O(\log|T|)$

Do the same for elements $t_i \in T$

2.2.4 Theorem

$O(n)$ -processor CREW-PRAM can sort n number in time $O((\log n)^2)$ time