

Sorting in Linear Time

Contents

- **Lower bounds for sorting**
- **Counting sort**
- **Radix sort**

Lower bounds for sorting

• Comparison sorts

- Sorting algorithms using only comparisons to determine *the sorted order of the input elements*.
- Use tests such as $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$.
- Heapsort, Mergesort, Insertion sort, Selection sort, Quicksort

• Lower bounds for (comparison) sorting

- Any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort n elements.

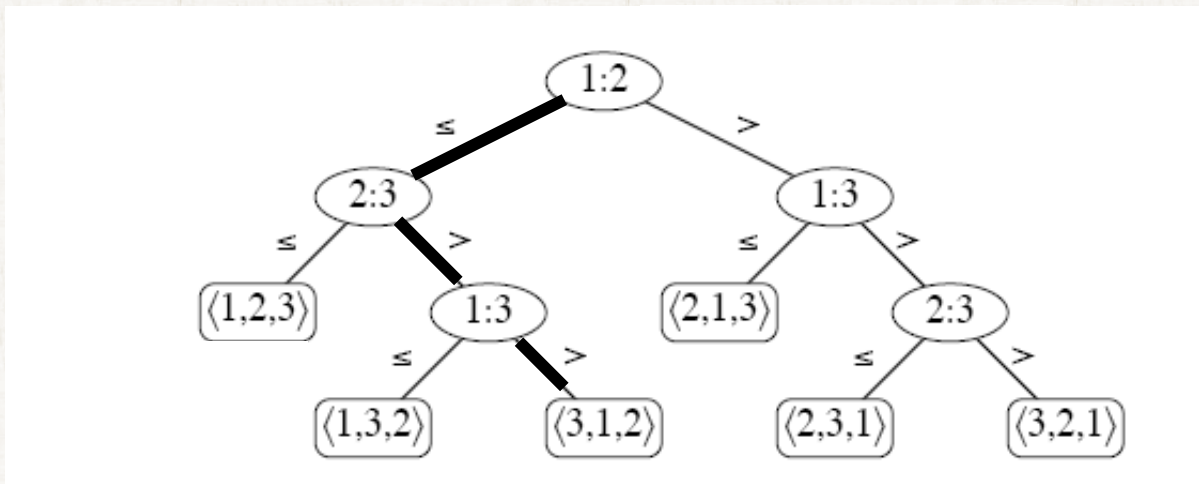
Lower bounds for sorting

• Comparison sort

- we assume without loss of generality that all of the input elements are distinct.
 - The comparisons $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$, and $a_i < a_j$ are all equivalent.
 - We assume that all comparisons have the form $a_i \leq a_j$

The decision-tree model

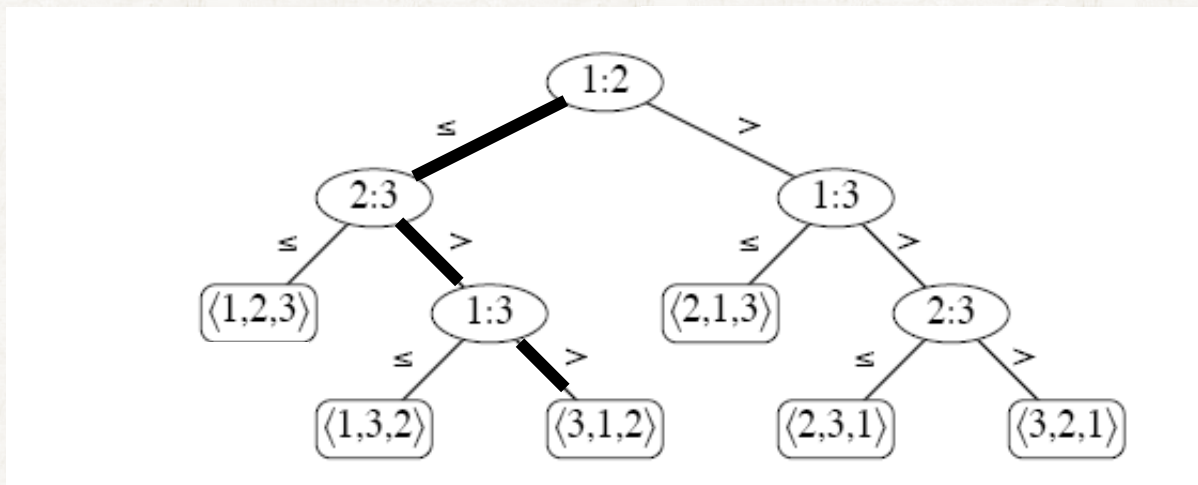
- Comparison sorts can be viewed in terms of *decision trees*.
 - A full binary tree.
 - Each leaf is a permutation of input elements.
 - Each internal node $i:j$ indicates a comparison $a_i \leq a_j$.



A decision tree for insertion sort

The decision-tree model

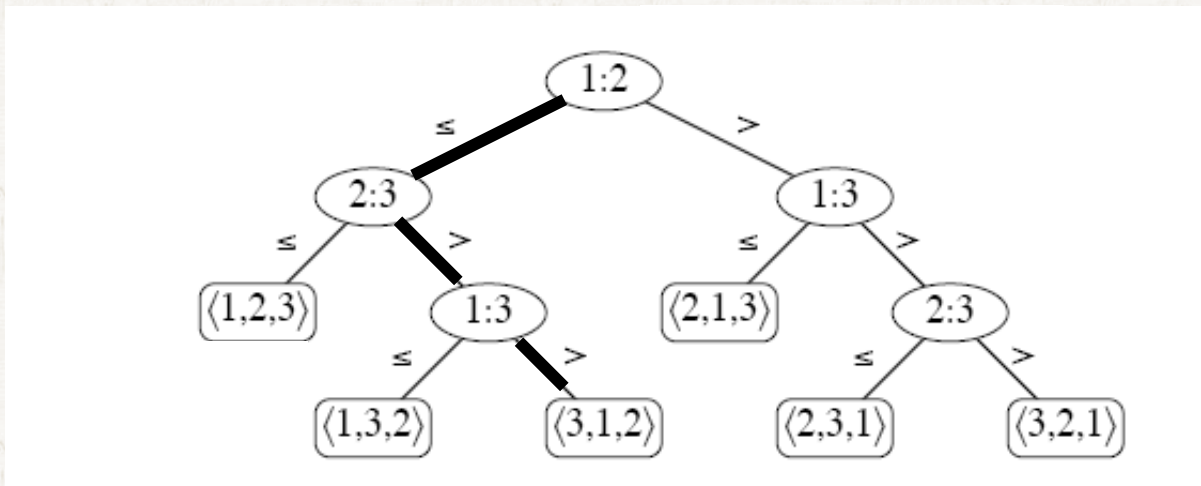
- The left subtree of the node $i:j$ includes all permutations for $a_i \leq a_j$.
- The right subtree includes all permutations for $a_i > a_j$.



A decision tree for insertion sort

The decision-tree model

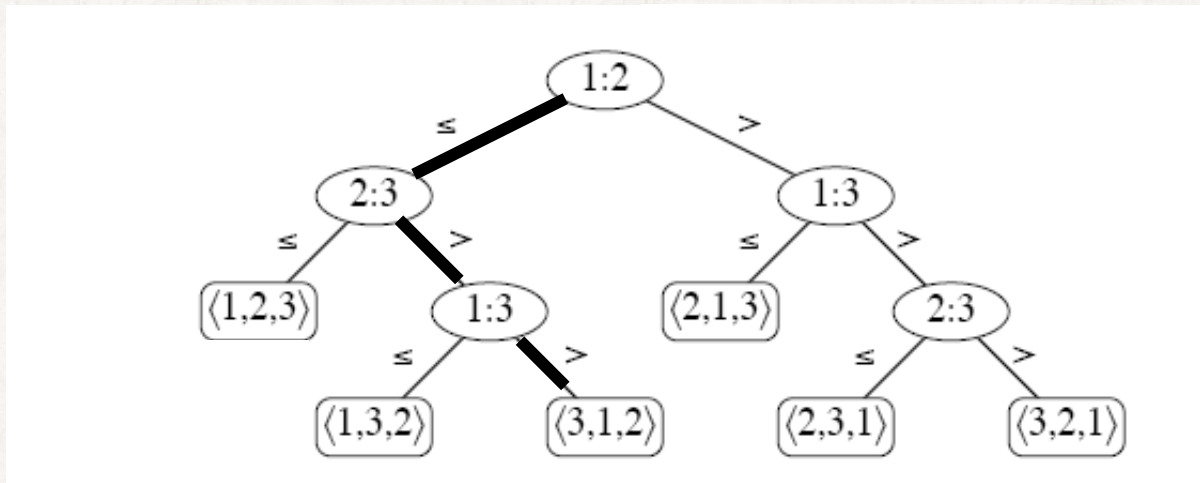
- The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf.



A decision tree for insertion sort

The decision-tree model

- the worst-case number of comparisons
= the height of its decision tree.



A decision tree for insertion sort

The decision-tree model

- **Theorem 8.1:** Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.
- **Proof:**
 - Height: h , Number of element: n
 - The number of leaves: $n!$
 - Each permutations for n input elements should appear as leaves.
 - $n! \leq 2^h$
 - $\lg(n!) \leq h$
 - $\Omega(n \lg n)$ (by equation (3.19) : $\lg(n!) = \Theta(n \lg n)$).

Self-study

- **Exercise 8.1-1**

- The smallest depth of a leaf in a decision tree

- **Exercise 8.1-3**

- Decision tree existence

- **Exercise 8.1-4**

- Lower bound of a decision tree

Counting sort

● Counting sort

- A sorting algorithm using *counting*.

A

0	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---

B

0	0	0	1	1	1	1	1
---	---	---	---	---	---	---	---

- Each input element x should be located in the i th place after sorting if the number of elements less than x is $i-1$.

Counting sort

	1	2	3	4	5	6	7	8
<i>A</i>	2	5	3	0	2	3	0	3

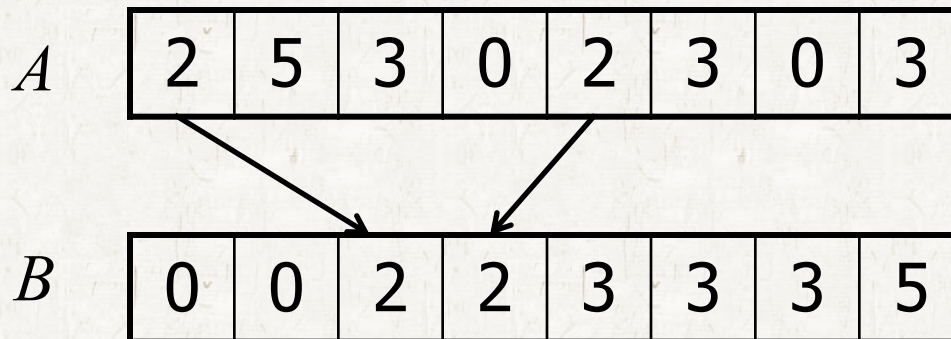
	0	1	2	3	4	5
<i>C</i>	0	0	0	0	0	0

<i>B</i>	0	0	2	2	3	3	3	5
----------	---	---	---	---	---	---	---	---

Counting sort

Counting sort

- Stable
 - Same values in the input array appear in the same order in the output array.



Counting sort

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

C'	2	2	4	7	7	8
------	---	---	---	---	---	---

Counting sort

	1	2	3	4	5	6	7	8
<i>A</i>	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
<i>C'</i>	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
<i>B</i>							3	

	0	1	2	3	4	5
<i>C'</i>	2	2	4	6	7	8

	1	2	3	4	5	6	7	8
<i>B</i>		0					3	

	0	1	2	3	4	5
<i>C'</i>	1	2	4	6	7	8

	1	2	3	4	5	6	7	8
<i>B</i>		0				3	3	

	0	1	2	3	4	5
<i>C'</i>	1	2	4	5	7	8

Counting sort

COUNTING-SORT(A, B, k)

$\Theta(k)$ $\left[\begin{array}{l} 1 \text{ for } i = 0 \text{ to } k \\ 2 \quad C[i] = 0 \end{array} \right.$

$\Theta(n)$ $\left[\begin{array}{l} 3 \text{ for } j = 1 \text{ to } A.length \\ 4 \quad C[A[j]] = C[A[j]] + 1 \end{array} \right.$

5 $\triangleright C[i]$ contains the number of elements equal to i .

$\Theta(k)$ $\left[\begin{array}{l} 6 \text{ for } i = 1 \text{ to } k \\ 7 \quad C[i] = C[i] + C[i - 1] \end{array} \right.$

8 $\triangleright C[i]$ contains the number of elements less than or equal to i .

$\Theta(n)$ $\left[\begin{array}{l} 9 \text{ for } j = A.length \text{ downto } 1 \\ 10 \quad B[C[A[j]]] = A[j] \\ 11 \quad C[A[j]] = C[A[j]] - 1 \end{array} \right.$

Counting sort

- The overall time is $\Theta(k+n)$ where k is the range of input integers.
- If $k = O(n)$, the running time is $\Theta(n)$.

Self-study

- **Exercise 8.2-1**

- A counting-sort example

- **Exercise 8.2-3**

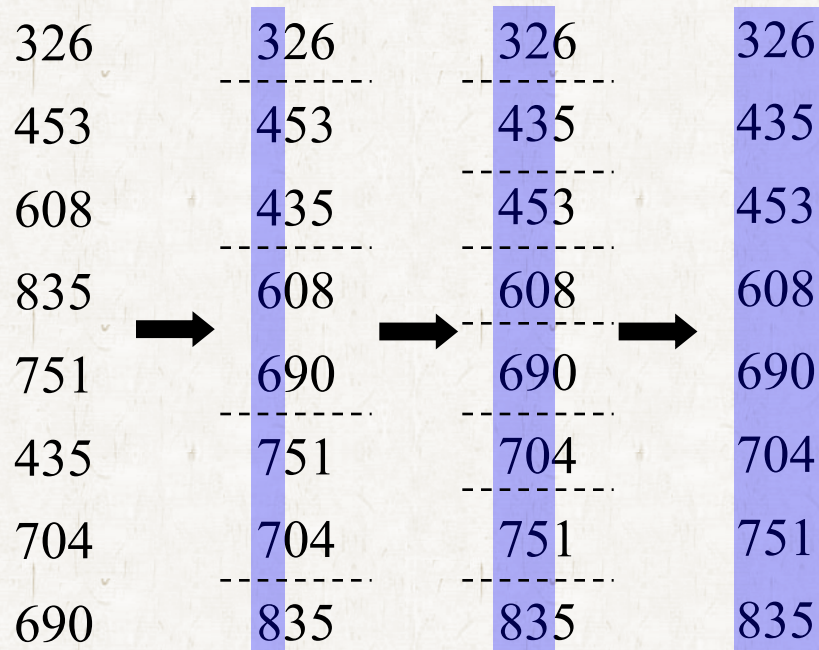
- Counting-sort stability

- **Exercise 8.2-4**

- A counting-sort application

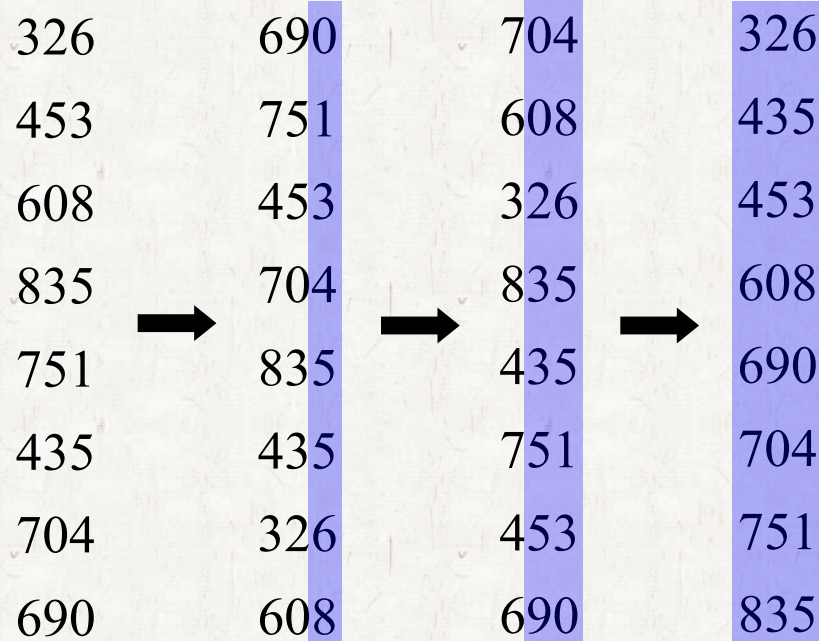
Radix sort

● Radix sort (MSB → LSB)



Radix sort

● Radix sort (MSB \leftarrow LSB)



Radix sort

RADIX-SORT(A, d)

1 for $i = 1$ to d

2 use a *stable sort* to sort array A on digit i

- RADIXSORT sorts in $\Theta(d(n + k))$ time when n d -digit numbers are given and each digit can take on up to k possible values.
- When d is constant and $k = O(n)$, radix sort runs in linear time.

Radix sort

Changing d and k

1326

4534

6018

8135

$d = ?$

$k = ?$

1326

4534

6018

8135

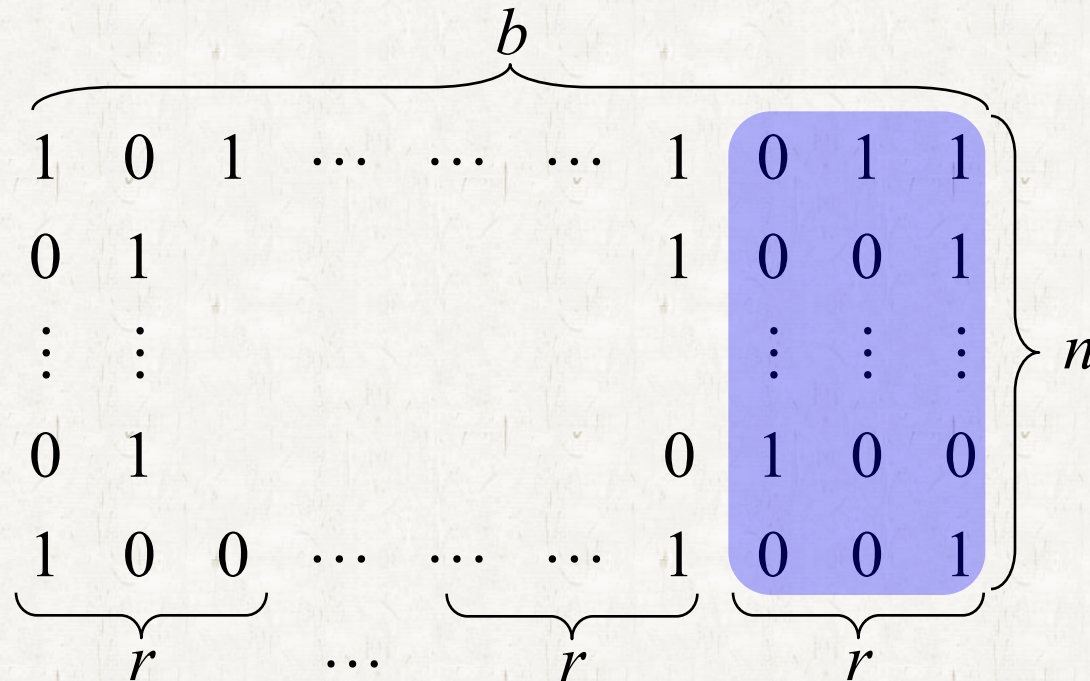
$d = ?$

$k = ?$

Radix sort

• **Lemma 8.4 (Self-study)**

Given n b -bit numbers and any positive integer $r \leq b$, RADIX-SORT correctly sorts these numbers in $\Theta((b/r)(n + 2^r))$ time.



Radix sort

- Computing optimal r minimizing $(b/r)(n + 2^r)$.

1. $b < \lfloor \lg n \rfloor$

for any value of r , $(n + 2^r) = \Theta(n)$ because $r \leq b$.

So choosing $r = b$ yields a running time : $(b/b)(n + 2^b) = \Theta(n)$, which is asymptotically optimal.

Radix sort

- Computing optimal r minimizing $(b/r)(n + 2^r)$.

2. $b \geq \lfloor \lg n \rfloor$

choosing $r = \lfloor \lg n \rfloor$ gives the best time to within a constant factor, $(b/\lg n)(n + 2^{\lg n}) = (b/\lg n)(2n) = \Theta(bn/\lg n)$.

- As we increase r above $\lfloor \lg n \rfloor$, the 2^r in the numerator increases faster than the r in the denominator.
- As we decrease r below $\lfloor \lg n \rfloor$, then the b/r term increases and the $n + 2^r$ term remains at $\Theta(n)$.

Radix sort

- Compare radix sort with other sorting algorithms.

- If $b = O(\lg n)$, we choose $r \approx \lg n$.

Radix sort: $\Theta(n)$

Quicksort: $\Theta(n \lg n)$

Radix sort

- The constant factors hidden in the Θ -notation differ.
 1. Radix sort may make fewer passes than quicksort over the n keys, each pass of radix sort may take significantly longer.
 2. Radix sort does not sort in place.

Self-study

- **Exercise 8.3-1**
 - Radix sort example
- **Exercise 8.3-2**
 - Stability
- **Exercise 8.3-4**
 - Radix sort application