(CS2009)-DESIGN AND ANALYSIS OF ALGORITHMS

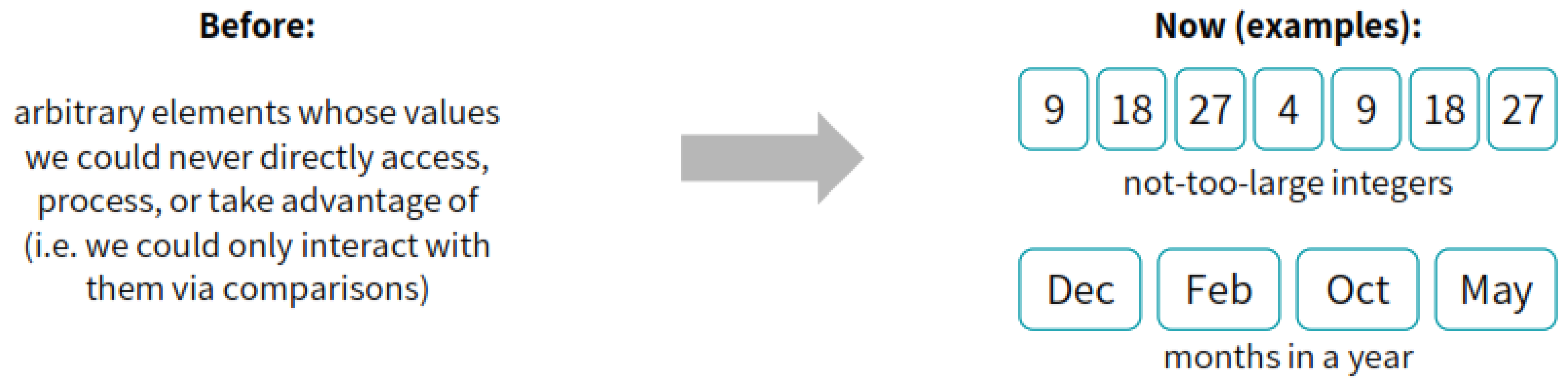
Presented By: Sandesh Kumar
Email: sandesh.kumar@nu.edu.pk

LECTURE 14-15
LINEAR TIME SORTING

Beyond comparison-based sorting algorithms!

A NEW MODEL OF COMPUTATION

- The elements we're working with have meaningful values


Before:
arbitrary elements whose values we could never directly access, process, or take advantage of (i.e. we could only interact with them via comparisons)
Now (examples):
9 18 27 4 9 18 27
not-too-large integers
Dec Feb Oct May
months in a year

- The worst-case complexity can be reduced further from "n.logn" without making comparisons, called linear sorting. Counting, Radix and Bucket sort are three examples.

- However, it is possible only under restrictive circumstances, for example, sorting small integers (exam scores), characters etc.

COUNTING SORT

- It is an integer sorting algorithm used in computer science to collect objects according to keys that are small positive integers.
- It works by determining the positions of each key value in the output sequence by counting the number of objects with distinct key values and applying prefix sum to those counts.
- We assume that there are only k different possible values in the array (and we know these k values in advance)
- For example: elements are integers in {10, 20, 30, 40, 50, 60}
- **https://www.cs.usfca.edu/~galles/visualization/CountingSort.html**

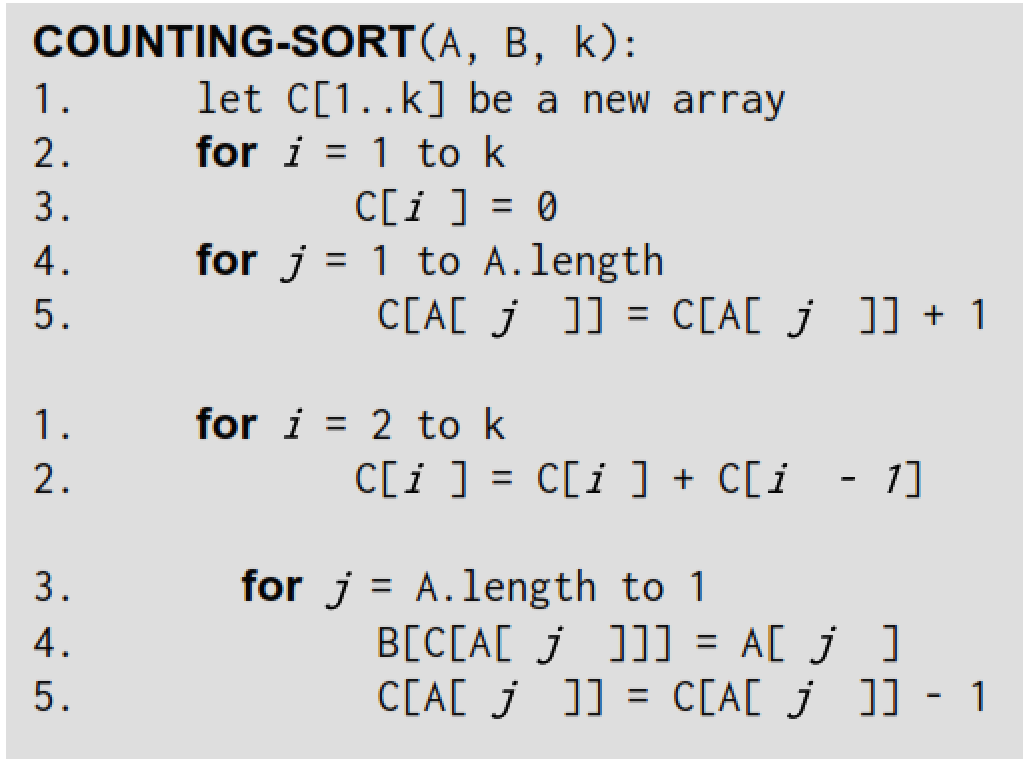
COUNTING SORT

- Input array A[1,…K]; k (elements in A have values from 1 to K)
- Output: sorted array A

Algorithm:

1. Create a counter array C[1,…,K]
2. Create an auxiliary array B[1,…,n]
3. Scan A once, record element frequency in C
4. Calculate prefix sum in C
5. Scan A in the reverse order, and copy each element to B at the correct position according to C.
6. Copy B to A

COUNTING SORT: PSEUDOCODE



```
COUNTING-SORT(A, B, k):
1.    let C[1..k] be a new array
2.    for i = 1 to k
3.        C[i ] = 0
4.    for j = 1 to A.length
5.        C[A[ j  ]] = C[A[ j  ]] + 1

1.    for i = 2 to k
2.        C[i ] = C[i ] + C[i  - 1]

3.    for j = A.length to 1
4.        B[C[A[ j  ]]] = A[ j  ]
5.        C[A[ j  ]] = C[A[ j  ]] - 1
```

ANALYSIS OF COUNTING SORT

- Input array A[1,…K]; k (elements in A have values from 1 to K)
- Output: sorted array A

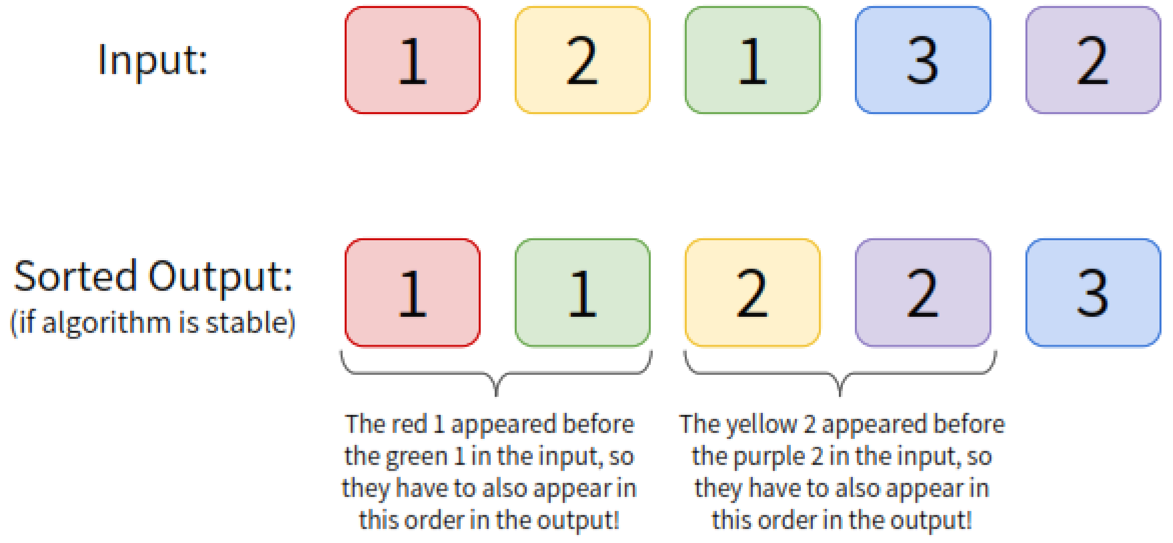
Algorithm:	Time	Space
1. Create a counter array C[1,…,K]		$O(k)$
2. Create an auxiliary array B[1,…,n]		$O(n)$
3. Scan A once, record element frequency in C	$O(n)$	
4. Calculate prefix sum in C	$O(k)$	
5. Scan A in the reverse order, and copy each element to B at the correct position according to C.	$O(n)$	
6. Copy B to A	$O(n)$	
	$O(n+k) = O(n)$ (if $k = O(n)$)	$O(n+k) = O(n)$ (if $k = O(n)$)

COUNTING SORT

- Is counting sort stable
  - The input order is maintained among items with equal keys
- Cool! *Why don't we always use counting sort?*
  - Because it depends on the range *k* of elements
- *Could we use counting sort to sort 32-bit integers? Why or why not?*
  - Answer: no, *k* is too large ($2^{32} = 4,294,967,296$)

STABLE SORTING

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input data set


Input:
1
2
1
3
2
Sorted Output:
(if algorithm is stable)
1
1
2
2
3
The red 1 appeared before the green 1 in the input, so they have to also appear in this order in the output!
The yellow 2 appeared before the purple 2 in the input, so they have to also appear in this order in the output!

RADIX SORT

- Radix sort is a small method that many people intuitively use when alphabetizing a large list of names. (Here Radix is 26, 26 letters of the alphabet). Specifically, the list of names is first sorted according to the first letter of each name, that is, the names are arranged in 26 classes.
- Intuitively, one might want to sort numbers on their most significant digit. But Radix sort does counter-intuitively by sorting on the least significant digits first. On the first pass, entire numbers are sorted on the least significant digit and combined in an array. Then on the second pass, the entire numbers are sorted again on the second least-significant digits and combined in an array and so on.

RADIX SORT

- The following example shows how Radix sort operates on seven 3-digit numbers

INPUT	1st pass	2nd pass	3rd pass
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

- **https://www.cs.usfca.edu/~galles/visualization/RadixSort.html**

RADIX SORT

- Represents keys as d-digit numbers in some base-k
  - key = $x_1x_2...x_d$ where $0 \le x_i \le k-1$
- Example: key=15
  - key_{10} = 15, d=2, k=10   where $0 \le x_i \le 9$
  - key_2 = 1111, d=4, k=2   where $0 \le x_i \le 1$

Assumptions: d=O(1)   and k =O(n)

- Sorting looks at one column at a time
  - For a d digit number, sort the least significant digit first
  - Continue sorting on the next least significant digit, until all digits have been sorted
  - Requires only d passes through the list

RADIX SORT ANALYSIS

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

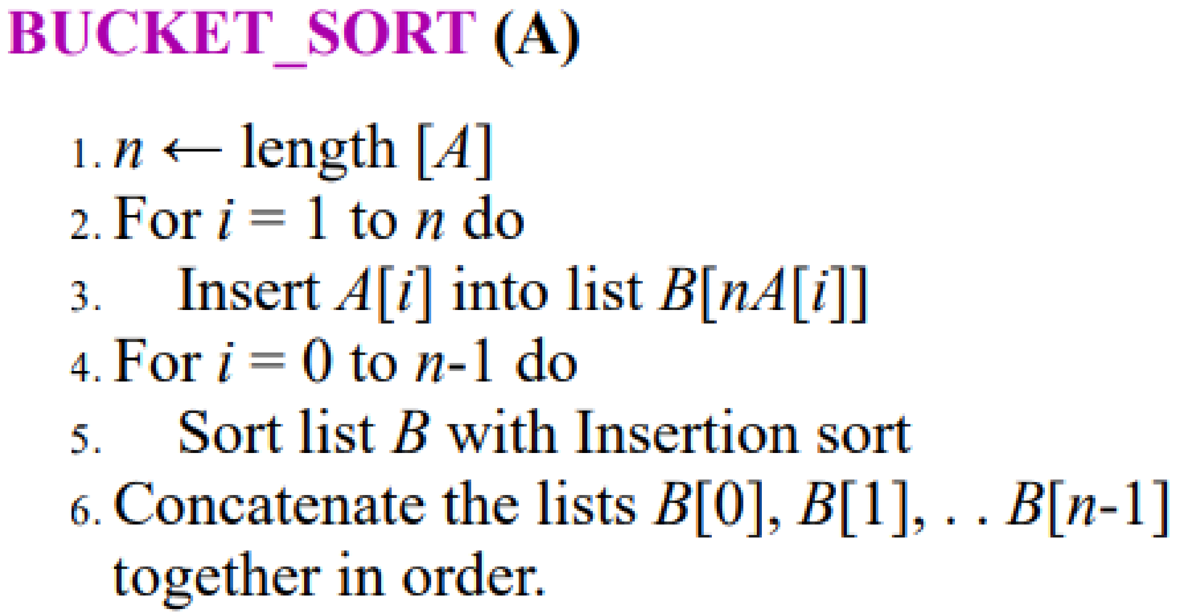
- The running time depends on the stable used as an intermediate sorting algorithm. When each digit is in the range 1 to k, and k is not too large, COUNTING_SORT is the obvious choice. In case of counting sort, each pass over n d-digit numbers takes O(n+k) time. There are d passes, so the total time for Radix sort is $\theta(n+k)$ time. There are d passes, so the total time for Radix sort is $\theta(dn+dk)$. When d is constant and $\mathrm{k}= \theta(n)$, the radix sort runs in linear time.

BUCKET SORT

- Bucket sort runs in linear time on the average. It assumes that the input is generated by a random process that distributes elements uniformly over the interval [0, 1).
- The idea of Bucket sort is to divide the interval [0, 1) into n equal-sized subintervals, or buckets, and then distribute the n input numbers into the buckets. Since the inputs are uniformly distributed over (0, 1), we don't expect many numbers to fall into each bucket. To produce the output, simply sort the numbers in each bucket and then go through the bucket in order, listing the elements in each.
- https://www.cs.usfca.edu/~galles/visualization/BucketSort.html

BUCKET SORT

- The code assumes that input is in n-element array A and each element in A satisfies 0 ≤ A[i] ≤ 1. We also need an auxiliary array B[0 . . n -1] for linked-lists (buckets).

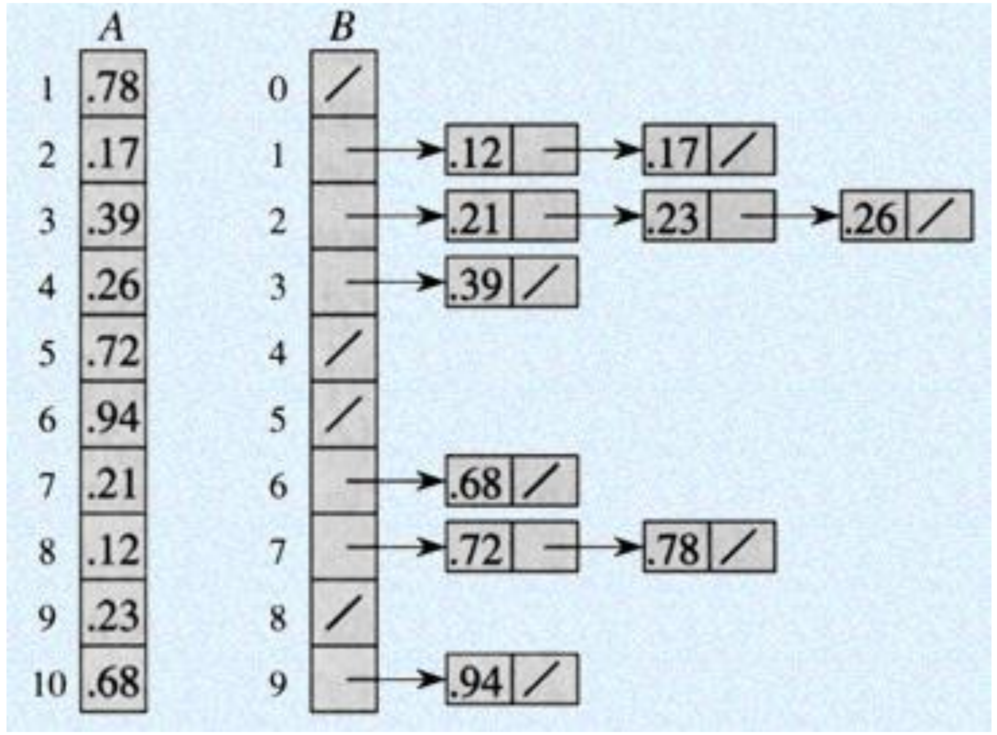


```
BUCKET_SORT (A)

1. $n \leftarrow$ length [$A$]
2. For $i$ = 1 to $n$ do
3.     Insert $A[i]$ into list $B[nA[i]]$
4. For $i$ = 0 to $n$-1 do
5.     Sort list $B$ with Insertion sort
6. Concatenate the lists $B[0]$, $B[1]$, . . $B[n-1]$
   together in order.
```

BUCKET SORT

- **Example:** Given input array A[1..10]. The array B[0..9] of sorted lists or buckets after line 5. Bucket i holds values in the interval [*i*/10, (*i* +1)/10]. The sorted output consists of a concatenation in order of the lists first B[0] then B[1] then B[2] ... and the last one is B[9].



BUCKET SORT

- **Analysis:** All lines except line 5 take O(n) time in the worst case. We can see inspection that total time to examine all buckets in line 5 is O(n-1) i.e., O(n).
- The only interesting part of the analysis is the time taken by Insertion sort in line 5. Let n_i be the random variable denoting the number of elements in the bucket $B[i]$. Since the expected time to sort by INSERTION_SORT is O(n^2), the expected time to sort the elements in bucket B[i] is

$$E[O({}^2n_i)] = O(E[{}^2n_i]]$$

Therefore, the total expected time to sort all elements in all buckets is

$$^{n-1}\sum_{i=0} O(E[{}^2n_i]) = O\ ^{n-1}\sum_{i=0} (E[{}^2n_i]) \qquad \text{------------ A}$$