



COSC 6365
Lecture 16
2008-03-06

CS@UH

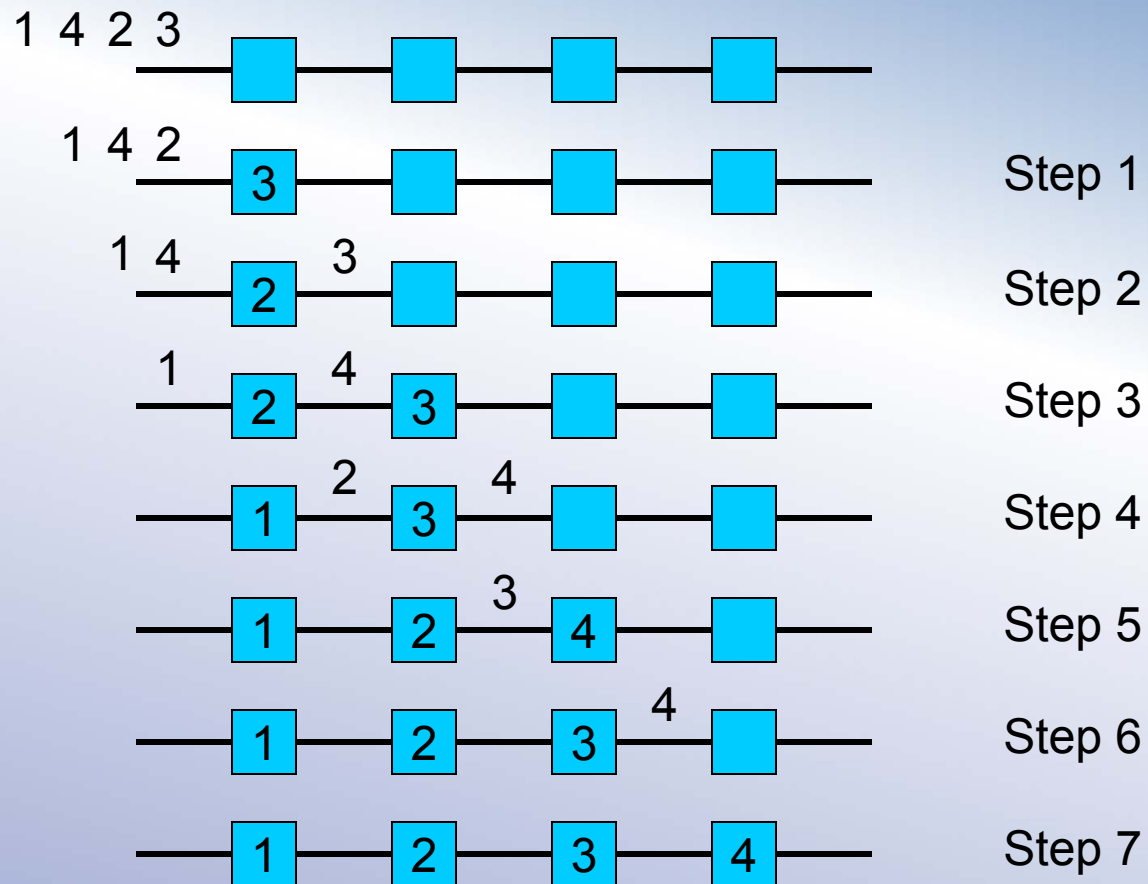
Introduction to HPC

Lecture 16

Lennart Johnsson
Dept of Computer Science
Director TLC²



Up-Down sorter





Up-Down sorter

Phase 1: Enter and sort values

1. Accept a value from the left neighbor
2. Compare the input value with the value retained from the previous comparison
3. Output the larger value to the right neighbor and store the smaller value

Phase 2: Output sorted values

1. Output the stored value to the left neighbor
2. Accept and store a value received from the right neighbor

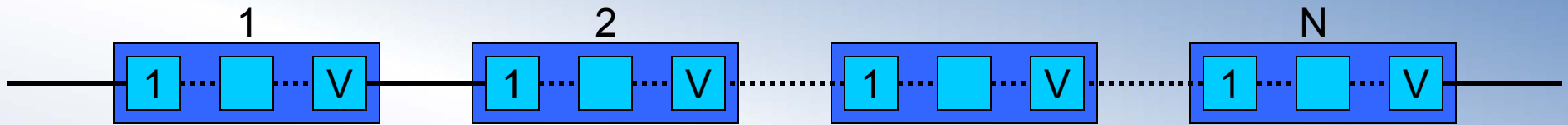
Time: $2N$

Speed-up: $\frac{O(N \log_2 N)}{2N} = O(\log_2 N)$

Efficiency: $\frac{O(\log_2 N)}{N}$



Limited Parallelism - Emulation



$$\text{Time: } 1, 2, 3, \dots, V, V, \dots, V, \dots, 3, 2, 1 = [V(V+1)/2 + (R-V)V]N = V(2R-V+1)$$

$$\text{Speedup: } \frac{O(R \log_2 R)}{(R/N)(2R - (R/N) + 1)} = \frac{O(N \log_2 R)}{2R - (R/N) + 1}$$

$$\text{Efficiency: } \frac{O(\log_2 R)}{2R - (R/N) + 1}$$

Emulation is a simple way of mapping fine grain algorithms onto limited resources. However, in this case the efficiency decreases with N



COSC 6365
Lecture 16
2008-03-06

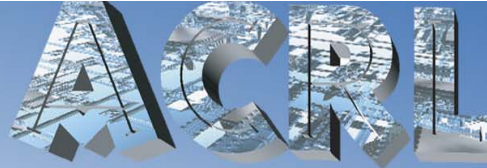
CS@UH

Comparison

Comparison - Serial

```
FOR j = k-1 STEP -1 UNTIL (input-bit ≠ stored-bit) DO  
  Output the input bit  
ENDFOR  
IF input-bit > output-bit THEN  
  Output the remaining input bits  
ELSE  
  Output the remaining memory-bits and  
  Store the remaining input-bits in place of  
  the memory-bits being output  
ENDIF
```

Sorting time actually $O(kN)$



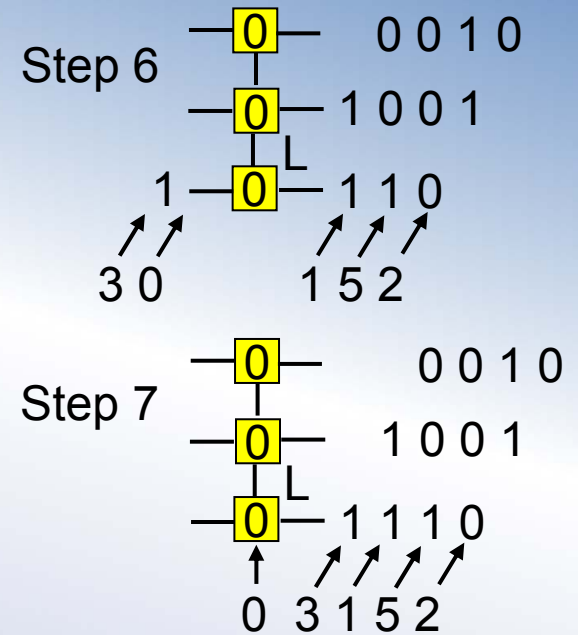
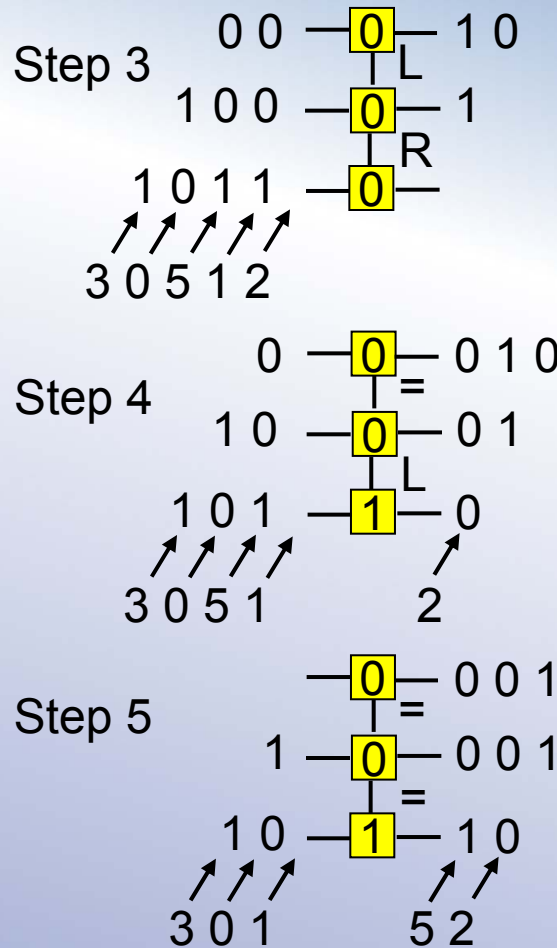
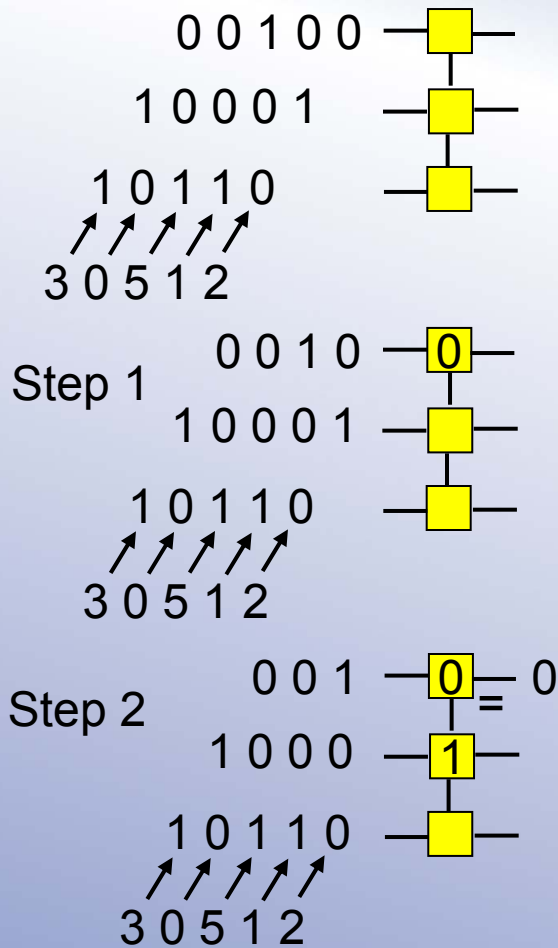
ADVANCED COMPUTING RESEARCH LABORATORY

COSC 6365
Lecture 16
2008-03-06



Comparison

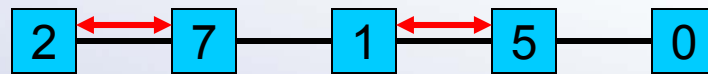
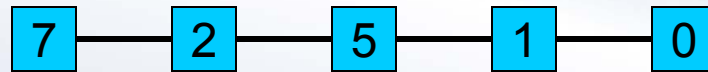
Parallel/pipelined





1-D array sort

Odd-even transposition sort



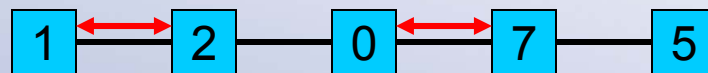
Step 1

Time: N



Step 2

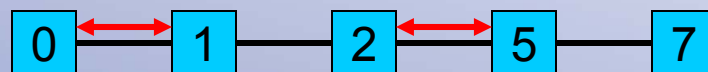
Speed-up: $O(\log_2 N)$



Step 3



Step 4



Step 5

Efficiency: $O(\log_2 N)/N$



0-1 Sorting Lemma

The 0-1 Sorting Lemma. If an oblivious comparison-exchange algorithm sorts all sets of 0s and 1s, then it also sorts all input sets of arbitrary values.

Proof: Assume all sequences of 0 and 1 are sorted, but there exist one sequence x_1, x_2, \dots, x_N that isn't sorted correctly. Assume the permutation π sorts the sequence x in non-descending order, i.e., $x_{\pi(1)}, x_{\pi(2)}, \dots, x_{\pi(N)}$, and let the actual permutation from attempting sort be ψ , i.e., the actual result is $x_{\psi(1)}, x_{\psi(2)}, \dots, x_{\psi(N)}$. Since x isn't sorted there exists a k such that for $i < k$ $x_{\pi(i)} = x_{\psi(i)}$ and $x_{\pi(k)} \neq x_{\psi(k)}$, in fact $x_{\psi(k)} > x_{\pi(k)}$. Hence, there exist an $r > k$ such that $x_{\psi(r)} = x_{\pi(k)}$.

Now, define a 0-1 sequence y_i such that $y_i = 1$ if $x_i > x_{\pi(k)}$, otherwise $y_i = 0$. Since $x_i \geq x_j$ implies $y_i \geq y_j$ applying the sorting algorithm to y yields the same permutation as applying it to x . Hence, the result is

$y_{\psi(1)}, y_{\psi(2)}, \dots, y_{\psi(k-1)}, y_{\psi(k)}, y_{\psi(k+1)}, \dots, y_{\psi(r-1)}, y_{\psi(r)}, y_{\psi(r+1)}, \dots, y_{\psi(N)}$, or
 $0, 0, \dots, 0, 1, \dots, 0, \dots$

This sequence is clearly not sorted. Contradiction.



Limited parallelism

Sorting R elements on N processors

FOR J=1 TO N/2 DO

Processor i merges its sorted sequence with processor i+1 for odd i, keeping the smaller half of the merged sequences and passing the larger half to processor i+1.

Processor i merges its sorted sequence with processor i+1 for even i, keeping the smaller half of the merged sequences and passing the larger half to processor i+1.

Time: $O((R/N)\log_2(R/N))+(2(R/N)-1)N$

Note:

N=R: Time = R

N= 1: Time = $O(R\log_2R)$

Speed-up: $\frac{O(R\log_2R)}{O((R/N)\log_2(R/N))+(2R-N)}$

Efficiency: $\frac{O(R\log_2R)}{O(R\log_2(R/N))+N(2R-N)}$

N=R: Eff. = $O(\log_2R)$

N= 1: Eff. = $O(1)$



2-D array sort – shear sort

FOR I=0 **TO** $\log_2 \sqrt{N}-1$ **DO**

Sort even rows in non-descending left-to-right order

Sort odd rows in non-descending right-to-left order

Sort all columns in a non-descending order from top to bottom

ENDFOR

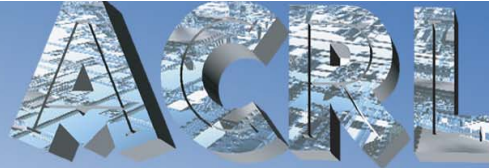
Sort even rows in non-descending left-to-right order

Sort odd rows in non-descending right-to-left order

Time: $\sqrt{N}(\log_2 N + 1)$

Speed-up: $\frac{O(N \log_2 N)}{\sqrt{N}(\log_2 N + 1)} = O(\sqrt{N})$

Efficiency: $\frac{O(\sqrt{N})}{N} = \frac{1}{O(\sqrt{N})}$



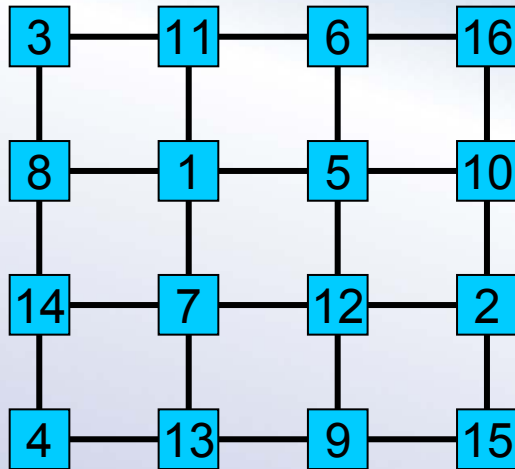
ADVANCED COMPUTING RESEARCH LABORATORY

COSC 6365
Lecture 16
2008-03-06

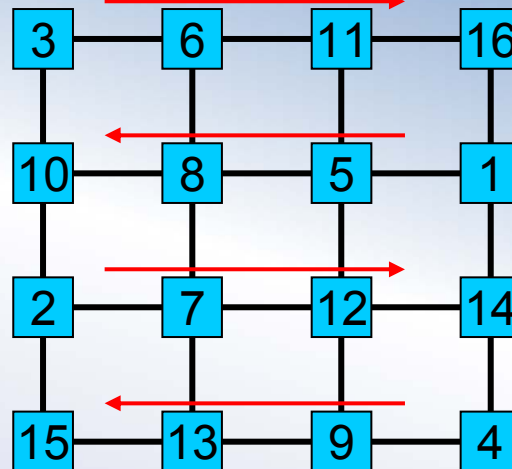


Shear Sort

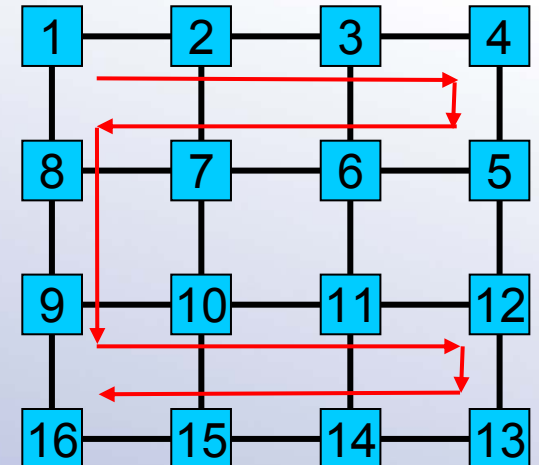
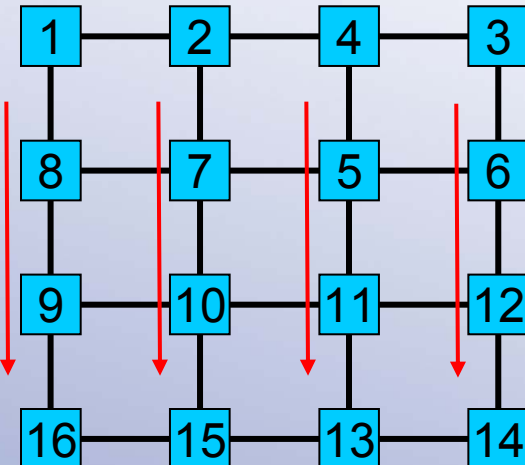
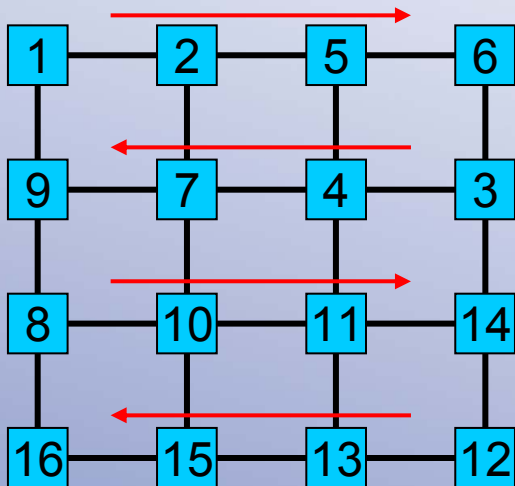
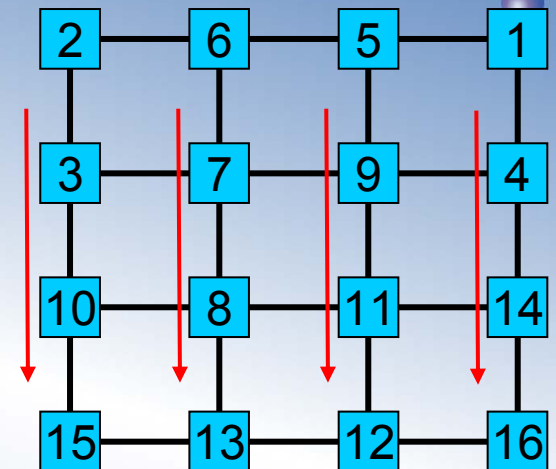
Initial allocation



Row sort in alternating order



Column sort



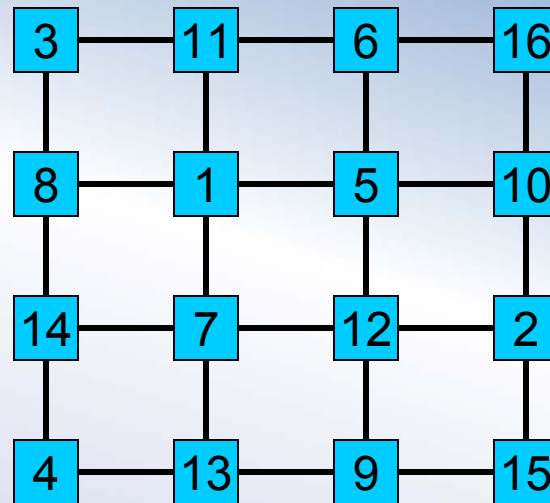
Row sort in alternating order

Column sort

$2\sqrt{N}$ steps of snake sort



Shear sort – correctness proof



0 0 0 0 0 1 1 1
 1 1 1 1 0 0 0 0

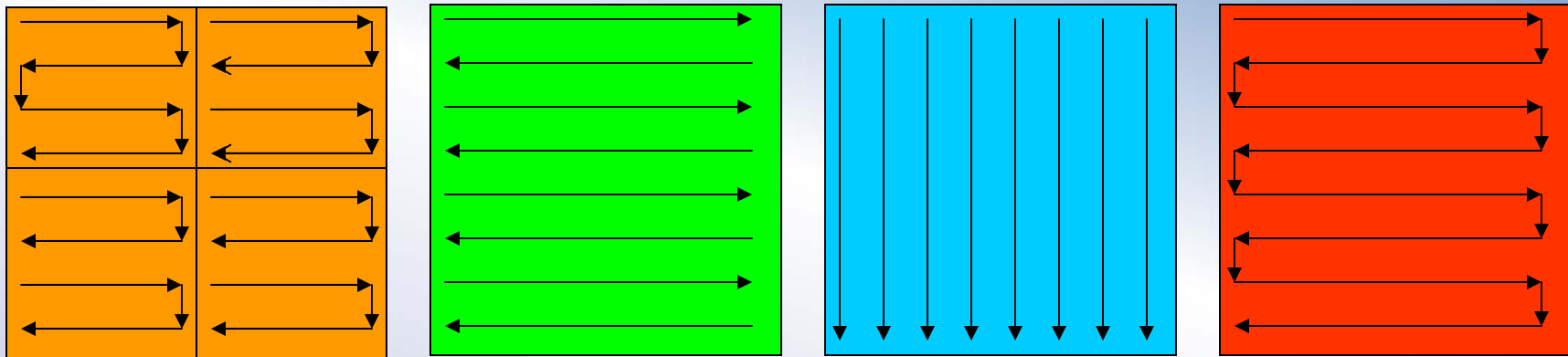
One row in a pair becomes clean after the column sort, or at least half the rows become clean.

At most one dirty row after $\log_2 N$ iterations!

One row sort cleans it!



Optimal Shear sort

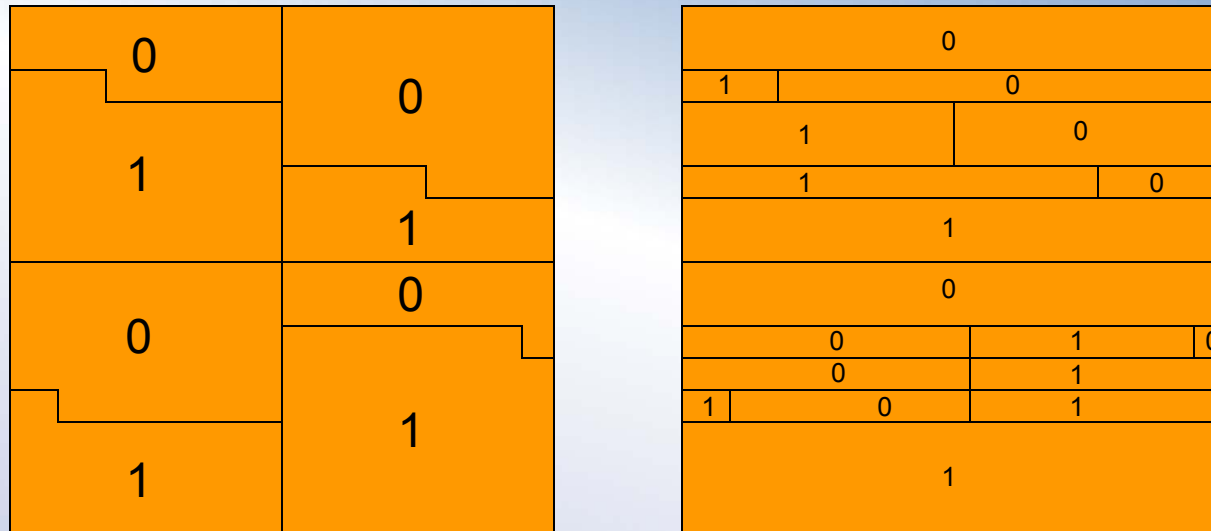


1. Recursively sort each quadrant of the array into snake-like order
2. Sort rows in alternate order
3. Sort the columns
4. Perform $2\sqrt{N}$ steps of odd-even transposition sort along the snake



Correctness proof

Result of quadrant snake sort



Case 1: Even number of 50/50 rows

At most one dirty row from the top half,
one from the bottom half

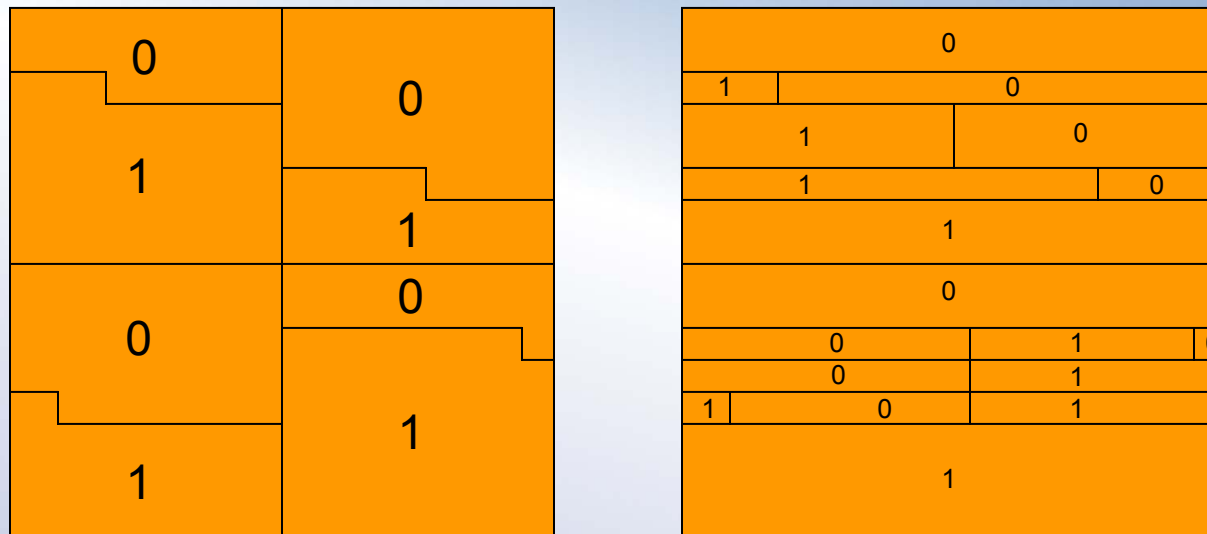
The two dirty rows are adjacent after the
column sort.

Snakesort with steps equal to two rows
cleans one of these two rows.

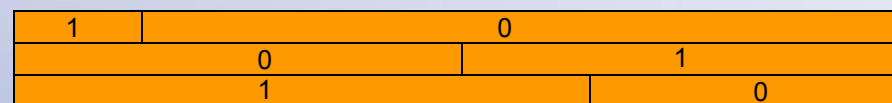


Correctness proof

Result of quadrant snake sort



Case 2: Odd number of 50/50 rows

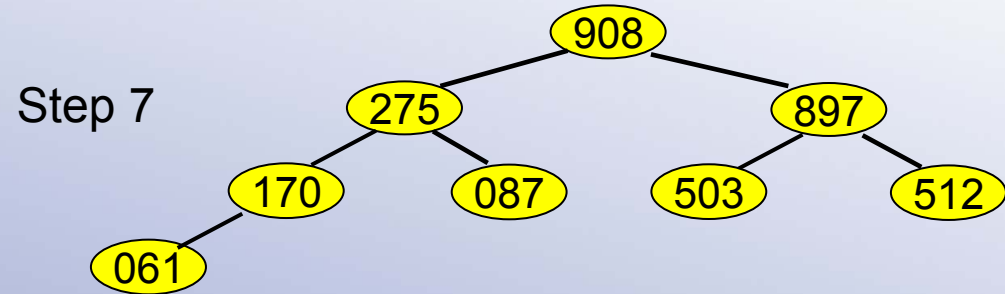
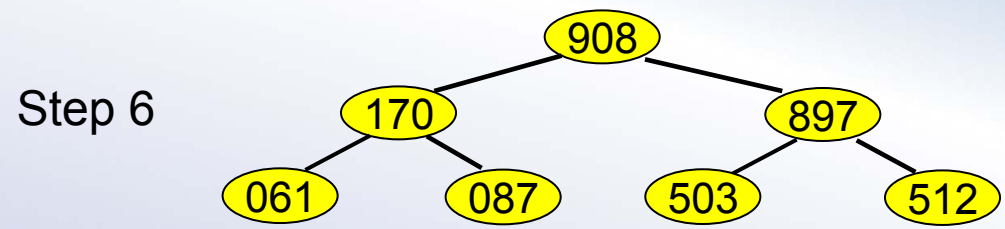
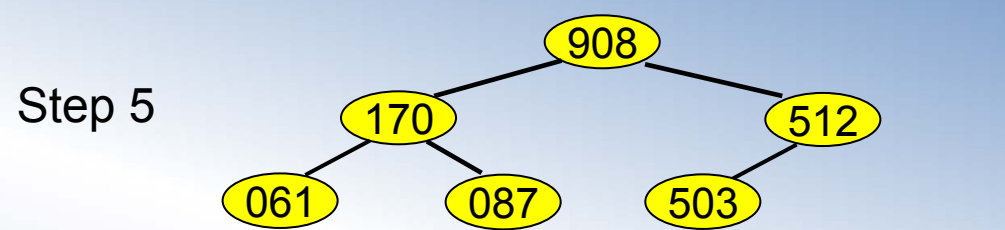
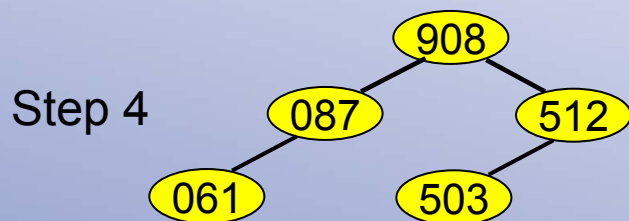
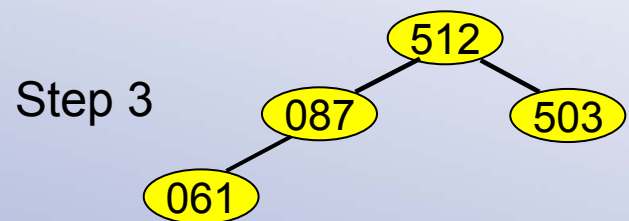


At most one dirty row from the top half,
one from the bottom half



Heap sort

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

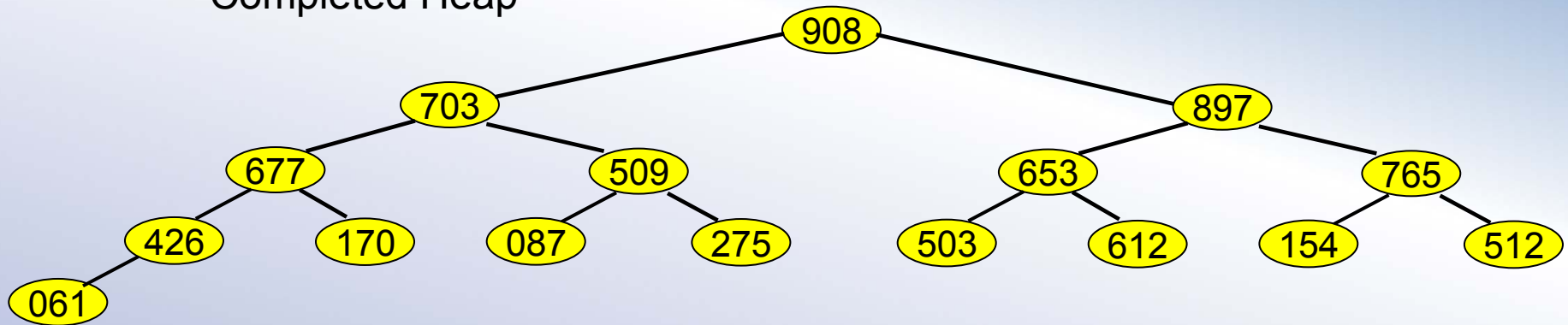




Heap sort

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

Completed Heap



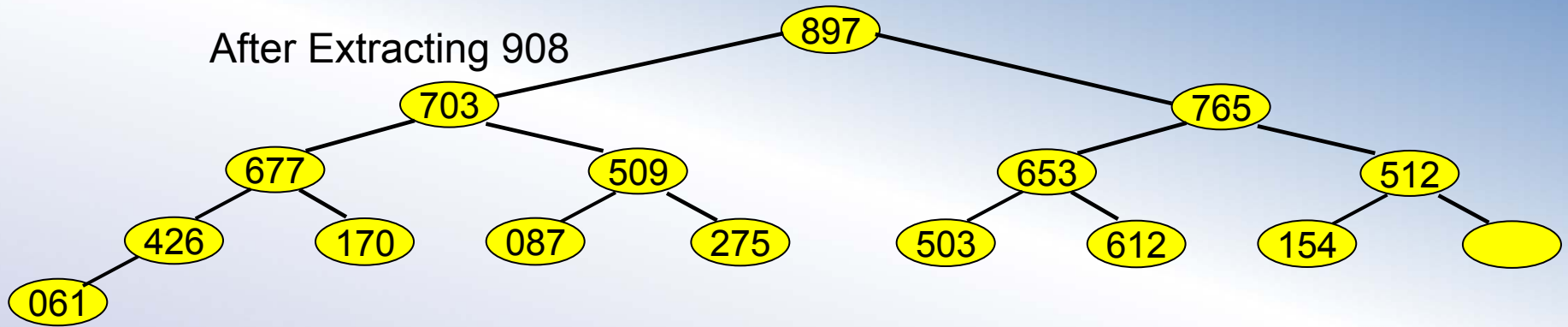
Time: N



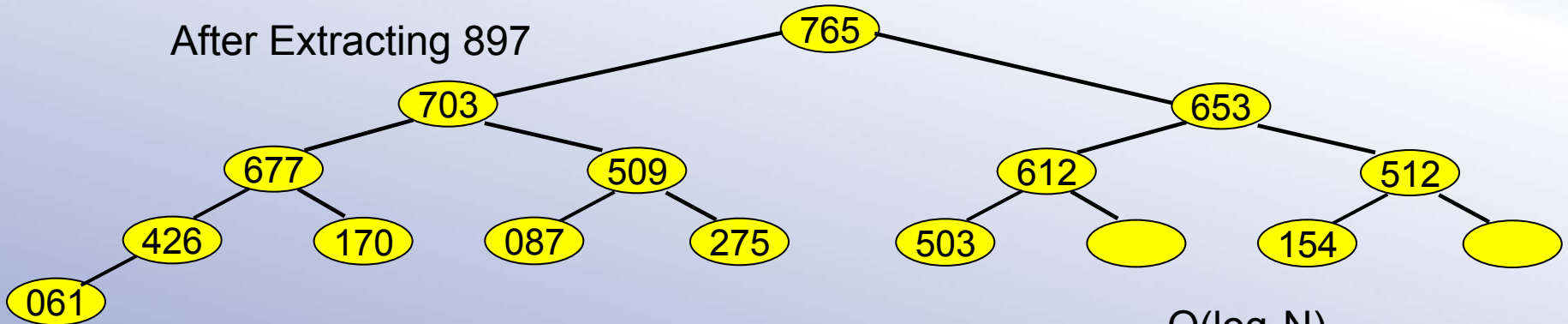
Heap sort

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

After Extracting 908



After Extracting 897



Total time: N

Speed-up: $O(\log_2 N)$

Efficiency: $\frac{O(\log_2 N)}{N}$