Lower bound for sorting algorithms

- Best worst-case running time we've seen so far: $\Theta(n \lg n)$
- Is that the best we can get?
- Clearly we need $\Omega(n)$ because we must look at all of the input.
- It turns out that we can show that we need $\Omega(n \lg n)$ for any comparison based sorting algorithm.
- A comparison sorting algorithm uses only comparisons between elements to determine their relative order.
- Insertion Sort, Merge Sort, Quicksort, are comparison based sorting algorithms.

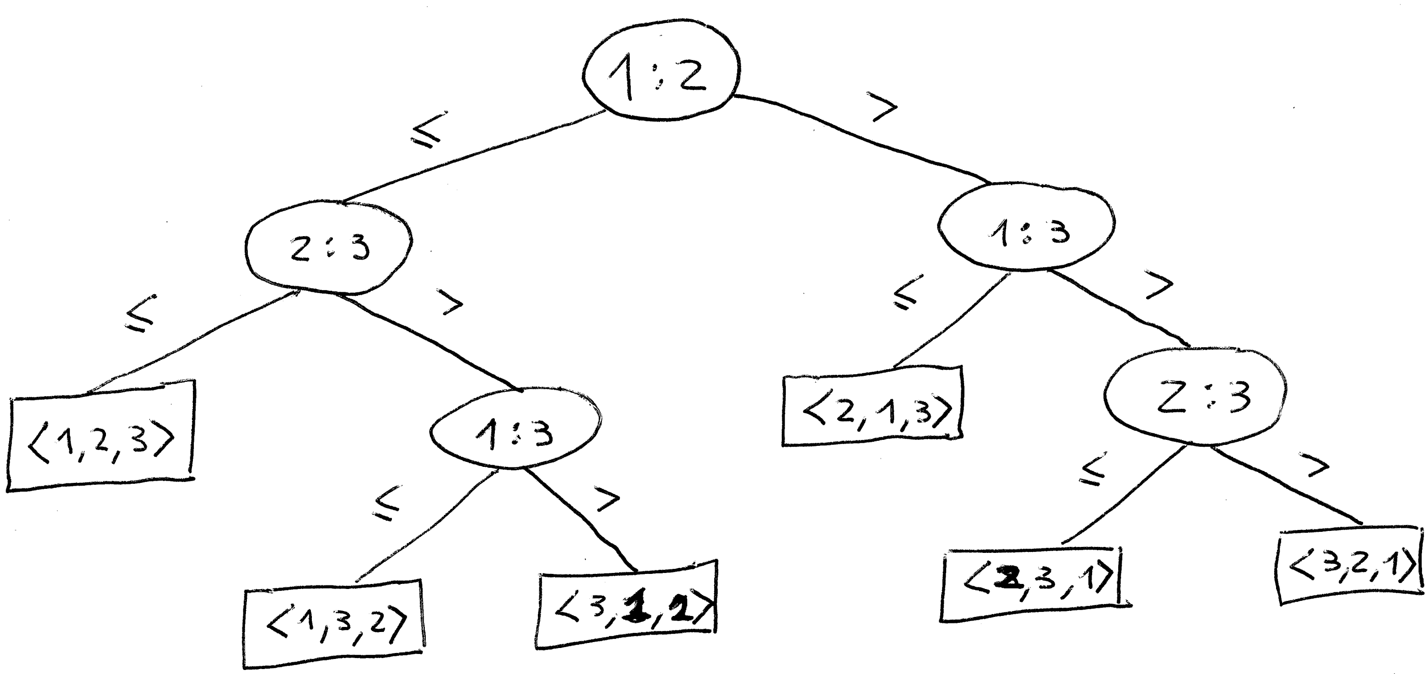
Decision-tree model

- Comparison sorting algorithms can be viewed in terms of decision trees.

- A decision tree is a full binary tree (every node is either a leaf or has two child nodes)

- The decision tree represents the comparisons that are performed by a particular sorting algorithm on inputs of a given size.

- Example with $n=3$:



- Each internal node is labeled $i:j$ and denotes a comparison between a_i and a_j
  - if $a_i \leq a_j$ the left subtree dictates further comparisons. Otherwise, the right subtree dictates further comparisions.

- Each leaf is labeled with a permutation of the input sequence.

  $\langle 2, 3, 1 \rangle$ means $a_2 \leq a_3 \leq a_1$

- The execution of the algorithm on a given input corresponds to the path taken from the root to a leaf
  - the comparisons made along the way are the ones that determined the ordering of the elements

Example

Sort $\langle a_1, a_2, a_3 \rangle = \langle 8, 3, 7 \rangle$

- Start at the root and compare a_1 with a_2 (8 with 3). $8 > 3 \Rightarrow$ move along right subtree

- Compare a_1 with a_3 (8 with 7).
  $8 > 7 \Rightarrow$ move along right subtree.

- Compare a_2 with a_3 (3 with 7).

  $3 \leq 7 \Rightarrow$ move along left subtree

- At this point we reach a leaf $\boxed{\langle 2, 3, 1 \rangle}$ which specifies the correct ordering of the input:

$$a_2 \leq a_3 \leq a_1$$

$$3 \leq 7 \leq 8$$

- worst case running time of algorithm
  = length of longest path from root to leaf for any given input

  = height of the tree

- We will now show that the height of the tree is $\Omega(n \lg n)$

  - The tree must have at least $n!$ leaves because there are $n!$ permutations of the input sequence.

  - Also, a binary tree of height h has at most 2^h leaves.

$$2^h \geq n!$$

$$\Longleftrightarrow \quad h \geq \lg(n!)$$

$$\geq \lg\left(\left(\tfrac{n}{e}\right)^n\right) \quad \text{// Stirling's formula}$$

$$= n \lg n - n \lg e$$

$$= \Omega(n \lg n)$$

Stirling's formula:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n}$$

where $\frac{1}{12n+1} < \alpha_n < \frac{1}{12n}$

$$\Rightarrow \quad n! \geq \left(\frac{n}{e}\right)^n$$

Sorting in linear time

- It's possible to sort in linear time if we use other things other than comparisons in order to determine the relative ordering of the input elements
- Resulting algorithms are not general
  $\Longrightarrow$ rely on assumptions about the input.

Counting Sort

- Assumption: input elements are integers in the range $1..k$

Input: $A[1..n]$ with each $A[i] \in \{1, 2, \dots, k\}$

Output: $B[1..n]$

Temporary storage: $C[1..k]$

- Basic idea: use array C to count the number of times each element occurs in the input sequence.

```
Counting-Sort(A, B, n, k)
let C[1..k] be a new array
for i = 1 to k
      C[i] = 0
for j = 1 to n
      C[A[j]] = C[A[j]] + 1          // C[i] = (# elems == i)
for i = 2 to k
      C[i] = C[i] + C[i-1]           // C[i] = (# elems ≤ i)
for j = n downto 1
      B[C[A[j]]] = A[j]
      C[A[j]] = C[A[j]] - 1
```

Example

$n = 6$
$k = 4$

A:

1	2	3	4	5	6
3	1	4	1	4	3

B:

C:

1	2	3	4

Just before the begining of the for loop in line 6:

C:

1	2	3	4
2	0	2	2

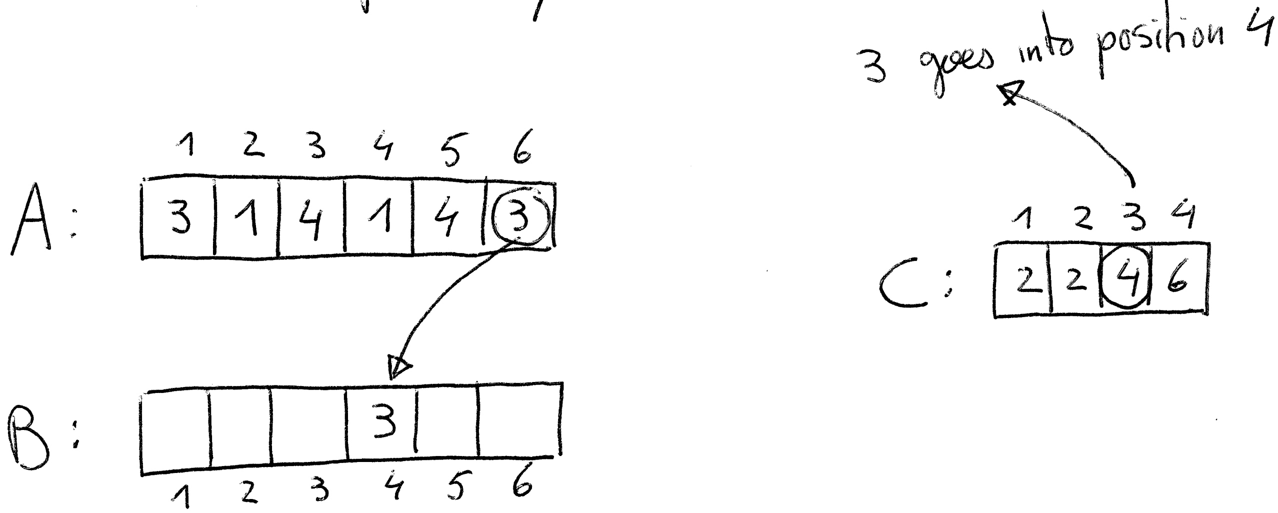
Just before the beginning of the for loop in line 8:

C:

1	2	3	4
2	2	4	6

6 elems ≤ 4
4 elems ≤ 3
2 elems ≤ 2
2 elems ≤ 1

Loop in line 8 puts the elements in their proper location in the output array B.

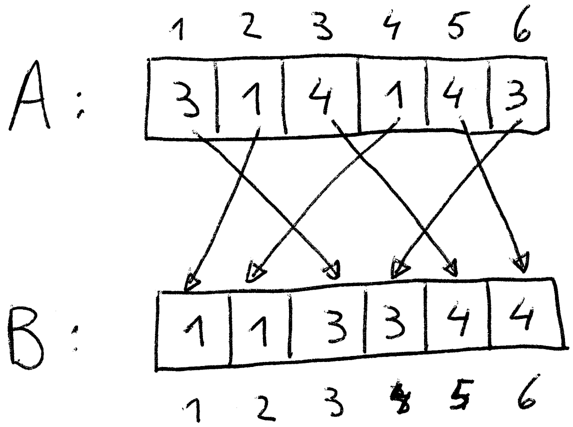


Updated C :

1	2	3	4
2	2	3	6

(next 3 will go into position 3)

and so on...



C:

1	2	3	4
2	2	4	6

$j=6$:	2	2	3	6
$j=5$:	2	2	3	5
$j=4$:	1	2	3	5
$j=3$:	1	2	3	4
$j=2$:	$\emptyset$	2	3	4
$j=1$:	$\emptyset$	2	2	4

Analysis of Counting Sort

1st for loop $\longrightarrow \Theta(k)$

2nd for loop $\longrightarrow \Theta(n)$

3rd for loop $\longrightarrow \Theta(k)$

4th for loop $\longrightarrow \Theta(n)$

Total: $\Theta(n+k)$

if k is $O(n)$ then Counting Sort is $\Theta(n)$

- Lower bound of $\Omega(n \lg n)$ for comparison sorting algorithms still valid.

- Notice that Counting Sort doesn't do a single comparison!

- Counting Sort is a <u>stable sorting</u> algorithm: it preserves the input order of elements with equal value.

    → this is an important property and allows it to be used as part of another sorting algorithm called Radix Sort.

Radix Sort

- Sort digit by digit using a stable sorting algorithm starting with the least significant digit

```
RADIX-SORT ( A, d )
  for i = 1 to d
      use a stable sort to sort array A on digit i
```

- Example:

329		720		720		329
457		355		329		355
657		436		436		436
839	→	457	→	839	→	457
436		657		355		657
720		329		457		720
355		839		657		839

- Can use induction on digit position to show that the algorithm is correct.

Analysis

Input: n d-digit numbers
each digit can take k possible values.

- $\Theta(n+k)$ for each digit sort using Counting Sort
- d digits $\longrightarrow$ $\Theta(d(n+k))$
- When d is a constant and $k = O(n)$, Radix Sort is $\Theta(n)$

Bucket Sort

- Assumes input is drawn from a uniform distribution over $[0, 1)$
- Algorithm has 4 steps:
  1) Divide $[0, 1)$ into n equal sized buckets.
  2) Distribute the n input elements into the buckets.
  3) Sort each bucket (e.g. with Insertion Sort)
  4) Output sorted buckets in order.
- Has similarities with hashing.

Bucket-Sort (A, n)

let $B[\emptyset .. n-1]$ be a new array

for $i = 1$ to $n-1$

  make $B[i]$ an empty list

for $i = 1$ to n

  insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$

for $i = \emptyset$ to $n-1$

  sort list $B[i]$ with Insertion Sort

concatenate $B[\emptyset], B[1], \ldots B[n-1]$ in order

return the concatenated list

Intuitive Analysis

- Avg number of elements per bucket is 1. Why?
- If each bucket has a constant number of elements then the time to sort each bucket is $\Theta(1)$

  $\Rightarrow$ time to sort n buckets is $\Theta(n)$

- Can make detailed analysis to show that the expected running time of Bucket Sort is $\Theta(n)$.

  See details in textbook.