

# Chapter 8: Sorting in Linear Time

## Lower Bound for Sorting

This section is **OPTIONAL**. The main take-away is that all comparison based sorting algorithms have a lower bound running time  $\Omega(n \log n)$  for the worst-case.

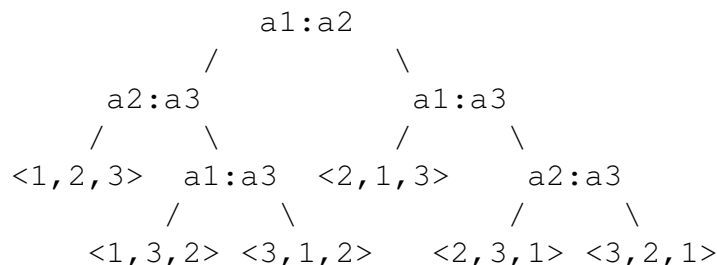
**Comparison sort:** A sorting algorithm that is based only on comparisons between the input elements.

Comparison sorts can be viewed abstractly in terms of decision trees.

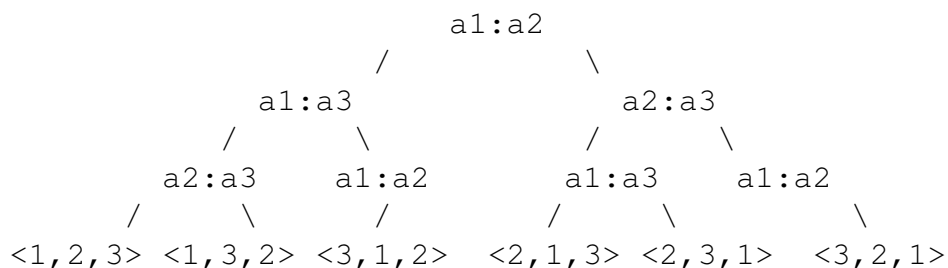
**Decision trees:** Given an input sequence  $\langle a_1, a_2, \dots, a_n \rangle$ ,

- Each internal node is denoted by  $a_i : a_j$ , for  $1 \leq i, j \leq n$ .
- Each leaf node is denoted by a permutation  $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$ .
- Each path from the root to a leaf corresponds to an execution of the sorting algorithm for a specific input.
- The left branch of an internal node means  $a_i \leq a_j$ .  
The right branch for an internal node means  $a_i > a_j$ .
- There are  $n!$  permutations for  $n$  elements  $\implies$  there are at least  $n!$  leaf nodes.

Ex: The decision tree for insertion sort with 3 elements.



Ex: The decision tree for selection sort with 3 elements.



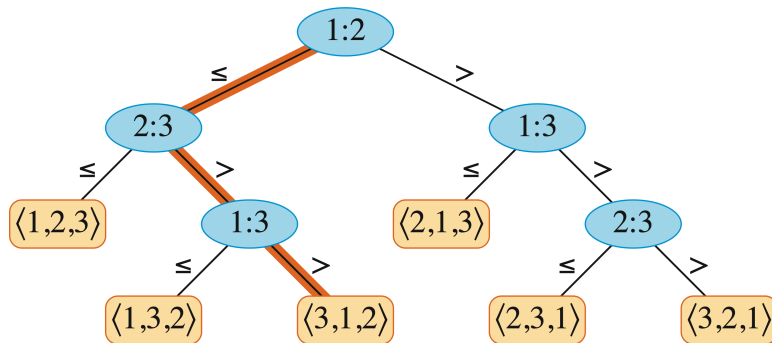
There are  $n!$  permutations for  $n$  elements  $\implies$  the tree has at least  $n!$  leaves.

Let  $h$  be the height of the tree  $\implies$  the tree has no more than  $2^h$  leaves.

Thus,

$$n! \leq 2^h \implies h \geq \log(n!) \implies h = \Omega(n \log n)$$

The height of a decision tree means the number of comparisons for sorting in the worst-case.



$\implies$  All comparison sorts have a lower bound running time  $\Omega(n \log n)$  for the worst-case.

$\implies$  It is impossible to find a new comparison based sorting algorithm that is asymptotically better than merge sort.

However, some non-comparison based sorting algorithms may run in linear time.

## Counting Sort

Counting sort assumes that each of the  $n$  input elements is an integer within a range  $[0..k]$ , for some integer  $k$ .

An input array  $A[1 : n]$ , an output array  $B[1 : n]$ , and a temporary working storage  $C[0 : k]$  are necessary for this algorithm. Thus, counting sort does not sort in place.

During the execution of counting sort,  $C[i]$  maintains the # of elements less than or equal to  $i$ . For each element  $j$  in  $A$ , we put it into  $B$  at position  $C[j]$ .

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

```

1. let  $B[1:n]$  and  $C[0:k]$  be new arrays
2. for  $i = 0$  to  $k$ 
3.      $C[i] = 0$ 
4. for  $j = 1$  to  $n$ 
5.      $C[A[j]] = C[A[j]] + 1$ 
6. //  $C[i]$  contains the # of elements that is equal to  $i$ 
7. for  $i = 1$  to  $k$ 
8.      $C[i] = C[i] + C[i-1]$ 
9. //  $C[i]$  now contains the # 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.      $B[C[A[j]]] = A[j]$ 
13.      $C[A[j]] = C[A[j]] - 1$  // to handle duplicate values

```

Ex:

	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)

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

(b)

	1	2	3	4	5	6	7	8
B						3		

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

(c)

	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)

	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)

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

(f)

- Running time analysis:

Counting-Sort's running time is  $\Theta(n + k)$ .

If  $k = O(n)$ , then  $\Theta(n + k) = \Theta(n)$ . It's a linear time!

- Counting sort is a **stable** sorting algorithm: elements with the same value in the output array should be in the same order as they do in the input array.
    - Insertion Sort: stable (if no “=” sign in comparison)
    - Selection Sort: stable (if no “=” sign in comparison)
    - Merge Sort: stable (if the “=” sign is in the comparison)
    - Heap Sort: not stable (exchange  $A[1] \rightarrow A[n]$ )
    - Quick Sort: not stable.
- Ex: input:  $\langle 5, 5', 5'', 3, 4 \rangle$  and the output is  $\langle 3, 4, 5'', 5, 5' \rangle$ .

# Radix Sort

The Radix-Sort sorts by the least significant digit first, then by the 2nd least significant digit, ....

The sorting algorithm used to sort each digit should be stable; otherwise Radix-Sort will not work.

Ex:

213		321		312		123
312		312		212		132
123		212		213		212
212	stable	132	stable	321	stable	213
321	----->	213	----->	123	----->	312
132		123		132		321
^		^		^		

Ex:

213		321		312		123
312		312		213 <-		132
123		212		212 <-		213 <-
212	stable	132	not stable	321	stable	212 <-
321	----->	213	----->	123	----->	312
132		123		132		321
^		^		^		

Another example:

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(A, d)

1. for i = 1 to d
2.     use a stable sort to sort array A[1:n] on digit i

Two questions:

- Why does the algorithm need to use a stable sort to sort each digit?
- Why does the sorting start from sorting the least significant digit first?

Running time analysis: Suppose all  $n$  numbers have  $d$  or less digits.

If we use Counting-Sort as the sorting algorithm to sort each digit, then the running time for Radix-Sort is  $d \cdot \Theta(n+k) = \Theta(dn+dk)$

If  $k = O(n)$  and  $d$  is a constant, then the running time becomes  $\Theta(n)$ .

It's a linear time!

**Recommended Exercise:** Show how to sort  $n$  integers in the range 0 to  $n^2 - 1$  in  $O(n)$  time.

**Solution:**

We will assume that each digit has value in the range  $0..n-1$ , that is  $k = n$ . That is, as if the numbers are written in radix- $n$  or base- $n$  (instead of the usual base 2 or base 10).

Counting sort now requires  $O(n+k) = O(n)$  time.

Then each number will have two digits, so  $d = 2$  as the range of the numbers is  $[0..n^2 - 1]$ .

Since  $d = 2$ , radix-sort requires two passes of Counting sort that each take  $O(n)$  time. This the total run-time for radix-sort for this type of input is  $O(n)$ .

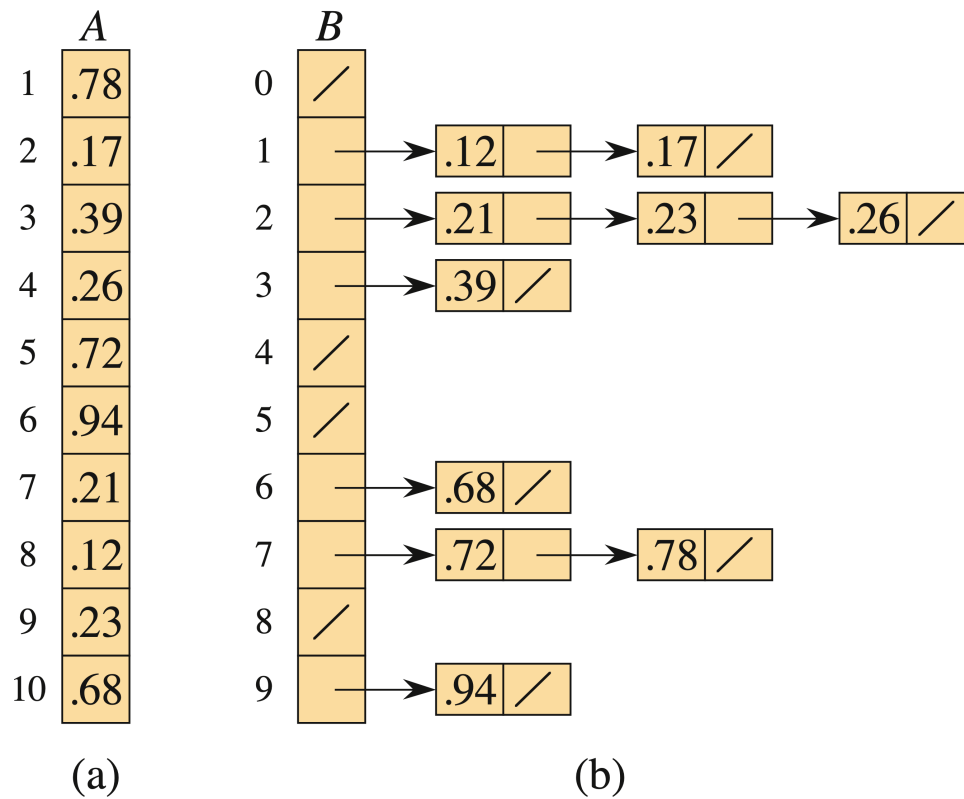
## Bucket sort

Assumption: input is drawn from the range  $[0..1)$  with a uniform probability distribution.

Average case is  $O(n)$ .

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[floor(n A[i])]
6. for i = 0 to n - 1
7.     sort B[i] using insertion sort
8. concatenate the lists B[0], B[1], ..., B[n-1] together in order
9. return the concatenated lists



If each list is of size  $O(1)$ , then the run-time is  $O(n)$ . The average case analysis requires advanced math so we will skip it here.

**Notes:** If the key values are distributed a known probability distribution, we can still get  $O(n)$  average case. How can we take advantage of knowing the probability distribution to get all buckets to be of size  $O(1)$  on the average?