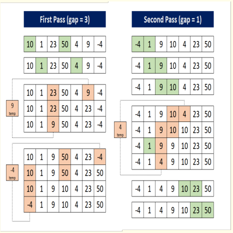
Introduction to Algorithms: Lecture 2b

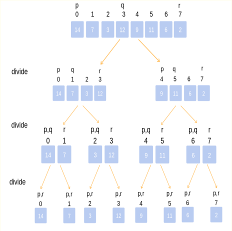
Xue Chen
xuechen1989@ustc.edu.cn
2025 spring

Outline

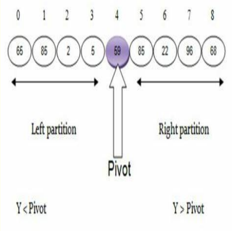
Motivation



(a) SHELLSORT



(b) MERGESORT

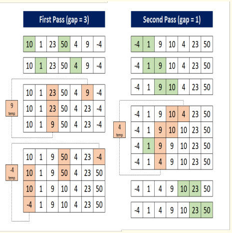


(c) QUICKSORT

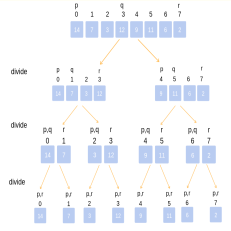
Questions

Several algorithms sort n elements in $O(n \log n)$ time
— Faster Algorithms? Is $O(n)$ possible?

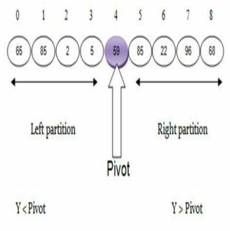
Motivation



(d) SHELLSORT



(e) MERGESORT



(f) QUICKSORT

Questions

Several algorithms sort n elements in $O(n \log n)$ time
— Faster Algorithms? Is $O(n)$ possible?

Remark: $\Omega(n)$ time to read all elements

Overview

Part 1: $\Omega(n \log n)$ lower bound for comparison sorts

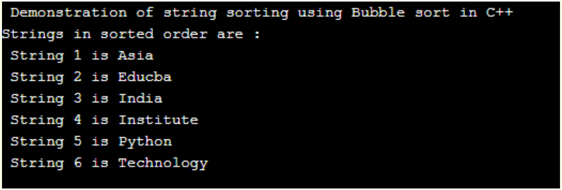
All previous sorting algorithm can sort strings, real numbers, and any objects — too flexible to get $O(n)$ time.

```
 Demonstration of string sorting using Bubble sort in C++
Strings in sorted order are :
 String 1 is Asia
 String 2 is Educba
 String 3 is India
 String 4 is Institute
 String 5 is Python
 String 6 is Technology
```

Overview

Part 1: $\Omega(n \log n)$ lower bound for comparison sorts

All previous sorting algorithm can sort strings, real numbers, and any objects — too flexible to get $O(n)$ time.


```
 Demonstration of string sorting using Bubble sort in C++
Strings in sorted order are :
 String 1 is Asia
 String 2 is Educba
 String 3 is India
 String 4 is Institute
 String 5 is Python
 String 6 is Technology
```

Part 2: $O(n)$ sorting

A linear-time algorithm for sorting bounded integers.

Outline

Comparison Sorts

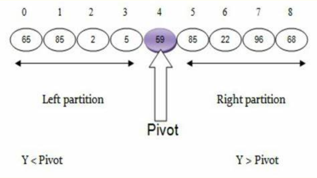
Definition: A sort algorithm is a comparison-sort if it uses only comparisons between two elements $A[i]$ and $A[j]$ to determine the output.

Comparison Sorts

Definition: A sort algorithm is a comparison-sort if it uses only comparisons between two elements $A[i]$ and $A[j]$ to determine the output.

Example: Quicksort

If pivot $x = A[i]$, $A[j]$ is in front of $A[i]$ for any $A[j] < x$.

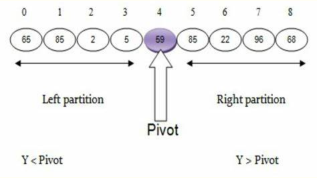


Comparison Sorts

Definition: A sort algorithm is a comparison-sort if it uses only comparisons between two elements $A[i]$ and $A[j]$ to determine the output.

Example: Quicksort

If pivot $x = A[i]$, $A[j]$ is in front of $A[i]$ for any $A[j] < x$.



OBS

For any comparisons sort, running time $\geqslant$ # comparisons.

Lower bound on comparisons

OBS

For any comparisons sort, running time $\geqslant$ # comparisons.

Lower bound on comparisons

OBS

For any comparisons sort, running time $\geqslant$ # comparisons.

Main Question: How many comparisons do we need to determine the order of A?

Lower bound on comparisons

OBS

For any comparisons sort, running time $\geqslant$ # comparisons.

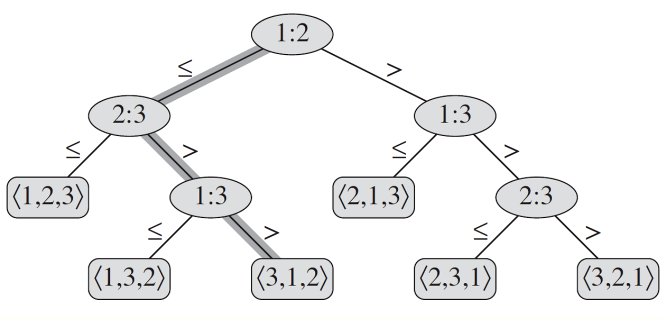
Main Question: How many comparisons do we need to determine the order of A?

For convenience, consider A as a permutation of $[n] := \{1, 2, \ldots, n\}$.

Comparisons

Any deterministic sort (excluding QUICKSORT) $\Rightarrow$ a decision tree:

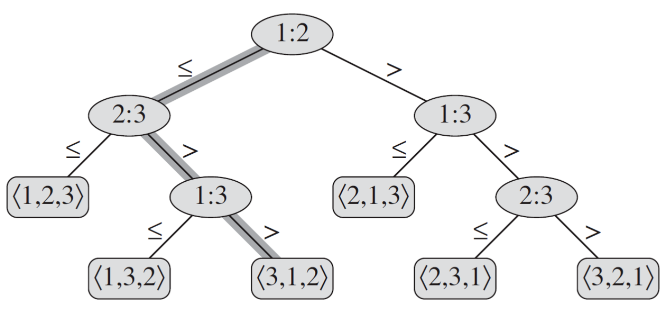
1. Starting from the root, each node (i, j) corresponds to a comparison



Comparisons

Any deterministic sort (excluding QUICKSORT) $\Rightarrow$ a decision tree:

1. Starting from the root, each node (i, j) corresponds to a comparison
2. each edge has two labels "$<$" and "$>$"
3. each leaf corresponds to a termination with a correct order



Conclusion

Observation: Worst running time $\geqslant$ length (longest path from the root).

THM 8.1 in CLRS

Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Conclusion

Observation: Worst running time $\geqslant$ length (longest path from the root).

THM 8.1 in CLRS

Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Proof.

1. Consider the decision tree corresponding to the sort algorithm.
2. The decision has $n!$ permutations on its leaves
   — so its depth is $\geqslant \log_2(n!)$.

$\square$

Conclusion

Observation: Worst running time $\geqslant$ length (longest path from the root).

THM 8.1 in CLRS

Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Proof.

1. Consider the decision tree corresponding to the sort algorithm.
2. The decision has $n!$ permutations on its leaves
— so its depth is $\geqslant \log_2(n!)$.

$\square$

Extensions: 1) The average-case running time is $\Omega(n \log n)$.
2) The running time of any randomized comparison sort is $\Omega(n \log n)$ with high prob.

Outline

Counting Sort

Suppose all elements are integers in $[0, 1, \ldots, k]$ for small $k = O(n)$.

Basic Idea

Instead of comparing them, one could count how many elements $= 0, = 1, \ldots, = k$ separately.

Counting Sort

Suppose all elements are integers in $[0, 1, \ldots, k]$ for small $k = O(n)$.

Basic Idea

Instead of comparing them, one could count how many elements $= 0, = 1, \ldots, = k$ separately.

$\Rightarrow$ Positions of elements with value ℓ are between $(\#\text{ elements} < \ell) + 1$ and $(\#\text{ elements} < \ell + 1)$.

Implementation: After counting $(\#\text{ elements} = 0), \ldots, (\#\text{ elements} = k)$, sum them up.

Description

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

on Page 195 of CLRS

Analysis

The correctness follows from the main properties of C.

Running time

Its running time is $O(n + k)$.

Analysis

The correctness follows from the main properties of C.

Running time

Its running time is $O(n + k)$.

Key property: Output is stable — numbers with the same value appear in output are in the same order as their order in input.

Analysis

The correctness follows from the main properties of C.

Running time

Its running time is $O(n+k)$.

Key property: Output is stable — numbers with the same value appear in output are in the same order as their order in input.

Next Question

What if k is huge say $k = n^{O(1)}$ or $k = 2^{64}$?

Radix Sort

If k is huge, consider the binary representation of all numbers.
Example:

$$
\begin{aligned}
A[1] &= 683 &&= (1010101011)_2 \\
A[2] &= 121 &&= (0001111001)_2 \\
&\vdots \\
A[n-1] &= 794 &&= (1100011010)_2 \\
A[n] &= 835 &&= (1101000011)_2
\end{aligned}
$$

Radix Sort

If k is huge, consider the binary representation of all numbers.
Example:

$$\begin{aligned} A[1] &= 683 && = (1010101011)_2 \\ A[2] &= 121 && = (0001111001)_2 \\ &\vdots && \\ A[n-1] &= 794 && = (1100011010)_2 \\ A[n] &= 835 && = (1101000011)_2 \end{aligned}$$

1st idea: Sort according to 1st bit, then 2nd bit, 3rd bit, ...

$$\begin{aligned} A[2] &= 121 && = ({\color{red}0}001111001)_2 \\ &\vdots && \\ A[i-1] &= 301 && = ({\color{red}0}100101101)_2 \\ A[i] &= 648 && = ({\color{red}1}010001000)_2 \\ &\vdots && \\ A[1] &= 683 && = ({\color{red}1}010101011)_2 \end{aligned}$$

Key Idea

Then sort the two groups separately.

Group 0

$$\begin{aligned} A[2] &= 121 &= (0001111001)_2 \\ &\vdots \\ A[i-1] &= 301 &= (0100101101)_2 \end{aligned}$$

Group 1

$$\begin{aligned} A[i] &= 648 &= (1010001000)_2 \\ &\vdots \\ A[1] &= 683 &= (1010101011)_2 \end{aligned}$$

Key Idea

Then sort the two groups separately.

Group 0

$$
\begin{aligned}
A[2] = 121 &= (0001111001)_2 \\
&\vdots \\
A[i-1] = 301 &= (0100101101)_2
\end{aligned}
$$

Group 1

$$
\begin{aligned}
A[i] = 648 &= (1010001000)_2 \\
&\vdots \\
A[1] = 683 &= (1010101011)_2
\end{aligned}
$$

However, this needs to store $\log_2 k$ levels (and many groups).

Question

Can we find a simpler solution?

An elegant solution: Radix-sort

Recall stable: numbers with the same value appear in the output are in the same order as their order in the input

---

procedure RADIX-SORT(d)
  for $i = 1, \ldots, d$ **do**
    use a stable sort (e.g., COUNTINGSORT) to sort array A on digit i

---

An elegant solution: Radix-sort

Recall stable: numbers with the same value appear in the output are in the same order as their order in the input

---

procedure RADIX-SORT(d)
  for $i = 1, \ldots, d$ **do**
    use a stable sort (e.g., COUNTINGSORT) to sort array A on digit i

---

Example: $1010, 0101, 1111, 0000, 0001, 0100, 1110, 0011$

An elegant solution: Radix-sort

Recall stable: numbers with the same value appear in the output are in the same order as their order in the input

---

procedure RADIX-SORT(d)
  for $i = 1, \dots, d$ **do**
    use a stable sort (e.g., COUNTINGSORT) to sort array A on digit i

---

Example: $1010, 0101, 1111, 0000, 0001, 0100, 1110, 0011$
$1010, 0000, 0100, 1110, 0101, 1111, 0001, 0011$

An elegant solution: Radix-sort

Recall stable: numbers with the same value appear in the output are in the same order as their order in the input

---

procedure RADIX-SORT(d)
  for $i = 1, \ldots, d$ **do**
    use a stable sort (e.g., COUNTINGSORT) to sort array A on digit i

---

$$\begin{aligned}\text{Example: } & 1010, 0101, 1111, 0000, 0001, 0100, 1110, 0011\\ & 1010, 0000, 0100, 1110, 0101, 1111, 0001, 0011\\ & \ldots\\ & 0000, 0001, 0011, 0100, 0101, 1010, 1110, 1111\end{aligned}$$

Analysis

Correctness and Running Time.

Analysis

Correctness and Running Time.

Correctness

If $A[i] < A[j]$, then it puts $A[i]$ in front of $A[j]$.

Analysis

Correctness and Running Time.

Correctness

If $A[i] < A[j]$, then it puts $A[i]$ in front of $A[j]$.

1. Consider the highest digit h where $A[i]$ and $A[j]$ are different. After the iteration $i = h$, $A[i]$ is in front of $A[j]$.

Analysis

Correctness and Running Time.

Correctness

If $A[i] < A[j]$, then it puts $A[i]$ in front of $A[j]$.

1. Consider the highest digit h where $A[i]$ and $A[j]$ are different. After the iteration $i = h$, $A[i]$ is in front of $A[j]$.
2. Then $A[i]$ is always in front of $A[j]$ because of the stable sort.

Running time

$O(d(n+k))$ if there are d digits where each digit takes up to k values.

Running time

$O(d(n+k))$ if there are d digits where each digit takes up to k values.

Setting parameters

Let $b = \log_2\big(\max_i A[i]\big)$, we could sort r bits (like one digit with $k = 2^r$) s.t. the time becomes $O\big(\frac{b}{r} \cdot (n + 2^r)\big)$.

Running time

$O(d(n+k))$ if there are d digits where each digit takes up to k values.

Setting parameters

Let $b = \log_2\big(\max_i A[i]\big)$, we could sort r bits (like one digit with $k = 2^r$) s.t. the time becomes $O\big(\frac{b}{r} \cdot (n + 2^r)\big)$.

1. Choose $r = \log n + O(1)$ s.t. the running time becomes $O(\frac{b}{\log n} \cdot n)$.

Running time

$O(d(n+k))$ if there are d digits where each digit takes up to k values.

Setting parameters

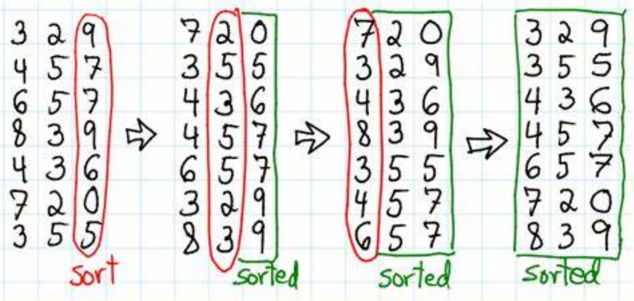
Let $b = \log_2\big(\max_i A[i]\big)$, we could sort r bits (like one digit with $k = 2^r$) s.t. the time becomes $O\big(\frac{b}{r} \cdot (n + 2^r)\big)$.

1. Choose $r = \log n + O(1)$ s.t. the running time becomes $O(\frac{b}{\log n} \cdot n)$.
2. When $b = O(\log n)$ — all numbers are in $\mathrm{poly}(n)$, linear time! ☺

Extensions

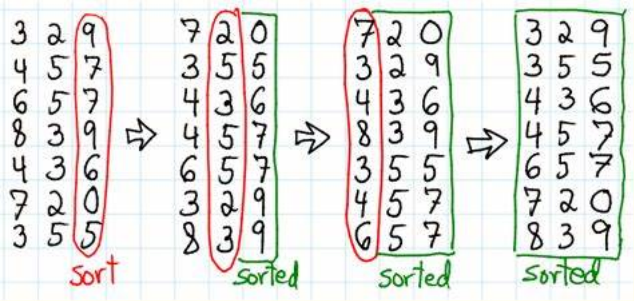
It works for strings, dates, and objects with several keywords.

Conclusion



1. COUNTSORT and RADIXSORT are fast and easy to implement.
2. Some restrictions.

Conclusion



1. CountSort and RadixSort are fast and easy to implement.
2. Some restrictions.
3. Two keys in RadixSort are (1) a delicate property called stable; (2) adjusting parameters.

Summary of sorting algorithms

Type	Time	Method	
SHELLSORT	$O(n \log^2 n)$	INSERTIONSORT	(1) $O(1)$-extra space; (2) easy to implement
MERGESORT	$O(n \log n)$	Divide & Conquer	(1) $O(n)$-extra space; (2) big constant in O
QUICKSORT	$O(n \log n)$	Divide & Conquer	(1) Most widely used; (2) Randomized
RADIXSORT	$O(n)$	COUNTINGSORT	(1) For integers $\leqslant n^{O(1)}$; (2) big constant in O

Table of sorting algorithms

More: (1) A lower bound $\Omega(n \log n)$ for comparison sort.
(2) Many algorithms could be applied to sort strings and other objects.

Questions?