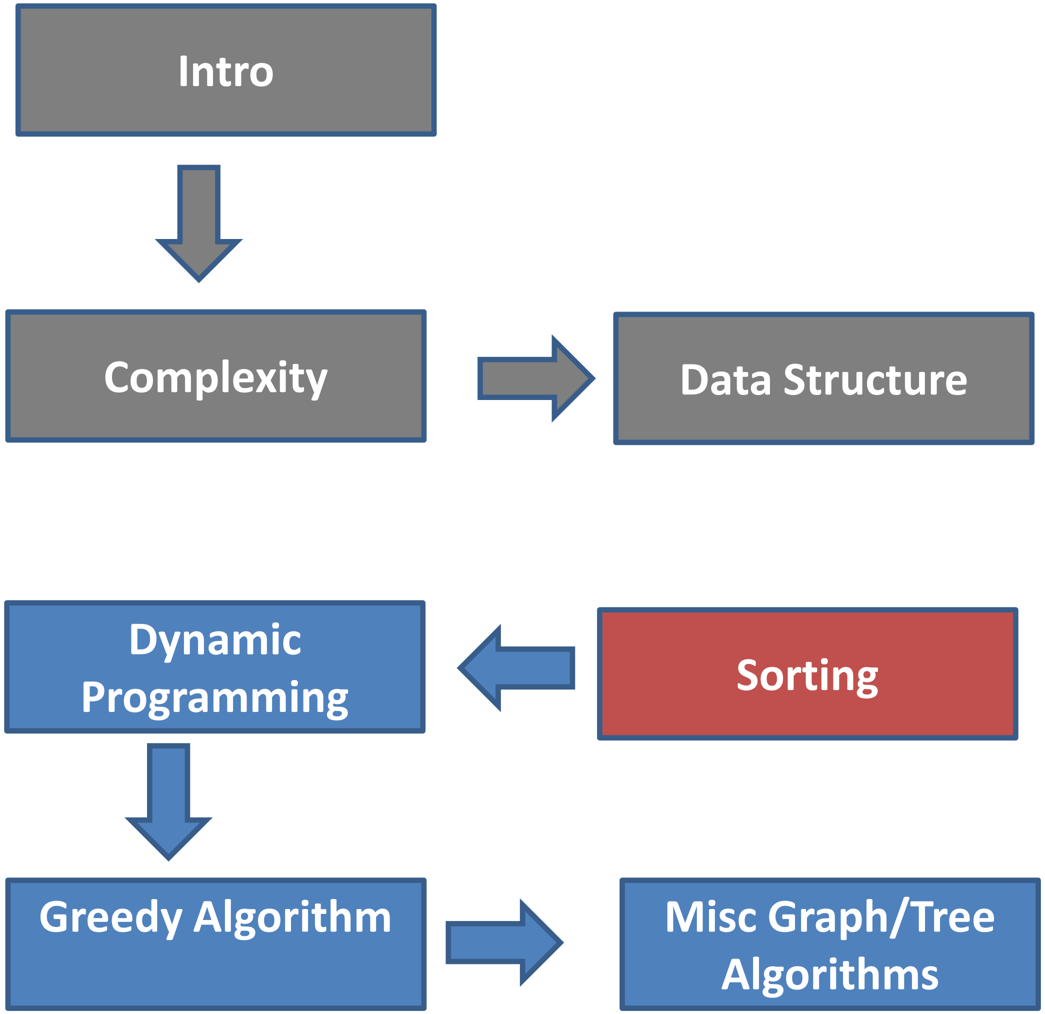
An Introduction to Algorithms

By

Hossein Rahmani

h_rahmani@iust.ac.ir

http://webpages.iust.ac.ir/h_rahmani/



Intro
Complexity
Data Structure
Trees
Hash Functions
Sorting
Dynamic Programming
Greedy Algorithm
Misc Graph/Tree Algorithms

Comparison Sorting Review

- Insertion sort:
  - Pro's:
    - Easy to code
    - Fast on small inputs (less than ~50 elements)
    - Fast on nearly-sorted inputs
  - Con's:
    - $O(n^2)$ worst case
    - $O(n^2)$ average case

Comparison Sorting Review

- Merge sort:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort sub-arrays
    - Linear-time merge step
  - Pro’s:
    - O(n lg n) worst case
  - Con’s:
    - Doesn’t sort in place

Comparison Sorting Review

- Heap sort:
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children's keys
  - Pro's:
    - O(n lg n) worst case
    - Sorts in place
  - Con's:
    - Fair amount of shuffling memory around

Comparison Sorting Review

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two sub-arrays, recursively sort
    - All of first sub-array < all of second sub-array
  - Pro's:
    - O(n lg n) average case
    - Sorts in place
    - Fast in practice
  - Con's:
    - O(n^2) worst case
      - Naïve implementation: worst case on sorted input
      - Good partitioning makes this very unlikely.

Non-Comparison Based Sorting

- Many times we have restrictions on our keys
  - Social Security Numbers
  - Employee ID's
- We will examine three algorithms which under certain conditions can run in $O(n)$ time.
  - Counting sort
  - Radix sort
  - Bucket sort

Counting Sort

- Depends on <u>assumption</u> about the numbers being sorted
  - Assume <u>numbers</u> are in the <u>range *1.. k*</u>
- The algorithm:
  - <u>Input</u>: A[1..*n*], where A[j] $\in$ {1, 2, 3, …, *k*}
  - <u>Output</u>: B[1..*n*], sorted (not sorted in place)
  - Also: Array C[1..*k*] for <u>auxiliary</u> storage

Counting Sort

```
CountingSort(A, B, k)
    for i=1 to k
        C[i]= 0;
    for j=1 to n
        C[A[j]] += 1;
    for i=2 to k
        C[i] = C[i] + C[i-1];
    for j=n downto 1
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```

This is called a ***histogram***.

Counting Sort Example

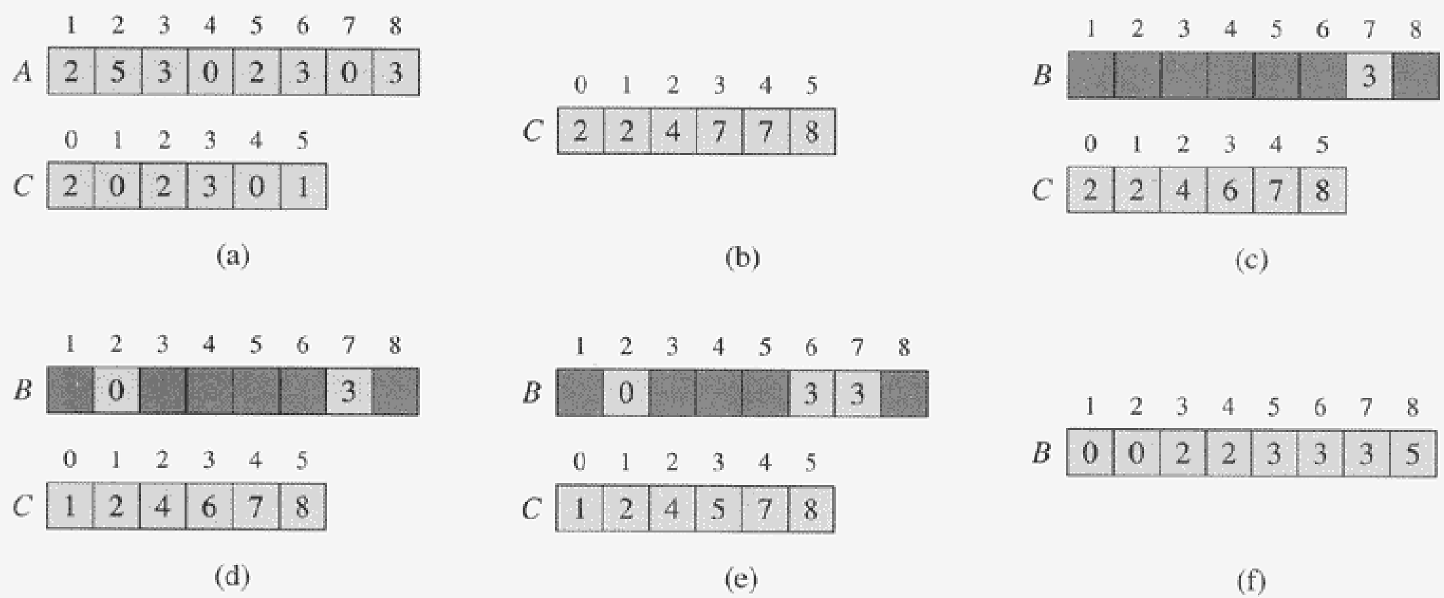


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 9–11, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

Counting Sort

```
CountingSort(A, B, k)
    for i=1 to k
        C[i]= 0;
    for j=1 to n
        C[A[j]] += 1;
    for i=2 to k
        C[i] = C[i] + C[i-1];
    for j=n downto 1
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```

Takes time O(k)

Takes time O(n)

What is the running time?

Total time: $O(n + k)$

Why don't we always use counting sort?

Depends on range k of elements.

Counting Sort Review

- **Assumption:** input taken from **small** set of **numbers** of size k
- Basic idea:
  - Count number of elements less than you for each element.
  - This gives the position of that number – similar to selection sort.
- Pro's:
  - Fast … $O(n+k)$
  - Simple to code
- Con's:
  - Doesn't sort in place.
  - Elements must be integers. ***countable***
  - Requires $O(n+k)$ extra storage.

Radix Sort

- Intuitively, you might sort on the most significant digit, then the second msd, etc.
- Problem: lots of intermediate piles of information to keep track of
- Key idea: sort the *least* significant digit first

```
RadixSort(A, d)
    for i=1 to d
        StableSort(A) on digit i
```

Radix Sort 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

Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

Radix Sort Correctness

- Sketch of an inductive proof of correctness (induction on the number of passes):
  - Assume lower-order digits $\{j: j<i\}$ are sorted
  - Show that sorting next digit *i* leaves array correctly sorted
    - If <u>two digits at position *i* are different</u>, ordering numbers by that digit is <u>correct</u> (lower-order digits irrelevant)
    - If they are the <u>same</u>, numbers are already sorted on the lower-order digits. Since we use a <u>stable sort</u>, the numbers stay in the right order

Radix Sort

- *What sort is used to sort on digits?*
- Counting sort is obvious choice:
  - Sort n numbers on digits that range from 1..k
  - Time: $O(n + k)$
- Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$

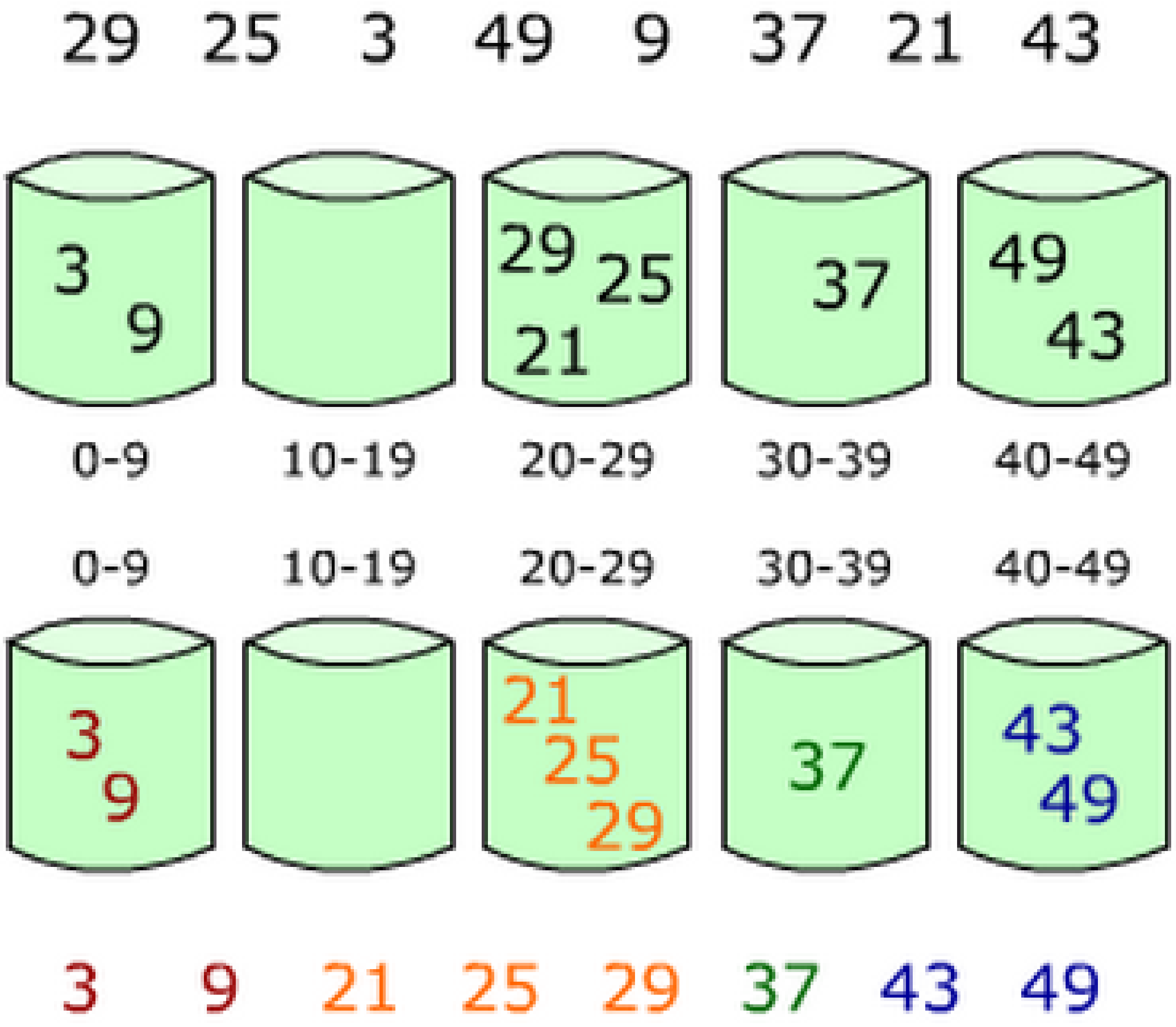
Radix Sort Review

- **Assumption:** input has *d* digits ranging from 0 to *k*
- Basic idea:
  - Sort elements by digit starting with *least* significant
  - Use a stable sort (like counting sort) for each stage
- Pro's:
  - Fast
  - Simple to code
- Con's:
  - Doesn't sort in place

Bucket Sort

Assumption: input elements distributed uniformly over some known range, e.g., [0,1), so all elements in A are greater than or equal to 0 but less than 1 . (Appendix C.2 has definition of uniform distribution)

1. Set up an array of initially empty "buckets".
2. **Scatter**: Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket.
4. **Gather**: Visit the buckets in order and put all elements back into the original array.


29 25 3 49 9 37 21 43
3
9
29
25
21
37
49
43
0-9
10-19
20-29
30-39
40-49
0-9
10-19
20-29
30-39
40-49
3
9
21
25
29
37
43
49
3 9 21 25 29 37 43 49

BUCKET-SORT(A)

```
let B[0 .. n - 1] be a new array
n = A.length
for i = 0 to n - 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊n A[i]⌋]
for i = 0 to n - 1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n - 1] together in order
```

Bucket Sort Review

- **Assumption:** input is uniformly distributed across a range
- Basic idea:
  - Partition the range into a fixed number of buckets.
  - Toss each element into its appropriate bucket.
  - Sort each bucket.
- Pro's:
  - Fast
  - Simple to code
- Con's:
  - Doesn't sort in place

Quiz 1

- What is the main characteristics of the sorting algorithm used in the Radix sort? Can we use Quick sort to improve the original implementation?

Quiz 2

- Discuss the complexity of linear sorting algorithms (Counting/Radix/Bucket) in worst/average/best cases?

Quiz 3

- How we could improve the Divide and Conquer approach?