SORTING IN LINEAR TIME

Analysis of Algorithm

Punjab University College of Information Technology (PUCIT)
University of the Punjab, Lahore, Pakistan.

Credit

- These notes contain material from Chapter 8 of Cormen, Leiserson, Rivest, and Stein (3rd Edition).

Comparison Sorts

- If sorting algorithms perform sorting by comparisons between the input elements, we call such sorting algorithms ***comparison sorts.***

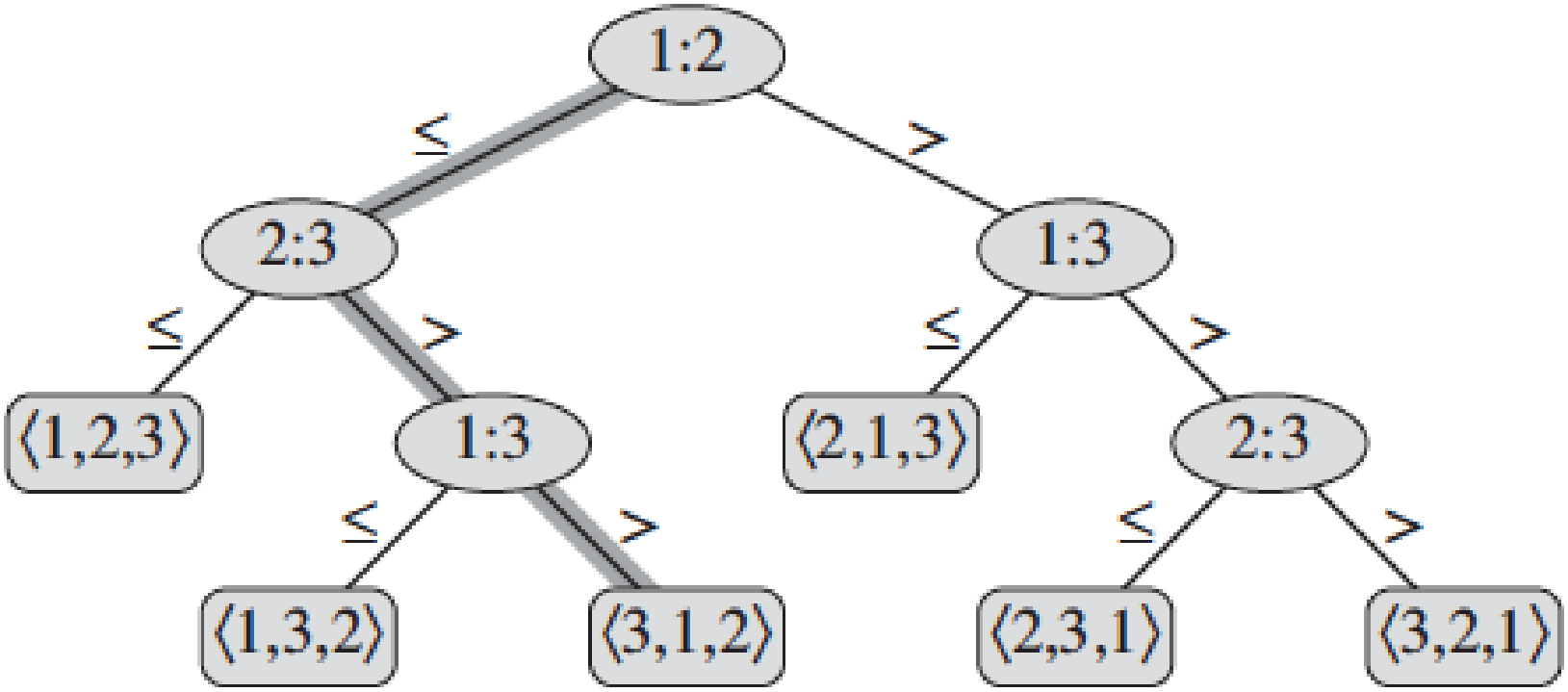
- Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Bubble Sort are few examples of comparison sorts.

Comparison Sorts (Cont.)

Any comparison sort makes **Ω(nlgn)** comparisons in worst case let see how!

- In a comparison sort, we use only comparisons between elements to gain order
- Lets assume we need to learn order of three input numbers represented as 1, 2, and 3. Lets draw a decision tree!

Comparison Sorts (Cont.)



The decision tree using three elements

Comparison Sorts (Cont.)

Consider a decision tree of height **h** with **l** reachable leaves corresponding to a comparison sort on **n** elements. Because each of the n! permutations of the input appears as some leaf, we have **$n! \le l$** . Since a binary tree of height **h** has no more than **2^h** leaves, we have

$$n! \le l \le 2^h ,$$

which, by taking logarithms, implies

$$\begin{aligned} h &\ge \lg(n!) && \text{(since the lg function is monotonically increasing)} \\ &= \Omega(n \lg n) && \text{(by equation (3.19))} . \end{aligned}$$

Stilring's approximation

Counting Sort

Counting sort assumes that each of the *n* input elements is an integer in the range *1* to k, for some integer *k.* When *k* = *O(n),* the sort runs in linear time.

The idea: determine, for each input element *x,* the number of elements less than *x*. This allows one to determine *x*'s position in the sorted array. The idea is used to handle also cases of ties.

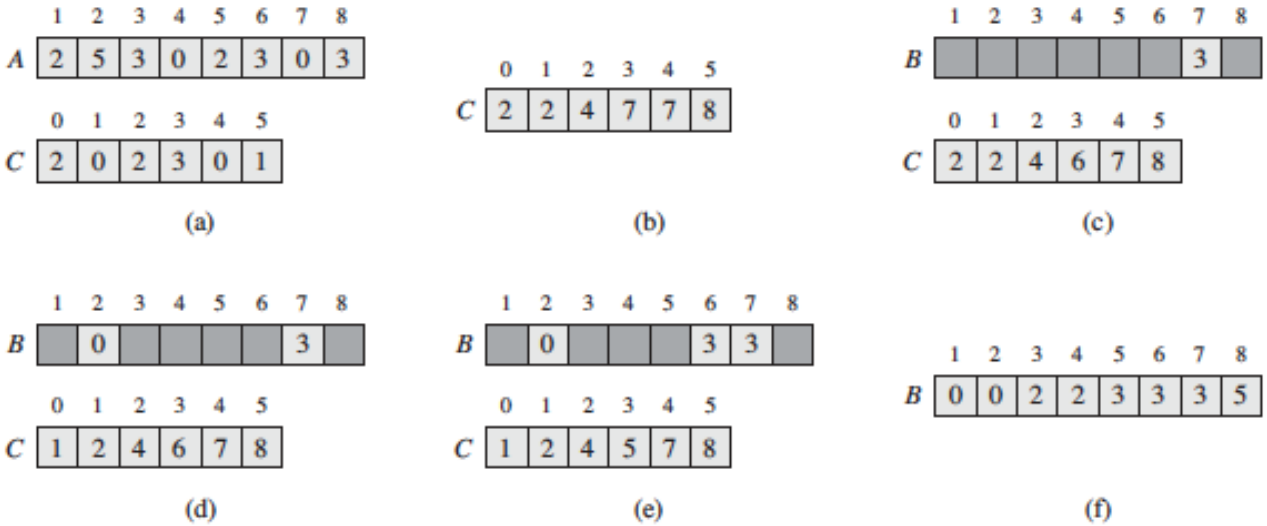
Counting Sort (Cont.)

The code makes use of the following arrays:

- *A[1..n]* is the input array, with *length[A]* = *n.*
- *B[1..n]* holds the sorted output.
- *C[1..k]* provides temporary storage.

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
```



Counting Sort (Cont.)

How much time does counting sort require? The **for** loop of lines 2–3 takes time $\Theta(k)$, the **for** loop of lines 4–5 takes time $\Theta(n)$, the **for** loop of lines 7–8 takes time $\Theta(k)$, and the **for** loop of lines 10–12 takes time $\Theta(n)$. Thus, the overall time is $\Theta(k + n)$. In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$.

Stable Sort?

Counting sort is stable: numbers with the same value appear in the output array in the same order as they do in the input array. That is, ties between two numbers are broken by the rule that whichever number appears first in the input array appears first in the output array.

Quick sort is unstable sort!

What is the importance of stable sort?

Radix Sort

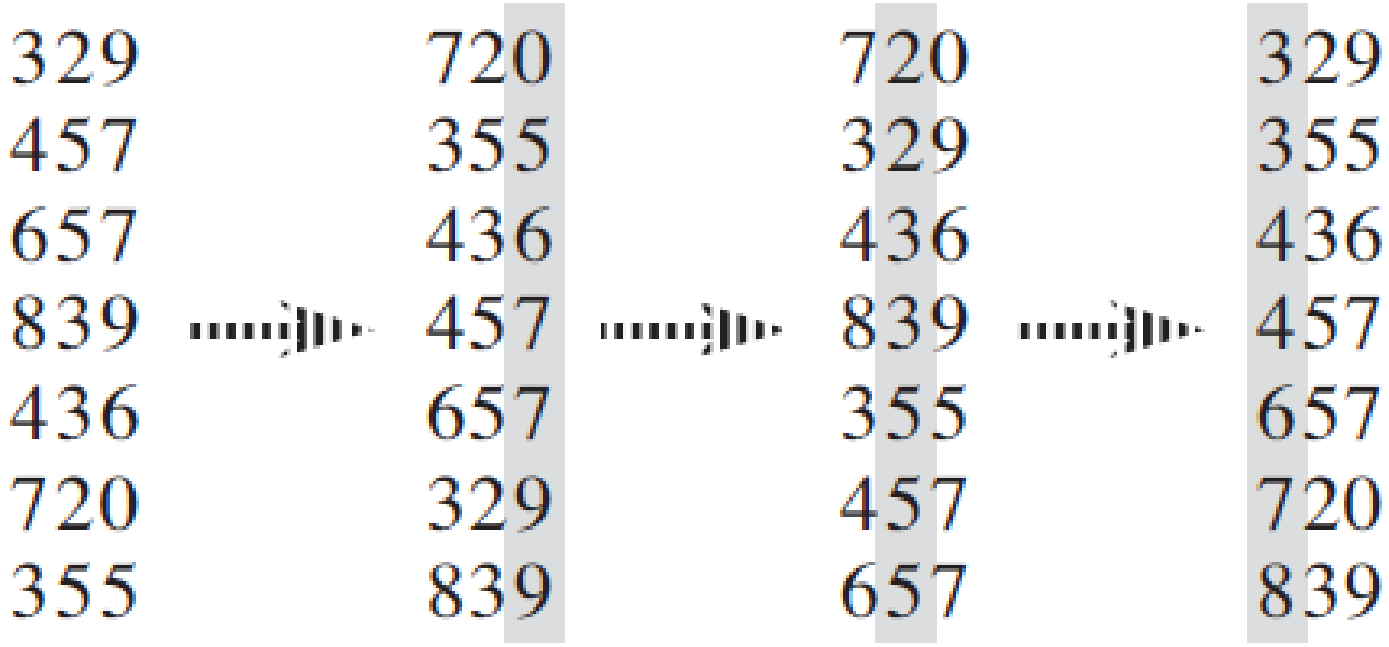
The Algorithm*:* There are two inputs: the array of numbers denoted by *A*, and the number of digits in each number is denoted by *d*.

$\text{Radix-Sort}(A, d)$

for $i \neg 1$ to d

do use a stable sort to sort array A on digit i

Radix Sort (Cont.)



Radix Sort (Cont.)

Running Time:

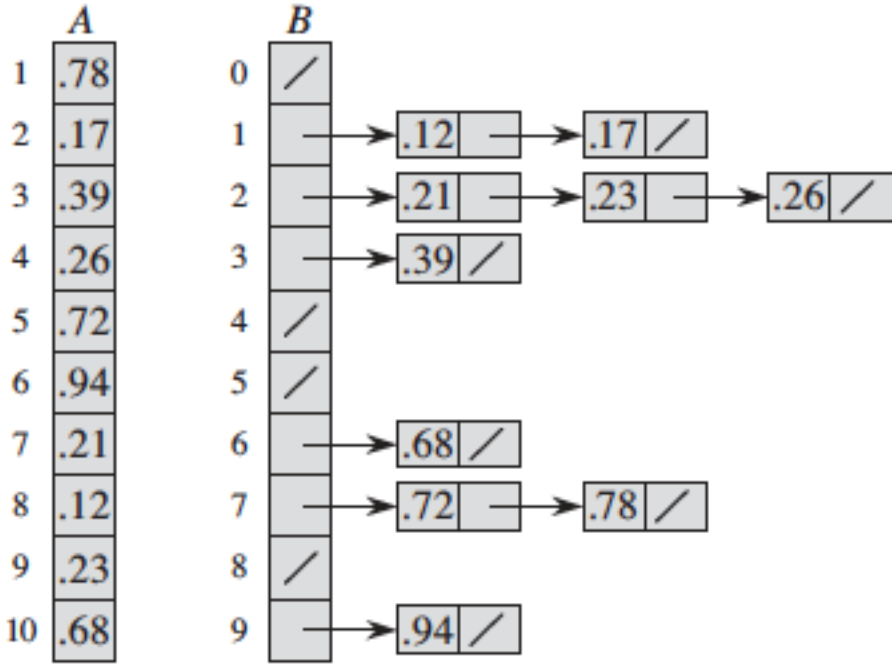
Assume that each digit is in the range 0,..,k-1 and we use Counting sort to sort each column. Then the running time is $O(d(n+k))$. If $k = O(n)$ and d is a constant then the running time is linear.

Bucket Sort

- It is fast because it assumes that the input is generated by a random process that distributes elements uniformly over the interval [*0,1).*

- The idea is to divide the interval into ***n*** equal-sized subintervals, or ***buckets***, and then distribute the ***n*** input numbers into the buckets. All that is left is to sort the numbers in each bucket, and then simply go from one bucket to the other, in order, and collect the numbers.

Bucket Sort (Cont.)




```
BUCKET-SORT(A)
1  let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4       make B[i] an empty list
5  for i = 1 to n
6       insert A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8       sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

Bucket Sort (Cont.)

Running time: Almost everything happens in ***O(n)*** units of time. The only difficult part to analyze is the sorting within the different buckets (Line 7-8): this part depends on the number of elements that “fall” into each bucket.

Lets analyze the algorithm!

Summary

- Comparison Sorts takes **nlgn** comparisons to determine the order of input elements

- Counting Sort, Radix Sort, and Bucket Sort give linear running time but with few limitations!