

## Lecture 7: Linear Sorts

### Course Learning Outcome :

Show how time and space complexities can be traded-off (e.g. **distribution counting sort**, hashing)

**Dr. Khalil Yousef**

Adopted from the Slides of the ECE 608 Computational Models and Methods Course at Purdue University

Read Chapter 8 of *Introduction to Algorithms*

## Comparison Sort Algorithms

---

How fast can we sort? Much depends on the assumptions one makes about how the sorting is accomplished.

So far we have considered only algorithms that sort by comparing pairs of elements (i.e., INSERTION-SORT, MERGE-SORT, HEAPSORT, and QUICKSORT). They are called **comparison sorts**.

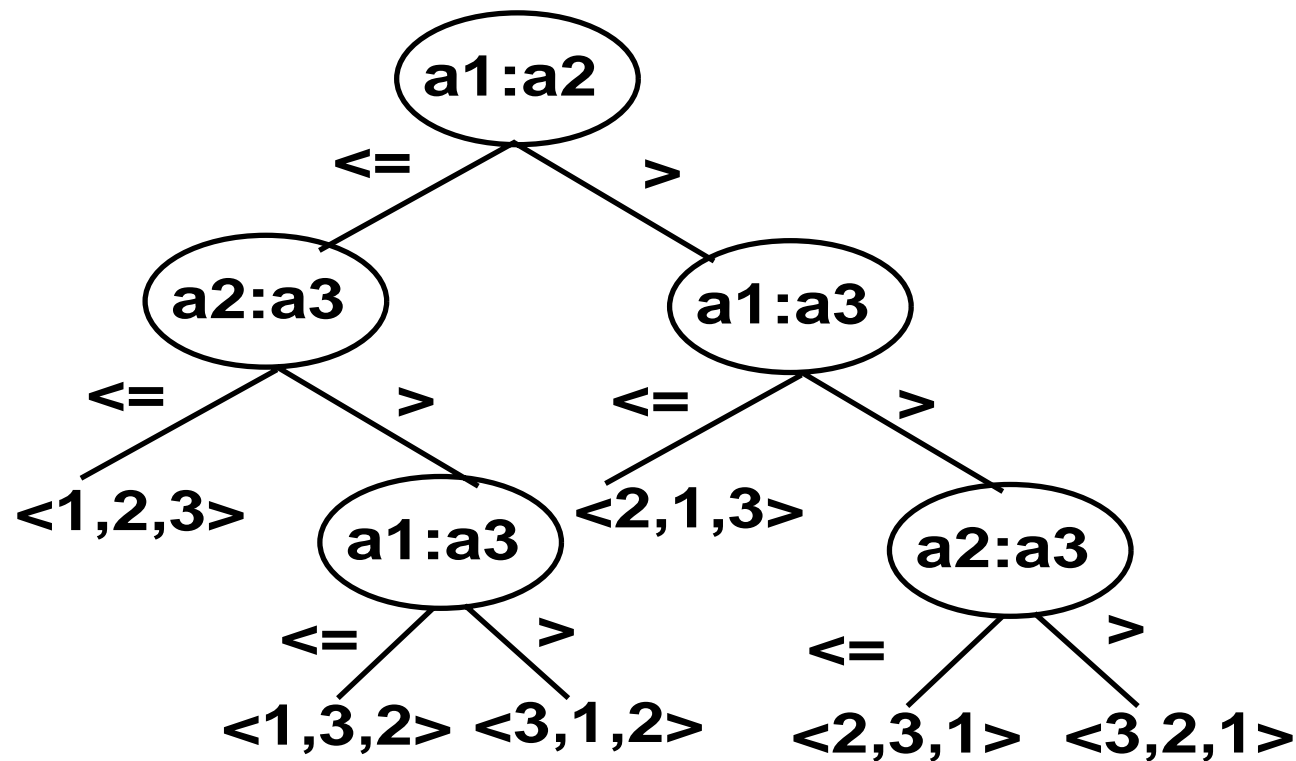
Any sort requires at least  $n - 1$  comparisons, or it won't have examined the input. However, all comparison sorts require  $\Omega(n \lg n)$  comparisons to sort in the worst case. Can we achieve a sort that is faster than  $\Omega(n \lg n)$ ?

We assume that all elements are distinct while we are discussing the lower bound on comparison sorts. In this case,  $a_i < a_j$ ,  $a_i > a_j$ ,  $a_i \leq a_j$ ,  $a_i \geq a_j$  provide equivalent information relative to ordering; hence, we use  $a_i \leq a_j$ .

## Decision Trees

---

A comparison sort can be viewed in terms of a **decision tree**, which represents the comparisons performed by the algorithm operating on an input of a certain size.



## Decision Trees *continued*

---

- There is a comparison tree for each input of size  $n$ .
- The leaves represent all possible permutations of the input array; hence, there are  $n!$  leaves in the tree.
- Internal nodes represent comparisons between pairs of elements.
- The path from the root to a leaf shows the comparisons for arriving at the leaf's permutation (an execution trace).
- The length of the longest path in the tree indicates its worst-case running time.

**Theorem 9.1:** Any decision tree that sorts  $n$  elements has height  $\Omega(n \lg n)$ , (i.e., at least one path is that length).

## Complexity of Comparison Sorts

---

**Proof:** (We ignore data movement, bookkeeping operations, etc.)

The number of leaves in the decision tree is at least  $n!$ , or two permutations go to the same leaf.

The number of leaves in a binary tree is  $\leq 2^h$ , and:

$$\begin{aligned} 2^h &\geq n! \\ h &\geq \lg(n!) \end{aligned}$$

Recall **Stirling's approximation**:  $n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n (1 + \Theta(1/n)) > \left(\frac{n}{e}\right)^n$ , hence:

$$\begin{aligned} h &\geq \lg\left(\frac{n}{e}\right)^n = n \lg n - n \lg e \\ h &= \Omega(n \lg n) \end{aligned}$$

## Decision Tree Questions

---

1. In a comparison sort decision tree, what do the leaves of the tree represent? How many leaves must there be in a decision tree sorting  $n$  elements?
2. In a comparison sort decision tree, what do the non-leaf nodes of the tree represent?
3. In a comparison sort decision tree, what do the edges of the tree represent?
4. What is the length of the longest possible path in any comparison sort decision tree for an input of size  $n$  such that comparisons are not duplicated? Why?
5. What is the length of the shortest possible path in any comparison sort decision tree? Why?

## Distribution Counting Sort or simply Counting Sort

---

**Assumption:** input elements are integers in the range 1 to  $k$ , where  $k$  can be different than  $n$ . Duplicates are allowed.

**Abstract:** The basic idea is to count, for each element  $x$ , the number of elements less than that element so that we can determine the index of placement for  $x$ . If 10 elements are less than  $x$ , then  $x$  goes into the 11th position. (The algorithm is slightly more complex than this because it handles duplicates.)

We will show that COUNTING-SORT's complexity is  $O(n + k)$ . If  $k = O(n)$ , then the complexity is  $O(n)$ .

### Data structures used:

1. **Input:**  $A[1..n]$  where  $A[j] \in \{1, 2, \dots, k\}$ .
2. **Output:**  $B[1..n]$ , which is sorted.
3. **Auxiliary:**  $C[1..k]$  is temporary storage such that  $C[i]$  will contain the number of elements less than or equal to key  $i$ .

## The COUNTING-SORT Algorithm

---

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

1. **for**  $i \leftarrow 1$  **to**  $k$

2.     **do**  $C[i] \leftarrow 0$

3. **for**  $j \leftarrow 1$  **to**  $\text{length}[A]$

4.     **do**  $C[A[j]] \leftarrow C[A[j]] + 1$

5.  $\triangleright C[i]$  now contains the number of elements equal to  $i$

6. **for**  $i \leftarrow 2$  **to**  $k$

7.     **do**  $C[i] \leftarrow C[i] + C[i - 1]$

8.  $\triangleright C[i]$  now contains the number of elements less than or equal to  $i$

9. **for**  $j \leftarrow \text{length}[A]$  **downto** 1

10.    **do**  $B[C[A[j]]] \leftarrow A[j]$

11.        $C[A[j]] \leftarrow C[A[j]] - 1$



COUNTING-SORT **Example:**  $A = \langle 4, 2, 1, 3, 2, 3 \rangle$

---

Initially  $A$  is: 

4	2	1	3	2	3
---	---	---	---	---	---

After executing lines 1 through 4,  $C$  is: 

1	2	2	1
---	---	---	---

After executing lines 6 through 7,  $C$  is: 

1	3	5	6
---	---	---	---

Now we begin iterating over the loop in lines 9 through 11.

**Iteration 1:**  $j = 6$ , so place  $A[6]$  into the correct position in  $B$  given  $C$ :  $B[C[A[6]]] = B[5] = A[6] = 3$  and  $C[A[6]] = C[A[6]] - 1 = 5 - 1 = 4$ , so:

$B$  is: 

				3	
--	--	--	--	---	--

 $C$  is: 

1	3	4	6
---	---	---	---

COUNTING-SORT **Example:**  $A = \langle 4, 2, 1, 3, 2, 3 \rangle$  *continued*

---

**Iteration 2:**  $j = 5$ , so place  $A[5]$  into the correct position in  $B$  given  $C$ :  $B[C[A[5]]] = B[3] = A[5] = 2$  and  $C[A[5]] = C[A[5]] - 1 = 3 - 1 = 2$ , so:

$B$  is: 

				3	
--	--	--	--	---	--

 $C$  is: 

1	2	4	6
---	---	---	---

**Iteration 3:**  $j = 4$ , so place  $A[4]$  into the correct position in  $B$  given  $C$ :  $B[C[A[4]]] = B[4] = A[4] = 3$  and  $C[A[4]] = C[A[4]] - 1 = 4 - 1 = 3$ , so:

$B$  is: 

		2	3	3	
--	--	---	---	---	--

 $C$  is: 

1	2	3	6
---	---	---	---

**Iteration 4:**  $j = 3$ , so place  $A[3]$  into the correct position in  $B$  given  $C$ :  $B[C[A[3]]] = B[1] = A[3] = 1$  and  $C[A[3]] = C[A[3]] - 1 = 1 - 1 = 0$ , so:

$B$  is: 

1		2	3	3	
---	--	---	---	---	--

 $C$  is: 

0	2	3	6
---	---	---	---

COUNTING-SORT **Example:**  $A = \langle 4, 2, 1, 3, 2, 3 \rangle$  *continued*

---

**Iteration 5:**  $j = 2$ , so place  $A[2]$  into the correct position in  $B$  given  $C$ :  $B[C[A[2]]] = B[2] = A[2] = 2$  and  $C[A[2]] = C[A[2]] - 1 = 2 - 1 = 1$ , so:

$B$  is: 

1	2	2	3	3	
---	---	---	---	---	--

 $C$  is: 

0	1	3	6
---	---	---	---

**Iteration 6:**  $j = 1$ , so place  $A[1]$  into the correct position in  $B$  given  $C$ :  $B[C[A[1]]] = B[6] = A[1] = 4$  and  $C[A[1]] = C[A[1]] - 1 = 6 - 1 = 5$ , so:

$B$  is: 

1	2	2	3	3	4
---	---	---	---	---	---

 $C$  is: 

0	1	3	5
---	---	---	---

## The COUNTING-SORT Algorithm *continued*

---

If the elements of  $A$  are distinct, then  $C[A[j]]$  contains the correct final position for  $A[j]$  in  $B$ ; however, if there are duplicates, then  $C[A[j]]$  is decremented each time we move  $A[j]$  into  $B$ , so that subsequent duplicates of  $A[j]$  can be correctly placed in  $B$ .

COUNTING-SORT is **stable**. A sort is stable if equal keys remain in the same relative order in the sorted sequence as in the original input sequence. For example,

$$\boxed{4} \boxed{2_1} \boxed{3} \boxed{2_2} \rightarrow \boxed{2_1} \boxed{2_2} \boxed{3} \boxed{4}$$

**Complexity:** COUNTING-SORT runs in  $\Theta(n + k)$  time. Hence, it is not bounded by  $\Omega(n \lg n)$ . Note that it is not a comparison sort, so there is no contradiction.

## Radix Sort

---

If  $k$  is very large, it may dominate the function  $O(n + k)$ .  $k$  can be reduced by combining COUNTING-SORT with RADIX-SORT.

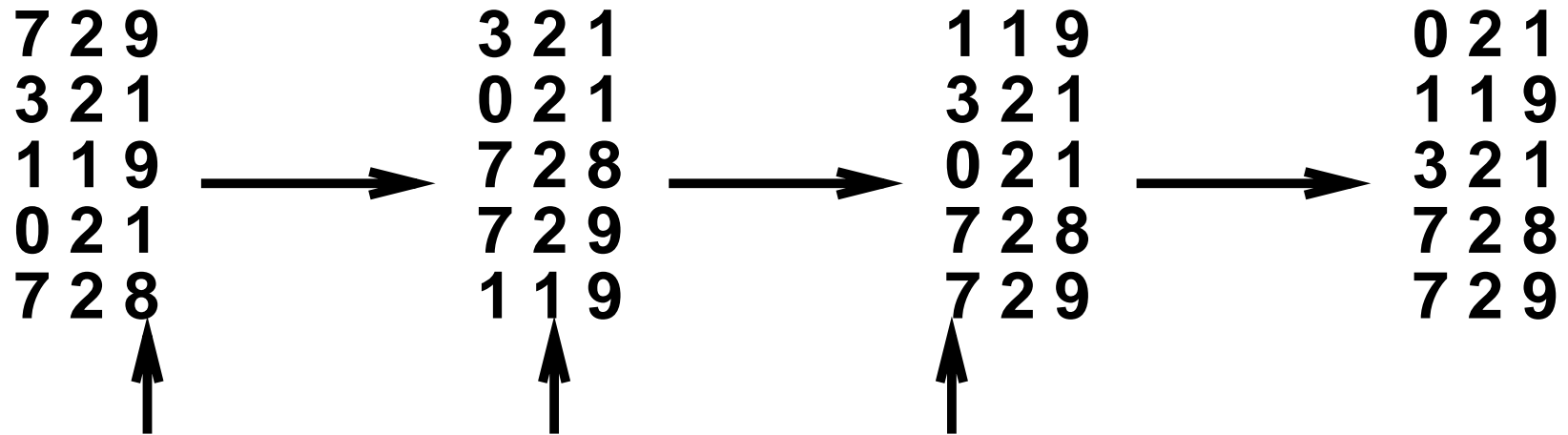
The basic idea is to sort  $n$   $d$ -digit numbers from the least significant digit to the most significant digit (where digit 1 is the lowest-order digit and  $d$  is the highest-order digit). This requires  $d$  passes with a stable sort; hence, COUNTING-SORT is an excellent choice.

RADIX-SORT( $A, d$ )

1. **for**  $i \leftarrow 1$  **to**  $d$
2.     **do** use a stable sort to sort array  $A$  on digit  $i$

## RADIX-SORT Example

---



The correctness of RADIX-SORT follows by induction on the digit being sorted.

Why do we work from the lowest order digit to the highest? What would be required to work from high to low?

## Correctness of RADIX-SORT

---

**Base Case:** If  $d = 1$ , there's only one digit, so sorting on that digit sorts the array.

**Assume:** The sort on the low-order  $d - 1$  digits correctly sorts the array if we ignore the higher order digits.

**Inductive step:** The sort on the  $d$ th digit will order the elements by their  $d$ th digit. Consider two elements,  $a$  and  $b$ , with  $d$ th digits  $a_d$  and  $b_d$ , respectively. There are three cases to consider:

1. If  $a_d < b_d$ , the sort will put  $a$  before  $b$  which is correct regardless of the lower order digits.
2. If  $a_d > b_d$ , the sort will put  $b$  before  $a$  which is correct regardless of the lower order digits.
3. If  $a_d = b_d$ , the sort will leave  $a$  and  $b$  in the same relative order because the sort is stable. Because the order was correct based on the  $d - 1$  lower-order digits,  $a$  and  $b$  are in the correct order.

## Complexity of RADIX-SORT

---

COUNTING-SORT is an excellent choice for the stable sorting algorithm used by RADIX-SORT. In that case,  $T(n) = \Theta(d(n + k))$ . If  $k = O(n)$ , then the running time is  $O(dn)$  for  $d$  passes of COUNTING-SORT.

**Example 1:** If we sort binary numbers in the range of 1 to  $n$ ,  $k = 2$  and  $d = \lg n$ , then  $T(n) = O(n \lg n)$ .

**Example 2:** Same scenario, except that we increase  $k$  and decrease  $d$  by grouping the bits into groups of  $r$  bits, so that  $k = 2^r$  and  $d = \lg n / r$ . Then  $T(n) = O(\frac{\lg n}{r}(n + 2^r))$ .

**Example 3:** Taking  $r = \lg n$  we would get  $T(n) = O(n)$  and  $d = 1$ ; we are just doing COUNTING-SORT.

RADIX-SORT is also useful for sorting on multiple keys, where we treat each key as a “digit”.



## Bucket Sort

---

BUCKET-SORT sorts in linear time on the average by making a different kind of assumption than COUNTING-SORT.

**Assumption:** input elements are evenly distributed over the interval  $[0, 1)$ .

**Data structures used:**

1. **Input:**  $A[1..n]$  where  $0 \leq A[j] < 1$ .
2. **Auxiliary:**  $B[0..n-1]$  is a set of *buckets* corresponding to subintervals of  $[0, 1)$  with which to classify and sort the elements of  $A$ . Each  $B[i]$  is a pointer to a sorted list of  $A[j]$ 's that fall into bucket  $i$ . We place  $A[j]$  into bucket  $B[\lfloor nA[j] \rfloor]$ , and bucket  $B[i]$  holds  $[\frac{i}{n}, \frac{i+1}{n})$ .

Given that the elements in  $A$  are evenly distributed over interval  $[0, 1)$ , the  $B[i]$  lists should be short.

## BUCKET-SORT Algorithm

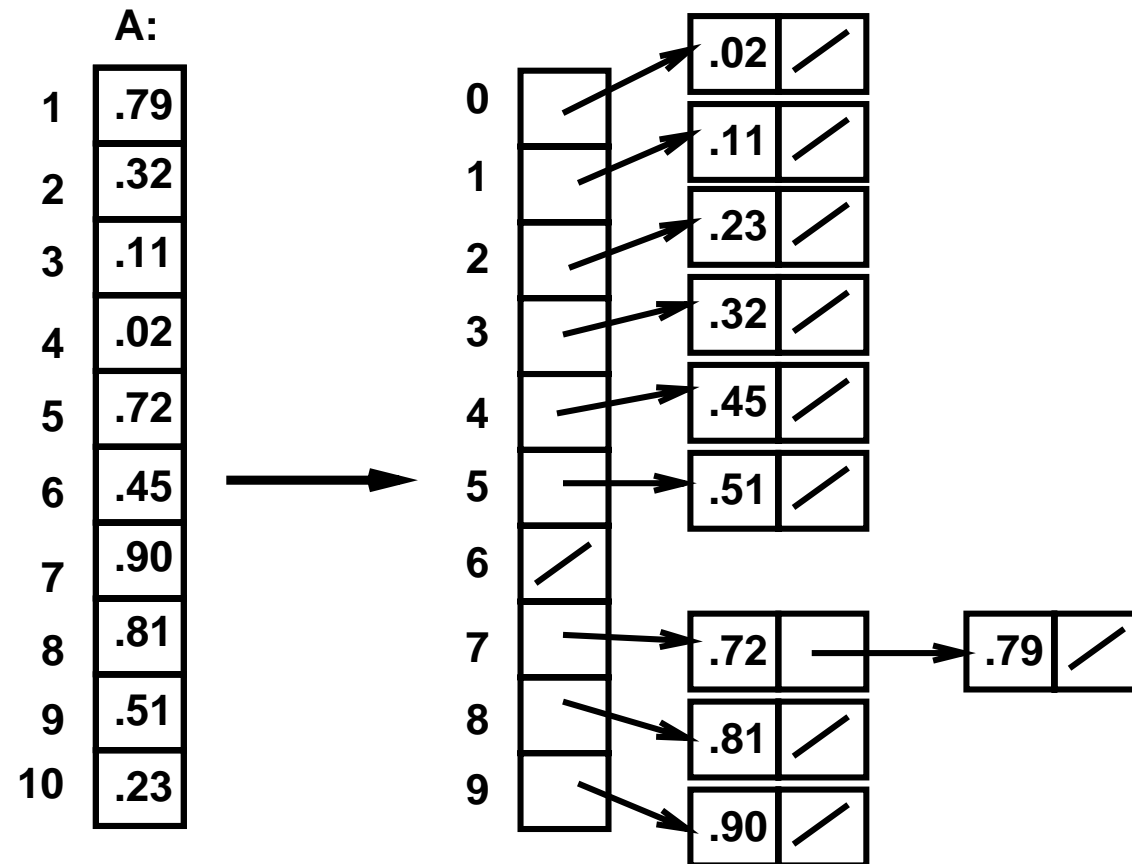
---

BUCKET-SORT( $A$ )

1.  $n \leftarrow \text{length}[A]$
2. **for**  $i \leftarrow 1$  **to**  $n$
3.     **do** insert  $A[i]$  into list  $B[\lfloor nA[i] \rfloor]$
4. **for**  $i \leftarrow 0$  **to**  $n - 1$
5.     **do** sort list  $B[i]$  with INSERTION-SORT
6. concatenate the lists  $B[0], B[1], \dots, B[n - 1]$  together in order

## BUCKET-SORT Example

---



## BUCKET-SORT Complexity

---

To analyze the running time, we observe that all steps except the INSERTION-SORT require  $O(n)$  time. The question is how much time do the INSERTION-SORT calls require?

Bucket sort runs in expected time  $O(n)$ . The worst case is  $O(n^2)$ , however, as all numbers could conceivably land in the same bucket.