

# Chapter 8. Sorting in Linear Time

Joon Soo Yoo

April 9, 2025

# Assignment

- ▶ Read §8
- ▶ Problems
  - ▶ §8.2 - 3
  - ▶ §8.3 - 3, 5
  - ▶ §8.4 - 2

## Chapter 8: Sorting in Linear Time

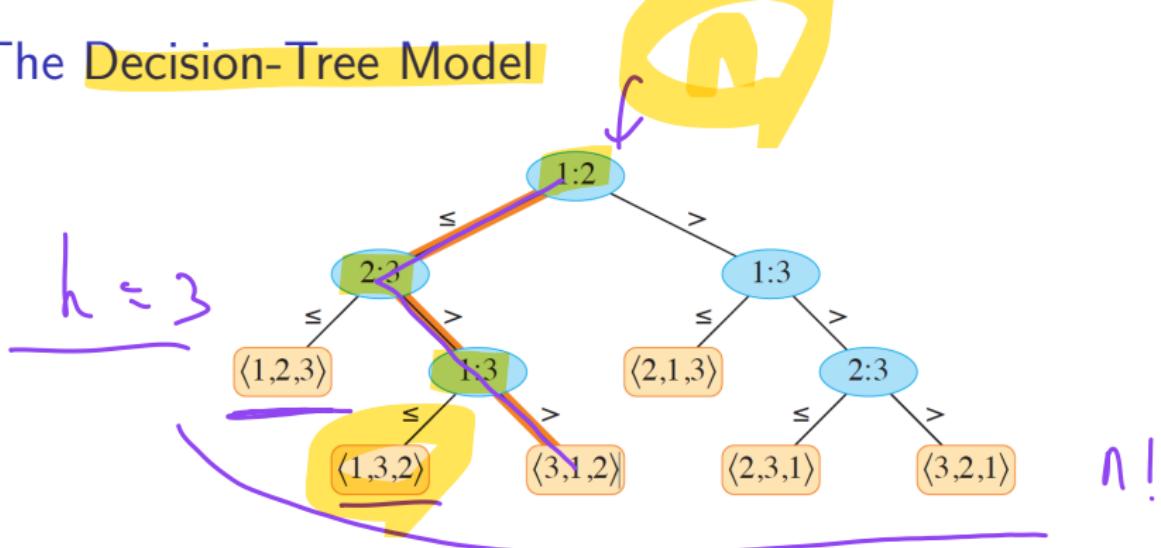
$n \log n$

- ▶ Chapter 8.1: Lower Bounds for Sorting
- ▶ Chapter 8.2: Counting Sort
- ▶ Chapter 8.3: Radix Sort
- ▶ Chapter 8.4: Bucket Sort

# Why Assume Distinct Elements in Lower Bound Proofs?

- ▶ We aim to prove a **lower bound** for **comparison-based sorting** algorithms.
- ▶ Sorting with **duplicates** can be easier:
  - ▶ Some comparisons may be skipped.
  - ▶ Fewer rearrangements may be required.
- ▶ To make the argument general, we assume all elements are **distinct**.
- ▶ **Key Insight:** Proving a lower bound for distinct elements means the bound also holds when duplicates are allowed.
- ▶ Comparisons like  $a_i = a_j$  are useless when all elements are distinct.
- ▶ So, we only consider comparisons of the form  $a_i < a_j$ .

## The Decision-Tree Model



- ▶ A **comparison sort** uses only element comparisons like  $a_i < a_j$  to determine order.
- ▶ We model these sorts using a **decision tree**:
  - ▶ A **full binary tree** where each internal node represents a comparison  $a_i : a_j$
  - ▶ Each **leaf** corresponds to a permutation of the input
- ▶ The path from **root to leaf** = sequence of comparisons made for a specific input

# Key Properties of the Decision Tree

- ▶ Internal nodes: comparisons  $a_i : a_j$
- ▶ Leaves: permutations  $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$
- ▶ The tree must have at least  $n!$  **reachable leaves** to represent all permutations
- ▶ Execution follows a single root-to-leaf path, depending on comparison outcomes
- ▶ The height of the tree = **worst-case number of comparisons**

# Theorem 8.1: Lower Bound for Comparison Sorts

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

**Proof:**

- ▶ A correct comparison sort must distinguish all  $n!$  input permutations.
- ▶ Therefore, its decision tree has at least  $n!$  **reachable leaves**.
- ▶ Any binary tree of height  $h$  has at most  $2^h$  leaves.

$$n! \leq 2^h \Rightarrow h \geq \log_2(n!) = \Theta(n \log n)$$

- ▶ Apply **Stirling's approximation**:

$$\log_2(n!) \approx \log_2\left(\sqrt{2\pi n} \left(\frac{n}{e}\right)^n\right) \Rightarrow \log_2(n!) = \Theta(n \log n)$$

- ▶ Thus, worst-case comparisons:

$$\Omega(n \log n)$$

## Corollary 8.2: Asymptotic Optimality of Heapsort and Merge Sort

Corollary: Heapsort and merge sort are asymptotically optimal comparison sorts.

$$O(n \log n)$$

### Justification:

- ▶ By Theorem 8.1, any comparison sort requires  $\Omega(n \log n)$  comparisons in the worst case.
- ▶ Both heapsort and merge sort achieve a worst-case time of  $O(n \log n)$ .
- ▶ Therefore, they match the lower bound up to constant factors:

$$\text{Worst-case time} = \Theta(n \log n)$$

**Conclusion:** No comparison sort is asymptotically faster than heapsort or merge sort.

# Chapter 8: Sorting in Linear Time

- ▶ Chapter 8.1: Lower Bounds for Sorting
- ▶ **Chapter 8.2: Counting Sort**
- ▶ Chapter 8.3: Radix Sort
- ▶ Chapter 8.4: Bucket Sort

$n \log n$



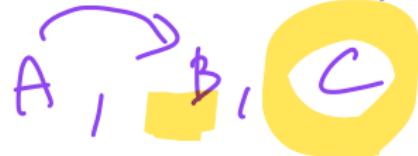
## Counting Sort — Introduction

$$\underline{0 \leq A[i] \leq k}$$
$$\underline{A[1], \dots, A[n]}$$

- ▶ Counting sort assumes that the **input array contains  $n$  integers** in the range 0 to  $k$ , for some integer  $k$ .
- ▶ It runs in  $\Theta(n + k)$  time.
- ▶ When  $k = O(n)$ , counting sort runs in  $\Theta(n)$  time.
- ▶ Unlike comparison sorts, it uses the **value of the input** to determine position.
- ▶ Key idea:
  - ▶ For each input  $x$ , count how many elements are  $\leq x$
  - ▶ Place  $x$  directly into its final position in the output array

**Example:** If **17 elements are  $\leq x$** , then  $x$  goes into position 17.

# Understanding COUNTING-SORT Step-by-Step



1. **Initialize:** Zero out array  $C[0 \dots k]$
2. **Count:** For each value  $x$  in  $A$ , increment  $C[x]$
3. **Accumulate:** Compute how many values are  $\leq i$  by cumulative sum
4. **Place:** For each  $A[j]$ , set  $B[C[A[j]]] = A[j]$ , and then decrement  $C[A[j]]$

**Key point:** Decrementing  $C[A[j]]$  handles duplicates by placing them in **stable**, ordered positions.

**Note:** We process  $A$  in reverse (from  $n$  to 1) to ensure **stability**.

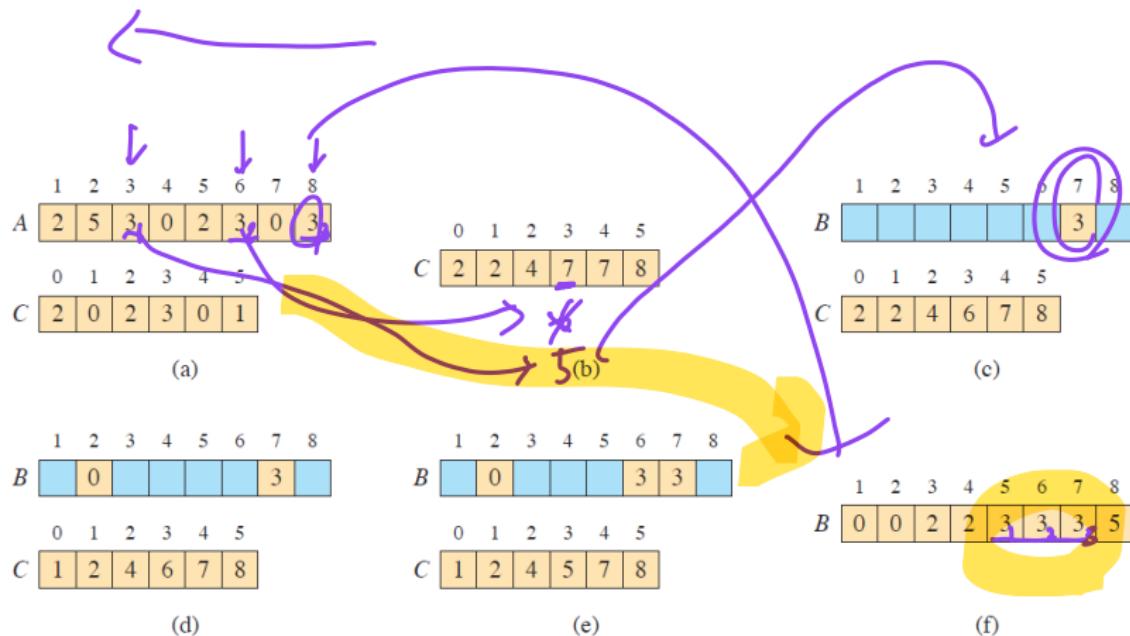
# COUNTING-SORT Procedure

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

```
1 let  $B[1 : n]$  and  $C[0 : k]$  be new arrays
2 for  $i = 0$  to  $k$ 
3    $C[i] = 0$  ←  $\Theta(k)$ 
4 for  $j = 1$  to  $n$ 
5    $C[A[j]] = C[A[j]] + 1$  ←  $\Theta(n)$ 
6 //  $C[i]$  now contains the number of elements equal to  $i$ .
7 for  $i = 1$  to  $k$ 
8    $C[i] = C[i] + C[i - 1]$  ←  $\Theta(k)$ 
9 //  $C[i]$  now contains the number of elements less than or equal to  $i$ .
10 // Copy  $A$  to  $B$ , starting from the end of  $A$ .
11 for  $j = n$  downto 1
12    $B[C[A[j]]] = A[j]$  ←  $\Theta(n)$ 
13    $C[A[j]] = C[A[j]] - 1$  // to handle duplicate values
14 return  $B$ 
```

$\Theta(n + k)$

# COUNTING-SORT Diagram

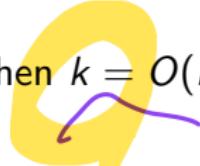


# Time Complexity of Counting Sort

- ▶ Let  $n$ : number of input elements,  $k$ : maximum value in the input
- ▶ Loop breakdown:
  - ▶ Lines 2–3: Initialize  $C[0 \dots k] \rightarrow \Theta(k)$
  - ▶ Lines 4–5: Count frequencies from  $A \rightarrow \Theta(n)$
  - ▶ Lines 7–8: Compute prefix sums  $\rightarrow \Theta(k)$
  - ▶ Lines 11–13: Place elements into  $B \rightarrow \Theta(n)$
- ▶ **Total running time:**

$$\Theta(n + \underline{k})$$

- ▶ When  $k = O(n)$ , counting sort runs in  $\Theta(n)$  time.



## Why Counting Sort Can Beat $\Omega(n \log n)$

- ▶ In Section 8.1, we proved that any comparison sort must take  $\Omega(n \log n)$  in the worst case.
- ▶ **Counting sort is not a comparison sort:**
  - ▶ It performs no comparisons between input elements.
  - ▶ Instead, it uses the **actual values** as indices into array  $C$ .
- ▶ Therefore, the lower bound of  $\Omega(n \log n)$  does not apply.

## Stability of Counting Sort

- ▶ **Definition:** A sorting algorithm is **stable** if it preserves the relative order of **equal elements**.
- ▶ In counting sort, if two elements have the same value, the one appearing first in the input appears first in the output.
- ▶ Stability matters when elements carry **satellite data**.
- ▶ **Example:**
  - ▶ Input: [(5, A), (3, B), (5, C)]
  - ▶ Stable output: [(3, B), (5, A), (5, C)]
- ▶ **Conclusion:** Counting sort's stability is **required** for **radix sort** to work correctly.

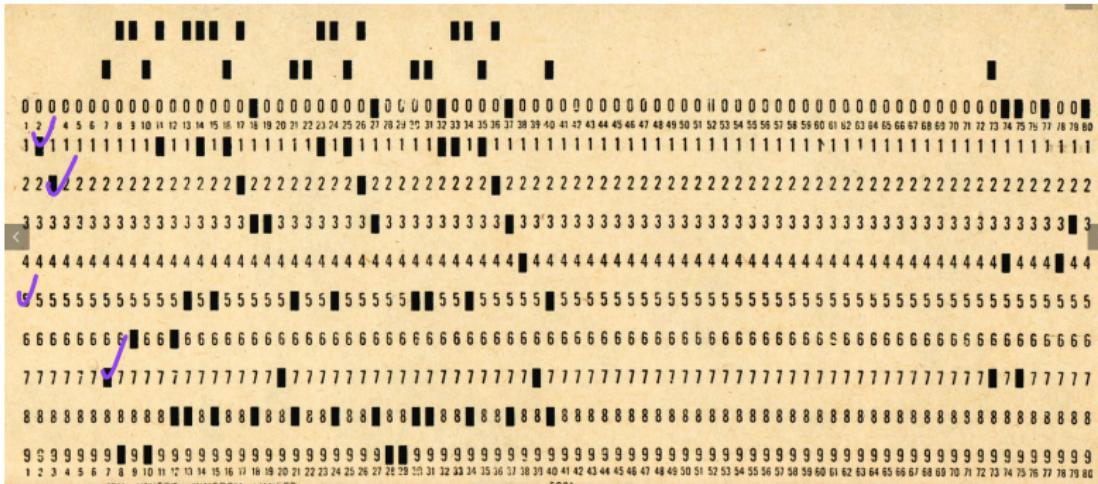
# Chapter 8: Sorting in Linear Time

- ▶ Chapter 8.1: Lower Bounds for Sorting
- ▶ Chapter 8.2: Counting Sort
- ▶ **Chapter 8.3: Radix Sort**
- ▶ Chapter 8.4: Bucket Sort

# Radix Sort and Card-Sorting Machines

- ▶ Radix sort was originally implemented using **mechanical card-sorting machines**.
- ▶ Each card had **80 columns**.
- ▶ In each column, a hole could be punched in **1 of 12 positions**.
- ▶ The sorter could be mechanically programmed to:
  - ▶ Examine a specific column
  - ▶ Distribute each card into one of 12 bins based on the punched position
- ▶ An operator would then **gather cards bin by bin**, so that cards with the first punch appear on top.

# Understanding Punch Card Encoding (IBM Standard)



- ▶ The top two unlabeled rows correspond to **zone punches**:
  - ▶ Top row = **row 12**
  - ▶ Second row = **row 11**
- ▶ Digits 0–9 are encoded by a single punch in rows 0–9.
- ▶ Letters are encoded by combining for instance, **Row 12 + 1–9 = A–I**

# Encoding Digits in Punch Cards

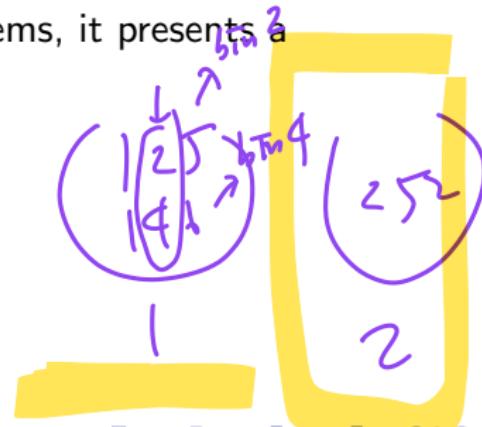
- ▶ For decimal digits, each column used only **10 of the 12 punch positions**:
  - ▶ Rows 0 through 9 → digits 0 through 9
  - ▶ Rows 11 and 12 reserved for encoding letters or special characters
- ▶ A  $d$ -digit number occupies a field of  **$d$  columns**.
- ▶ Since the card sorter can look at only **one column at a time**, sorting  $n$  cards by a  $d$ -digit number requires a sorting algorithm that processes each digit.

# The Need for a Sorting Algorithm

124 → 124  
572 → 572

- ▶ You might intuitively think to sort numbers by the **most significant digit (MSD)**:
  - ▶ Sort by the leftmost digit
  - ▶ Recursively sort each bin by the next digit
  - ▶ Combine the bins in order to complete the sort
- ▶ This seems natural, but in physical systems, it presents a practical challenge...

125 → bin 1  
146 → bin 1  
252 → bin 2



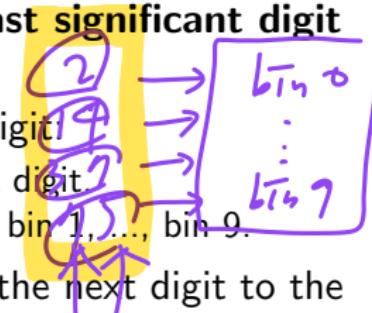
## Drawback of MSD Sorting

- ▶ MSD-based sorting creates **many intermediate piles** of cards.
- ▶ After sorting by the **first digit**, you have **10 bins**.
- ▶ To recursively sort one bin, the other **9 bins** must be **set aside and tracked**.
- ▶ This results in increased **space** and logistical **complexity**, especially in manual sorting.

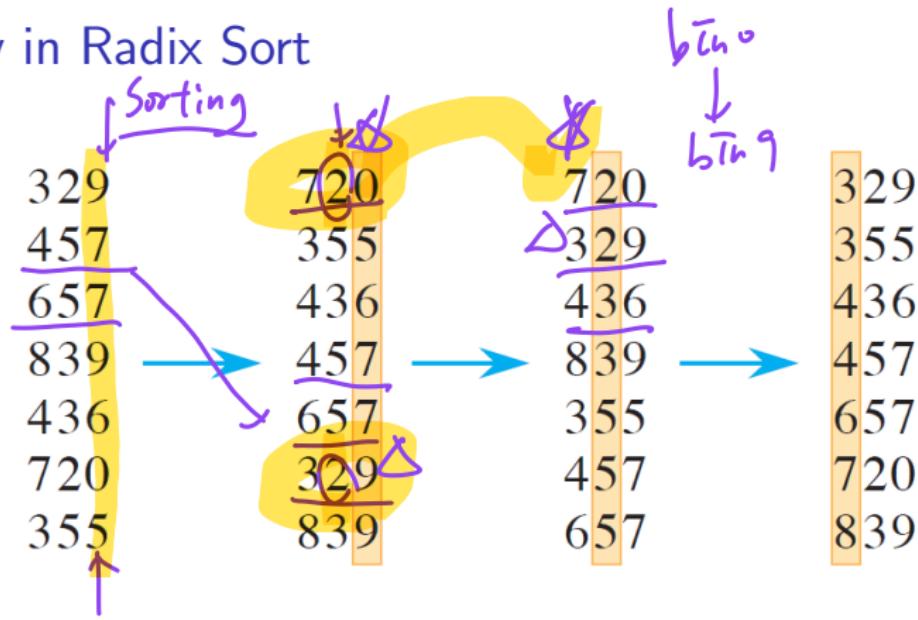
**Conclusion:** MSD sorting leads to extra work managing intermediate piles—hence a better algorithmic strategy is needed.

## Radix Sort: LSD-First Strategy and Stability

- ▶ Radix sort solves the problem of card sorting **counterintuitively** by sorting on the **least significant digit first**.
- ▶ Cards are first sorted by the rightmost digit
  - ▶ Cards go into **bins (0–9)** based on that digit
  - ▶ Bins are gathered in order: bin 0, then bin 1, ..., bin 9.
- ▶ Then the entire deck is sorted again on the next digit to the left, and recombined in the same way.
- ▶ This process continues until **all  $d$  digits** have been sorted.
- ▶ **Remarkably**, at that point the cards are **fully sorted by their entire  $d$ -digit number**.
- ▶ Thus, radix sort requires only  $d$  passes through the deck.



## Stability in Radix Sort



- ▶ For radix sort to work correctly, the digit-level sorting must be **stable**.
- ▶ Stability ensures that the relative order of elements with equal digits is preserved.
- ▶ Mechanical card sorters are stable by design.

## Radix Sort for Multi-Field Sorting (e.g., Dates)

- ▶ Radix sort can be used to sort **records with multiple fields**.
- ▶ Example: sorting dates by **year**, **month**, and **day**.
- ▶ One approach:
  - ▶ Use a comparison sort with a comparator that checks year first, then month, then day.
- ▶ Alternatively, use radix sort:
  - ▶ **First pass:** sort by day (least significant)
  - ▶ **Second pass:** sort by month
  - ▶ **Final pass:** sort by year (most significant)
- ▶ Requires each sort to be **stable**.

# RADIX-SORT Pseudocode

**RADIX-SORT assumes:**

- ▶ Each element in array  $A[1 \dots n]$  has  $d$  digits
- ▶ Digit 1 is the **least significant digit (LSD)**, and digit  $d$  is the most significant

The diagram shows the pseudocode for RADIX-SORT enclosed in a brace. Above the brace, there is handwritten text:  $d \leftarrow 1$  with a vertical line and dots indicating it continues; below the brace, there is handwritten text:  $A[1], \vdots, A[n]$ . The brace groups the entire pseudocode block.

```
RADIX-SORT( $A, n, d$ )
1   for  $i = 1$  to  $d$ 
2       use a stable sort to sort array  $A[1 : n]$  on digit  $i$ 
```

- ▶ The pseudocode does not specify which stable sort to use
- ▶ **COUNTING-SORT** is commonly used for digit-level sorting
- ▶ Why counting sort?
  - ▶ Runs in  $\Theta(n + k)$ , where  $k$  is the digit range (e.g., 0–9)
  - ▶ Naturally stable and efficient for small-digit alphabets

# Time Complexity of RADIX-SORT

## Lemma 8.3.

Given  $n$   $d$ -digit numbers in which each digit can take on up to  $k$  possible values, **RADIX-SORT correctly sorts** these numbers in

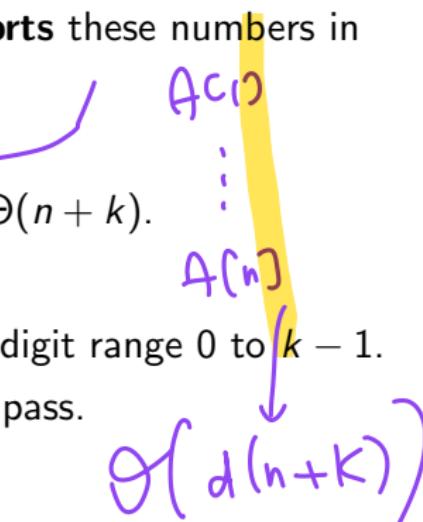
$$\Theta(d(n + k))$$

time, assuming the stable sort used runs in  $\Theta(n + k)$ .

## Proof:

- ▶ Each pass processes  $n$  items, each with digit range 0 to  $k - 1$ .
- ▶ COUNTING-SORT takes  $\Theta(n + k)$  per pass.
- ▶ With  $d$  passes, total running time is:

$$\Theta(d(n + k))$$



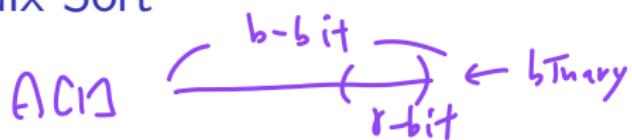
## Corollary: Linear-Time Sorting When $d$ and $k = O(n)$

- ▶ If  $d$  is constant and  $\underline{k = O(n)}$ , then:

$$\Theta(d(n + k)) = \Theta(n)$$

- ▶ So **radix sort runs in linear time** in this case.
- ▶ We also have flexibility in how we define “**digits**,” especially for binary keys.

## Lemma 8.4: Binary Keys in Radix Sort



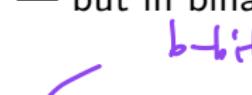
### Lemma 8.4.

Given  $n$  binary keys of  $b$  bits each, and any positive integer  $r \leq b$ , radix sort can sort the keys in:

$$\Theta\left(\frac{b}{r}(n + 2^r)\right)$$

time, assuming the stable sort used takes  $\Theta(n + k)$  time for inputs in the range  $[0, k]$ .

## Proof of Lemma 8.4

- ▶ A binary key is just a number made of  $b$  bits (e.g., 32-bit integers).
  - ▶ Radix sort requires us to define “digits” — but in binary, we get to choose what a digit means.  

  - ▶ Let  $r$  be the number of bits per digit.
  - ▶ Then:
    - ▶ Number of digits  $d = \lceil b/r \rceil$
    - ▶ Each digit ranges from 0 to  $2^r - 1$
    - ▶ Counting sort takes  $\Theta(n + 2^r)$  per pass
    - ▶ Total time:  $\Theta(d(n + 2^r)) = \Theta(\frac{b}{r}(n + 2^r))$

$$\Theta(n + k)$$

$$\underline{\Theta(d)(n+2^r)}$$

## Example: Sorting 32-bit Keys with Byte-wise Digits

- ▶ Let  $b = 32$ ,  $r = 8$  (one digit = 1 byte).

▶ Then:

- ▶ Number of digits:  $d = b/r = 4$
- ▶ Digit range: 0 to  $2^8 - 1 = 255$
- ▶ Each counting sort pass:  $\Theta(n + 256)$

▶ Total cost:

$$\Theta(4(n + 256)) = \Theta(n)$$

- ▶ **Conclusion:** Efficient for fixed-width binary keys like 32-bit integers.

## Why Is This Useful?

- ▶ **Tuning  $r$**  gives you flexibility:
  - ▶ Larger  $r$ : fewer passes, but larger bins
  - ▶ Smaller  $r$ : more passes, but smaller bins
- ▶ **Summary:** Lemma 8.4 gives a general formula to analyze radix sort's behavior on binary data.

# Chapter 8: Sorting in Linear Time

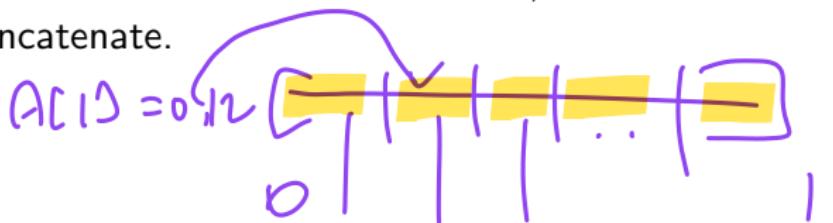
- ▶ Chapter 8.1: Lower Bounds for Sorting
- ▶ Chapter 8.2: Counting Sort
- ▶ Chapter 8.3: Radix Sort
- ▶ **Chapter 8.4: Bucket Sort**

## Bucket Sort Overview

(**Bucket sort**) assumes the input consists of  $n$  real numbers drawn uniformly and independently from the interval  $(0, 1)$ .

- ▶ Like **counting sort**, it achieves **average-case** running time of  $\Theta(n)$  by making assumptions about the input.
- ▶ **Counting sort** assumes input integers lie in a **small range**.
- ▶ **Bucket sort** assumes inputs are randomly spread over  $(0, 1)$  (uniform distribution).

**Key idea:** Partition the interval  $(0, 1)$  into  $n$  equal-sized buckets, distribute elements into these buckets, sort within each bucket, and concatenate.



## Bucket Sort: Why It Works Well

- ▶ Since inputs are **uniformly and independently** distributed:
  - ▶ Most buckets will have **very few elements**
  - ▶ This makes **sorting** inside each bucket fast
- ▶ We sort each bucket using a simple algorithm like **insertion sort**
- ▶ Finally, we **concatenate** the sorted buckets in order

**Conclusion:** With high probability, the total **running time** is  $\Theta(n)$

# Bucket Sort Assumptions and Structure

## Assumptions:

- ▶ Input array  $A[1 \dots n]$  such that each  $A[i] \in [0, 1)$

## Algorithm structure:

- ▶ Create an auxiliary array  $B[0 \dots n - 1]$  of empty linked lists (buckets)
- ▶ For each input  $A[i]$ , place it in bucket:  
$$B[\lfloor n \cdot A[i] \rfloor]$$

- ▶ Sort each bucket using **insertion sort**
- ▶ Scan all buckets in order and concatenate results

## Example Setup: Input Array $A[1 \dots 10]$

**Input values (uniform in  $[0, 1)$ ):**

$$A = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]$$

**Buckets  $B[0 \dots 9]$**  will be created. Each value goes into:

Bucket  $\lfloor 10 \cdot A[i] \rfloor$

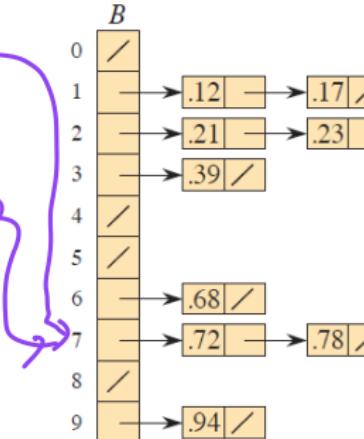
For example:

- ▶  $0.78 \rightarrow B[7]$
- ▶  $0.17 \rightarrow B[1]$
- ▶  $0.39 \rightarrow B[3]$
- ▶  $0.94 \rightarrow B[9]$

$$\lfloor 10 \cdot 0.17 \rfloor = 1$$

## Example of BUCKET-SORT ( $n = 10$ )

	<i>A</i>
1	.78
2	.17
3	.39
4	.26
5	.72
6	.94
7	.21
8	.12
9	.23
10	.68



(a)

(b)

- ▶ (a) Input array  $A[1 \dots 10]$
  - ▶ (b) Buckets  $B[0 \dots 9]$  after distributing and sorting
  - ▶ Each bucket  $B[i]$  holds values in the interval:

$$\left[ \frac{i}{10}, \frac{i+1}{10} \right)$$

- ▶ Slashes indicate the end of each bucket
  - ▶ Final output is the concatenation of all buckets in order

# BUCKET-SORT Pseudocode and Why It Works

BUCKET-SORT( $A, n$ )

- 1 let  $B[0:n - 1]$  be a new array
- 2 for  $i = 0$  to  $n - 1$
- 3 make  $B[i]$  an empty list
- 4 for  $i = 1$  to  $n$
- 5 insert  $A[i]$  into list  $B[\lfloor n \cdot A[i] \rfloor]$
- 6 for  $i = 0$  to  $n - 1$
- 7 sort list  $B[i]$  with insertion sort
- 8 concatenate the lists  $B[0], B[1], \dots, B[n - 1]$  together in order
- 9 return the concatenated lists

$\rightarrow \Theta(n)$

$\Theta(n)$

$\Theta(n^2)$



## Why it works:

- ▶ For any two elements  $A[i] \leq A[j]$ :
  - ▶ If they go into the same bucket  $\rightarrow$  insertion sort maintains order (line 7)
  - ▶ If they go into different buckets  $\rightarrow$  lower-indexed bucket comes first (line 8)
- ▶ So final output is fully sorted

# Analyzing the Running Time of Bucket Sort

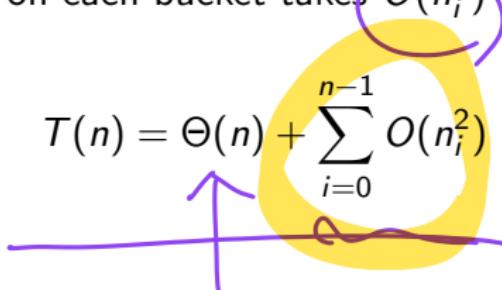
**Observation:** All lines **except line 7** (insertion sort) run in  $O(n)$  time.

**Main cost:** Line 7 — insertion sort in each of the  $n$  buckets.

- ▶ Let  $n_i$  be the number of elements in bucket  $B[i]$
- ▶ Insertion sort on each bucket takes  $O(n_i^2)$

**Total time:**

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \quad (8.1)$$



## Average-Case Running Time of BUCKET-SORT

We analyze the expected time based on the input distribution (uniform over  $[0, 1]$ ).

$$\begin{aligned}\mathbb{E}[T(n)] &= \mathbb{E} \left[ \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathbb{E} [O(n_i^2)] \quad (\text{by linearity of expectation}) \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(\mathbb{E}[n_i^2])\end{aligned}\tag{8.2}$$

We now compute  $\mathbb{E}[n_i^2]$  for each bucket  $i \in \{0, 1, \dots, n - 1\}$



Claim:  $\mathbb{E}[n_i^2] = 2 - \frac{1}{n}$  (Equation 8.3)

Each input element is equally likely to fall into any of the  $n$  buckets.

$\frac{1}{n}$

$\frac{1}{n}, 1 - \frac{1}{n}$

- ▶ Let  $n_i$  be the number of elements in bucket  $B[i]$
- ▶ View  $n_i$  as the number of **successes** in  $n$  Bernoulli trials:
  - ▶ Success: input falls into bucket  $B[i]$
  - ▶ Probability of success:  $p = 1/n$ , failure  $q = 1 - 1/n$
  - ▶ So  $n_i \sim \text{Binomial}(n, 1/n)$

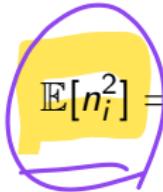
$B(n, p)$

Using binomial distribution properties:

$$\mathbb{E}[n_i] = np = 1 \quad \text{and} \quad \text{Var}(n_i) = npq = 1 - \frac{1}{n}$$

From identity:  $\text{Var}(\bar{x}) = \mathbb{E}(x^2) - (\mathbb{E}(x))^2$

$$\mathbb{E}[n_i^2] = \text{Var}(n_i) + (\mathbb{E}[n_i])^2 = \left(1 - \frac{1}{n}\right) + 1^2 = 2 - \frac{1}{n} \quad (8.3)$$



## Final Result: Linear Expected Time

Now plug Equation (8.3) into Equation (8.2):

$$\begin{aligned}\mathbb{E}[T(n)] &= \Theta(n) + \sum_{i=0}^{n-1} O\left(2 - \frac{1}{n}\right) \\ &= \Theta(n) + n \cdot O\left(2 - \frac{1}{n}\right) \\ &= \Theta(n) + O(n) = \boxed{\Theta(n)}\end{aligned}$$

**Conclusion:** Bucket sort runs in expected linear time under uniform distribution.

## Appendix - Binomial Distribution

The **binomial distribution** models the number of successes in  $n$  independent yes/no trials, each with the same probability of success.

**Definition:** Let  $X \sim \text{Binomial}(n, p)$ , where:

- ▶  $n$ : number of trials
- ▶  $p$ : probability of success on each trial
- ▶  $q = 1 - p$ : probability of failure
- ▶  $X$ : number of successes in  $n$  trials

**Properties:**

$$\mathbb{E}[X] = np, \quad \text{Var}(X) = np(1 - p)$$

Binomial distribution is used to model bucket sizes in the analysis of bucket sort.

## Example: Tossing Coins

Imagine tossing a fair coin  $n = 10$  times:

- ▶ Each toss has:
  - ▶ Success = heads, with probability  $p = 0.5$
  - ▶ Failure = tails, with probability  $q = 0.5$
- ▶ Let  $X$  be the number of heads obtained
- ▶ Then  $X \sim \text{Binomial}(10, 0.5)$

**Expected number of heads:**

$$\mathbb{E}[X] = 10 \cdot 0.5 = 5$$

# Question?