Outline

Definition

A comparison based sorting algorithm sorts objects by comparing pairs of them.

Definition

A comparison based sorting algorithm sorts objects by comparing pairs of them.

Example

Selection sort and merge sort are comparison based.

Lemma

Any comparison based sorting algorithm performs $\Omega(n \log n)$ comparisons in the worst case to sort n objects.

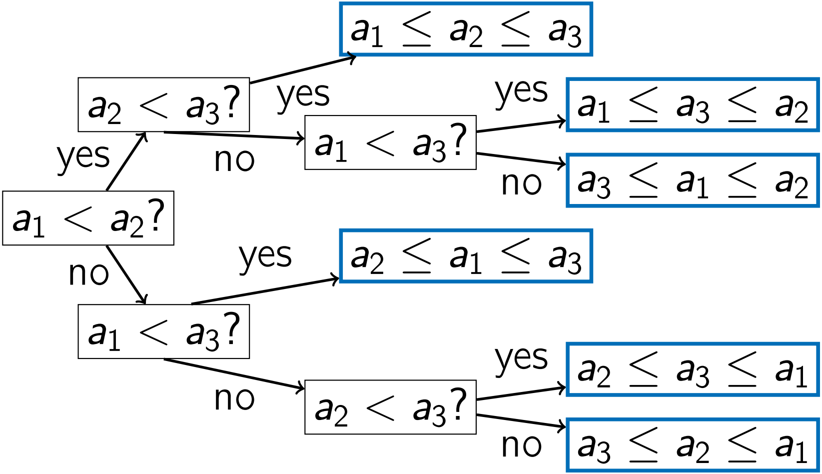
Lemma

Any comparison based sorting algorithm performs $\Omega(n \log n)$ comparisons in the worst case to sort n objects.

In other words

For any comparison based sorting algorithm, there exists an array $A[1 \ldots n]$ such that the algorithm performs at least $\Omega(n \log n)$ comparisons to sort A.

Decision Tree



Estimating Tree Depth

- the number of leaves ℓ in the tree must be at least $n!$ (the total number of permutations)

Estimating Tree Depth

- the number of leaves ℓ in the tree must be at least $n!$ (the total number of permutations)
- the worst-case running time of the algorithm (the number of comparisons made) is at least the depth d

Estimating Tree Depth

- the number of leaves ℓ in the tree must be at least $n!$ (the total number of permutations)
- the worst-case running time of the algorithm (the number of comparisons made) is at least the depth d
- $d \geq \log_2 \ell$ (or, equivalently, $2^d \geq \ell$)

Estimating Tree Depth

- the number of leaves ℓ in the tree must be at least $n!$ (the total number of permutations)
- the worst-case running time of the algorithm (the number of comparisons made) is at least the depth d
- $d \geq \log_2 \ell$ (or, equivalently, $2^d \geq \ell$)
- thus, the running time is at least

$$\log_2(n!) = \Omega(n \log n)$$

Lemma

$\log_2(n!) = \Omega(n \log n)$

Proof

$$
\begin{aligned}
\log_2(n!) &= \log_2(1 \cdot 2 \cdot \cdots \cdot n) \\
&= \log_2 1 + \log_2 2 + \cdots + \log_2 n \\
&\geq \log_2 \frac{n}{2} + \cdots + \log_2 n \\
&\geq \frac{n}{2} \log_2 \frac{n}{2} = \Omega(n \log n)
\end{aligned}
$$

$\square$

Outline

1. Problem Overview
2. Selection Sort
3. Merge Sort
4. Lower Bound for Comparison Based Sorting
5. Non-Comparison Based Sorting Algorithms

Example: sorting small integers

	1	2	3	4	5	6	7	8	9	10	11	12
A	2	3	2	1	3	2	2	3	2	2	2	1

Example: sorting small integers

	1	2	3	4	5	6	7	8	9	10	11	12
A	2	3	2	1	3	2	2	3	2	2	2	1

$$\downarrow$$

	1	2	3
$Count$	2	7	3

Example: sorting small integers

	1	2	3	4	5	6	7	8	9	10	11	12
A	2	3	2	1	3	2	2	3	2	2	2	1

$\downarrow$

	1	2	3
$Count$	2	7	3

$\downarrow$

	1	2	3	4	5	6	7	8	9	10	11	12
A'	1	1	2	2	2	2	2	2	2	3	3	3

Example: sorting small integers

	1	2	3	4	5	6	7	8	9	10	11	12
A	2	3	2	1	3	2	2	3	2	2	2	1

we have sorted these numbers
without actually comparing them!

	1	2	3	4	5	6	7	8	9	10	11	12
A'	1	1	2	2	2	2	2	2	2	3	3	3

Counting Sort: Ideas

- Assume that all elements of $A[1 \dots n]$ are integers from 1 to M.

Counting Sort: Ideas

- Assume that all elements of $A[1 \ldots n]$ are integers from 1 to M.
- By a single scan of the array A, count the number of occurrences of each $1 \leq k \leq M$ in the array A and store it in $Count[k]$.

Counting Sort: Ideas

- Assume that all elements of $A[1 \ldots n]$ are integers from 1 to M.
- By a single scan of the array A, count the number of occurrences of each $1 \leq k \leq M$ in the array A and store it in $Count[k]$.
- Using this information, fill in the sorted array A'.

`CountSort`$(A[1 \dots n])$

$Count[1 \dots M] \leftarrow [0, \dots, 0]$
`for` i `from 1 to` n:
  $Count[A[i]] \leftarrow Count[A[i]] + 1$
{k `appears` $Count[k]$ `times in` A}
$Pos[1 \dots M] \leftarrow [0, \dots, 0]$
$Pos[1] \leftarrow 1$
`for` j `from 2 to` M:
  $Pos[j] \leftarrow Pos[j-1] + Count[j-1]$
{k `will occupy range` $[Pos[k] ... Pos[k+1] - 1]$}
`for` i `from 1 to` n:
  $A'[Pos[A[i]]] \leftarrow A[i]$
  $Pos[A[i]] \leftarrow Pos[A[i]] + 1$

Lemma

Provided that all elements of $A[1 \ldots n]$ are integers from 1 to M, `CountSort`(A) sorts A in time $O(n + M)$.

Lemma

Provided that all elements of $A[1 \dots n]$ are integers from 1 to M, `CountSort`(A) sorts A in time $O(n + M)$.

Remark

If $M = O(n)$, then the running time is $O(n)$.

Summary

- Merge sort uses the divide-and-conquer strategy to sort an n-element array in time $O(n \log n)$.
- No comparison based algorithm can do this (asymptotically) faster.
- One can do faster if something is known about the input array in advance (e.g., it contains small integers).