UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS ILMU KOMPUTER

# Sorting in Linear Time

Lower bound for comparison sort, Non comparison sort: Counting Sort, Radix Sort

(Arlisa Yuliawati)

# Introduction

- Previously learned sorting algorithms (Insertion sort, Merge sort) sort the elements by comparing them. Such sorting algorithms are called **Comparison-based sorting algorithms.**
  - Include the heapsort, quicksort and other sorting algorithms that use comparison between elements to obtain the sorted version.
- We are going to prove that any comparison-based sorting takes $\Omega(n \lg n)$ comparisons in the worst case to sort n elements.
  - Thus, algorithm that run in $\Theta(n \lg n)$ including merge sort and heap sort are asymptotically optimal
- Some sorting algorithms do not use comparison to determine the sorted order, they run in linear time. For example: Counting sort, radix sort, bucket sort. The $\Omega(n \lg n)$ lower bound does not apply to them.

# Comparison

- From a list of $n$ elements $\langle a_1, a_2, a_3, \dots, a_n \rangle$, given two elements $a_i$ and $a_j$, we test whether they satisfy one of the following conditions to determine their relative order.
  - $a_i < a_j$
  - $a_i \leq a_j$
  - $a_i = a_j$
  - $a_i \geq a_j$
  - $a_i > a_j$

We assume that all comparisons have the form $a_i \leq a_j$

# Decision Tree

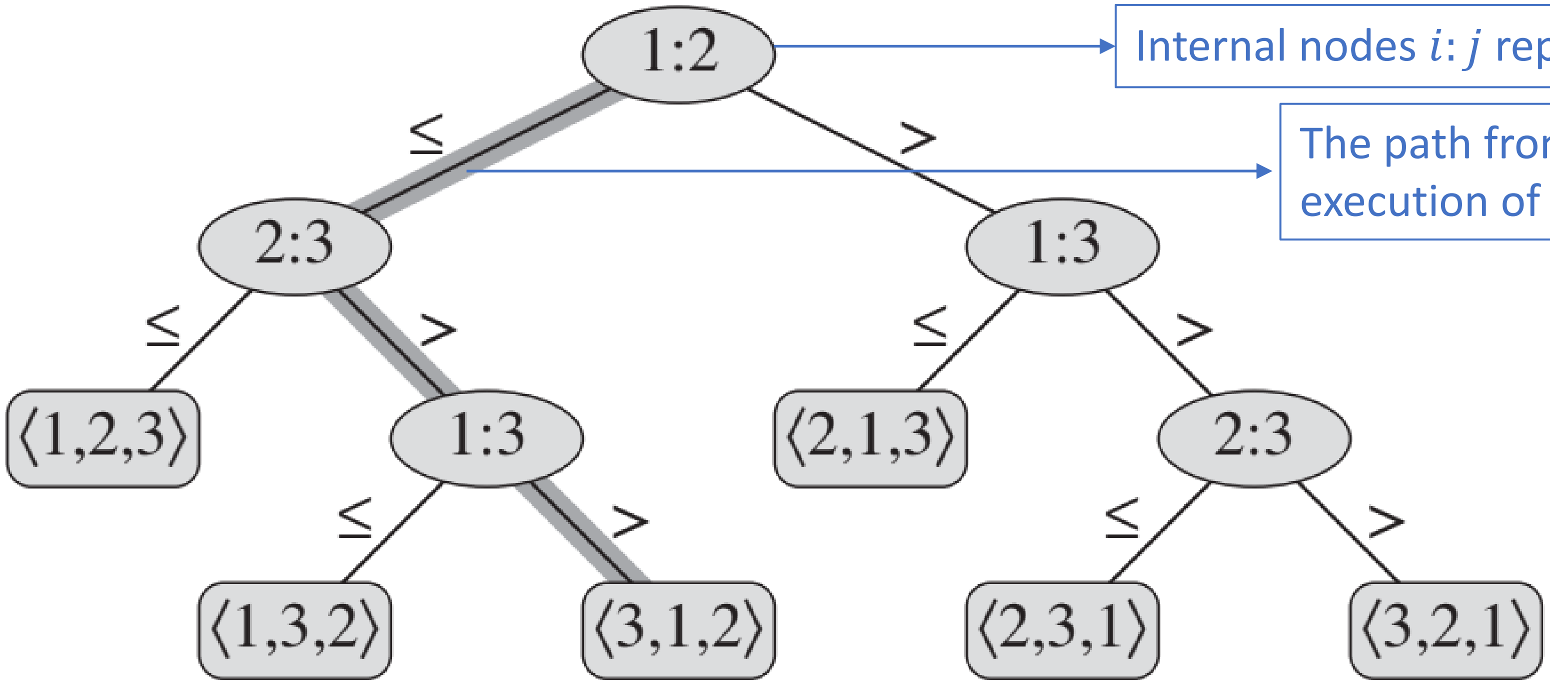


- Comparison sorts can be viewed in terms of **Decision Tree**
  - A full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

- By assuming that the input elements are distinct, we exclude the comparison "$a_i = a_j$", and the other comparisons are equivalent, so we only consider a comparison in form "$a_i \leq a_j$"

# Decision Tree

- Example:
  - A decision tree for **Insertion Sort** operating on three elements $(a_1, a_2, a_3)$. In each node, $i$: $j$ denotes the comparison between $a_i$ and $a_j$.




Internal nodes $i$: $j$ represents the comparison of $a_i$ and $a_j$

The path from root to the leaf represents the execution of the sorting algorithm

# Decision Tree

- Based on the previous decision tree:
  - Each leaf represents the permutation of $n$ elements.
  - For a comparison sort to be correct, each of the $n!$ permutation on $n$ elements must appear as one of the leaves of the decision tree, and each of these leaf must be **reachable** from the root by a path corresponding the actual execution of the comparison sort.
  - The **worst-case number of comparison** is represented by the height of the decision tree. Why?
  - **A lower bound on the heights of all decision trees** in which each permutation appears as reachable leaf <u>is a lower bound on the running time of any comparison sort algorithm</u>.

# Lower Bound for Comparison Sort

- Theorem:

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

- Proof:

Consider a decision tree of height $h$ with $k$ reachable leaves corresponds to a comparison sort of $n$ elements.

Each of $n!$ permutations appear as some leaves, and binary tree of height $h$ has no more than $2^h$ leaves. Thus, we have $n! \leq k \leq 2^h$

$$n! \leq 2^h$$
$$\lg n! \leq h$$
$$h = \Omega(n \lg n)$$

# Sorting in Linear Time

# Counting Sort



- Assumption: each of the input elements is an integer in the range 0 to $k$. When $k = O(n)$, then counting sort runs in $\Theta(n)$ time.

- The basic idea of counting sort:
  - For each input element $x$, determine the number of elements less than $x$.
  - Place the element $x$ directly into its position in the output array.
  - Example: If there are 9 elements less than $x$, then $x$ belongs in output position 9.

# Counting Sort

```
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
```

Initialize array C to all zeros

C[i] is incremented if the current value of input array equals to i

Keeping a running sum of the array C

To place each element into its sorted correct position in the input array B

# Counting Sort

- Example: Counting sort on ⟨2,5,3,0,2,3,0,3⟩

|   | 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 |

(a) After line 5

|   | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| C | 2 | 2 | 4 | 7 | 7 | 8 |

(b) After line 8

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| B |   |   |   |   |   |   | 3 |   |

|   | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| C | 2 | 2 | 4 | 6 | 7 | 8 |

(c) Line 10-12

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| B |   | 0 |   |   |   |   | 3 |   |

|   | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| C | 1 | 2 | 4 | 6 | 7 | 8 |

(d) Line 10-12

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| B |   | 0 |   |   |   | 3 | 3 |   |

|   | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| C | 1 | 2 | 4 | 5 | 7 | 8 |

(e) Line 10-12

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| B | 0 | 0 | 2 | 2 | 3 | 3 | 3 | 5 |

(f) Sorted array B

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FAKULTAS
ILMU
KOMPUTER

# Running time of Counting Sort

COUNTING-SORT$(A, B, k)$

```
1   let C[0..k] be a new array
2   for i = 0 to k
3       C[i] = 0
4   for j = 1 to A.length
5       C[A[j]] = C[A[j]] + 1
6   // C[i] now contains the number of elements equal to i.
7   for i = 1 to k
8       C[i] = C[i] + C[i - 1]
9   // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] - 1
```

Line 2-3: $\Theta(k)$

Line 4-5: $\Theta(n)$

Line 7-8: $\Theta(k)$

Line 10-12: $\Theta(n)$

Total: $\boldsymbol{\Theta(n + k)}$

Counting sort is usually used when $k = O(n)$

Counting sort beats the lower bound for comparison sort $\Omega(n \lg n)$

# Counting Sort is Stable



- Another important point of Counting sort is that it is **stable –** numbers with the same value appear in the output array in the same order as they do in the input array.
  - It break ties between two elements by the rule that whichever element appears first in the input array, appears first in the output array.
- The property of stability is important only when satellite data are carried around with the element being sorted.
- Counting sort is often used as subroutine in radix sort.

# Exercise

- Use the procedure of Counting-Sort to illustrate the sorting process on the array $A = \langle 6,0,2,0,1,3,4,6,1,3,2 \rangle$.

# Radix Sort

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FAKULTAS ILMU KOMPUTER

- For a set of $d$ digit numbers, this algorithm sort on the <u>least significant digit</u> first. For each digit, it use stable sorting algorithm.

| | | | |
|---|---|---|---|
| 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 |

# Radix Sort

- The following is the procedure of Radix Sort. It assumes that each element in the $n$-elements array $A$ has $d$ digits. The sorting process starts from the lowest-order digit (1) to the highest-order digit ($d$).

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

- Lemma

Given $n$ $d$-digit numbers in which each digit can take on up to $k$ possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d(n+k))$ time if the stable sort it uses takes $\Theta(n+k)$ time.

# Radix Sort



When $d$ is constant and $k = O(n)$, we can make radix sort run in linear time. More generally, we have some flexibility in how to break each key into digits.

- Lemma

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 if the stable sort it uses takes $\Theta(n + k)$ time for inputs in the range 0 to $k$.

# Radix Sort

***Proof*** For a value $r \leq b$, we view each key as having $d = \lceil b/r \rceil$ digits of $r$ bits each. Each digit is an integer in the range 0 to $2^r - 1$, so that we can use counting sort with $k = 2^r - 1$. (For example, we can view a 32-bit word as having four 8-bit digits, so that $b = 32$, $r = 8$, $k = 2^r - 1 = 255$, and $d = b/r = 4$.) Each pass of counting sort takes time $\Theta(n + k) = \Theta(n + 2^r)$ and there are $d$ passes, for a total running time of $\Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r))$. ■

- Given the values of $n$ and $b$, we wish to choose the value of $r$, with $r \leq b$, that minimize the expression $\left(\frac{b}{r}\right)(n + 2^r)$.
  - If $b < \lfloor \lg n \rfloor$, then choose any value of $r \leq b$ will result in $\Theta(n + 2^r)$.
  - If $b \geq \lfloor \lg n \rfloor$, then choose $r = \lfloor \lg n \rfloor$ gives the best time to within a constant factor

# Radix Sort



- Radix Sort can be used to sort records of information that are keyed by multiple fields.
  - For example, to sort the date (year, month, day), ID (with specific representation of each digit).

- Note that radix sort which uses counting sort as the intermediate stable sorting algorithm does not sort in place. Thus, when primary memory is at a premium, an in-place sorting algorithm (such as quick sort) may be preferable.

# Summary

- Comparison sorts are sorting algorithms that use the comparison of the input elements to obtain the sorted version.
- Any comparison-sorts take $\Omega(n \lg n)$ in the worst case.

- Non-comparison sorts do not compare the element, for example, counting sort get the correct position of each element by counting how many elements are less than or equal to it.
  - Counting sort is stable sorting algorithm, and it does not sort in place.

- Radix Sort use a stable sorting algorithm to sort its elements from the least significant digit to most significant one. Counting sort or quick sort can be the options.

# References


- Lecturer Slides by Bapak L. Yohanes Stefanus
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.


UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FAKULTAS ILMU KOMPUTER