Department of
Software Engineering
BAHRIA UNIVERSITY
Discovering Knowledge

Lecture 4

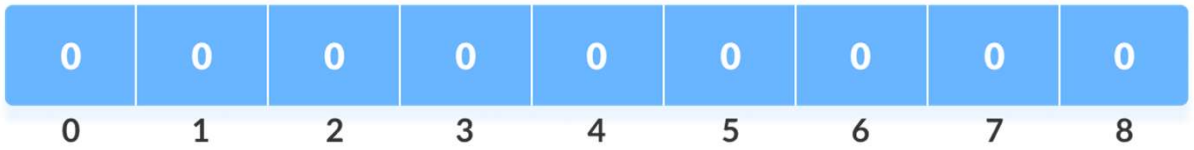
Sorting Algorithms

More Sorting Algorithms

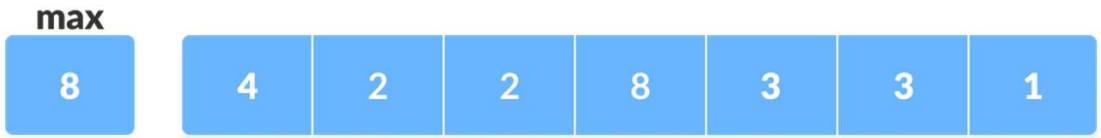
- Counting Sort
  - Countingsort works if the values we are sorting are integers that lie in a relatively small range
  - For example, if we need to sort 1 million integers with values between 0 and 1,000, coutingsort can provide amazingly fast performance
  - The basic idea behind countingsort is to count the number of items in the array that have each value
  - Then it is relatively easy to copy each value, in order, the required number of times back into the array

- How Counting Sort Works?
- Find out the maximum element (let it be max) from the given array.

- Initialize an array of length max+1 with all elements 0. This array is used for storing the count of the elements in the array.
-

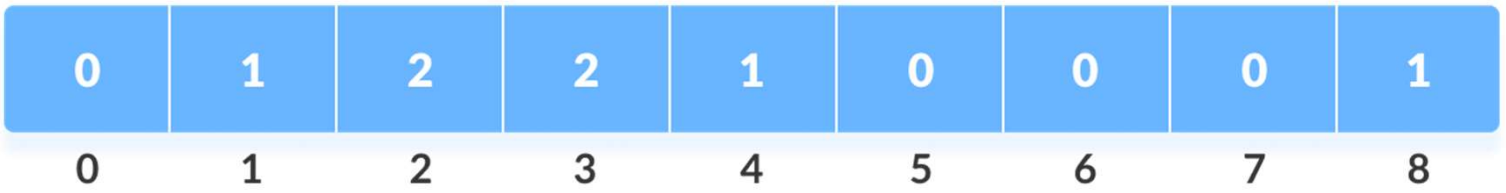


- Store the count of each element at their respective index in count array

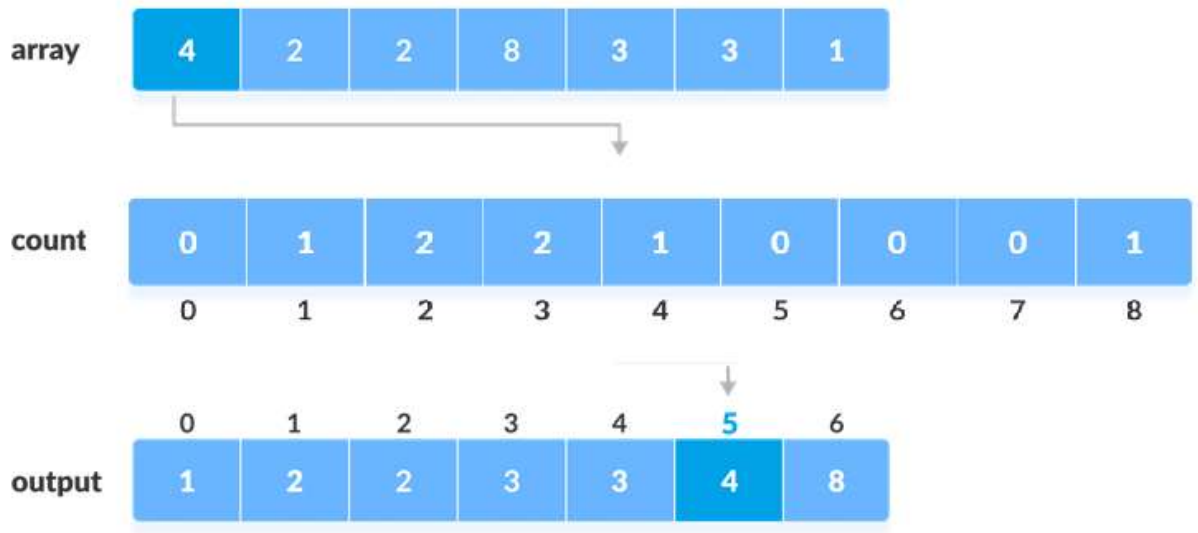


- For example: if the count of element 3 is 2 then, 2 is stored in the 3rd position of count array. If element "5" is not present in the array, then 0 is stored in 5th position

- Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.



- Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as shown in figure below.



- After placing each element at its correct position, decrease its count by one.

More Sorting Algorithms

- Counting Sort (contd.)

```
countingSort(array, size)
  max <- find largest element in array
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique element and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```

More Sorting Algorithms

- Counting Sort (contd.)
  - Let M be the number of items in the counts array (so M = max_value +1) and let N be the number of items in the **values** array, if your programming language doesn't automatically initialize the **counts** array so that it contains 0s, the algorithm spends M steps initializing the array. It then takes N steps to count the values in the array
  - The algorithm finishes by copying the values back into the original array
  - Each value is copied once, so that part takes N steps. If any of the entries in the **counts** array is still 0, the program also spends some time skipping over that entry
  - In the worst-case, if all the values are the same, so that the counts array contains mostly 0s, it takes M steps to skip over the 0 entities
  - That makes the total runtime O(2 * N + M) = O(N+M)
  - If M is relatively small compared to N, this is much smaller than the O(N log N) performance given by other algorithms previously
  - In one test, quicksort(worst-case) took 4.29 seconds to sort 1 million items with values between 0 and 1,000, but it took countingsort only 0.03 seconds.
  - With Similar values, heapsort took roughtly 1.02 seconds

Bucket Sort

- Bucket sort works by partitioning the elements into buckets and the return the result
- Buckets are assigned based on each element’s search key
- To return the result, concatenate each bucket and return as a single array

Bucket Sort

- Some variations
  - Make enough buckets so that each will only hold one element, use a count for duplicates
  - Use fewer buckets and then sort the contents of each bucket
  - Radix sort (which I'll demonstrate next)

- The more buckets you use, the faster the algorithm will run but it uses more memory

Bucket Sort

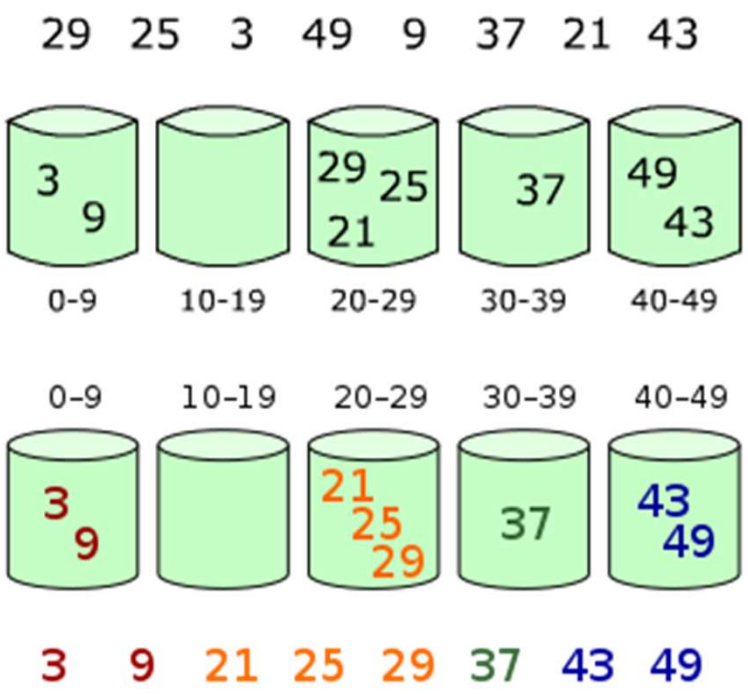
- Time complexity is reduced when the number of items per bucket is evenly distributed and as close to 1 per bucket as possible

- Buckets require extra space, so we are trading increased space consumption for a lower time complexity

- In fact Bucket Sort beats all other sorting routines in time complexity but can require a lot of space

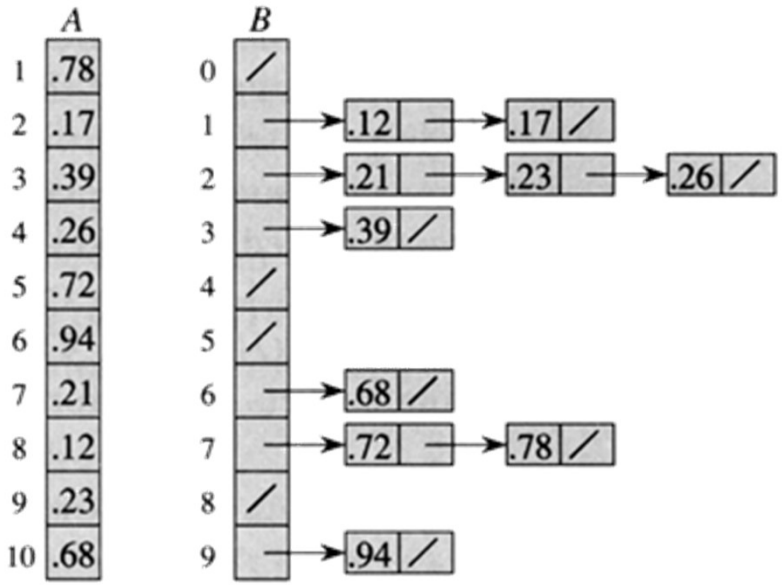
Bucket Sort

Multiple items per bucket:



Bucket Sort

In array form:



Bucket Sort Algorithm

```
bucketSort()
  create N buckets each of which can hold a range of values
  for all the buckets
    initialize each bucket with 0 values
  for all the buckets
    put elements into buckets matching the range
  for all the buckets
    sort elements in each bucket
  gather elements from each bucket
end bucketSort
```

Bucket Sort Animation

https://www.cs.usfca.edu/~galles/visualization/BucketSort.html

Linked List array index= Value* #of Elements/(Max Value+1)

Bucket Sort Complexity

- Worst Case Complexity: O(n2)
  - When there are elements of close range in the array, they are likely to be placed in the same bucket. This may result in some buckets having more number of elements than others.
  - It makes the complexity depend on the sorting algorithm used to sort the elements of the bucket.
  - The complexity becomes even worse when the elements are in reverse order. If insertion sort is used to sort elements of the bucket, then the time complexity becomes O(n2).
- Best Case Complexity: O(n+k)
  - It occurs when the elements are uniformly distributed in the buckets with a nearly equal number of elements in each bucket.
  - The complexity becomes even better if the elements inside the buckets are already sorted.
  - If insertion sort is used to sort elements of a bucket then the overall complexity in the best case will be linear ie. O(n+k). O(n) is the complexity for making the buckets and O(k) is the complexity for sorting the elements of the bucket using algorithms having linear time complexity at the best case.
- Average Case Complexity: O(n)
  - It occurs when the elements are distributed randomly in the array. Even if the elements are not distributed uniformly, bucket sort runs in linear time. It holds true until the sum of the squares of the bucket sizes is linear in the total number of elements.

Radix Sort

- Improves on bucket sort by reducing the number of buckets
- Maintains time complexity of O(n)

- Radix sort executes a bucket sort for each significant digit in the data-set
  - 100's would require 3 bucket sorts
  - 100000's would require 6 bucket sorts

Radix Sort

Sort: 36 9 0 25 1 49 64 16 81 4

First Buckets:

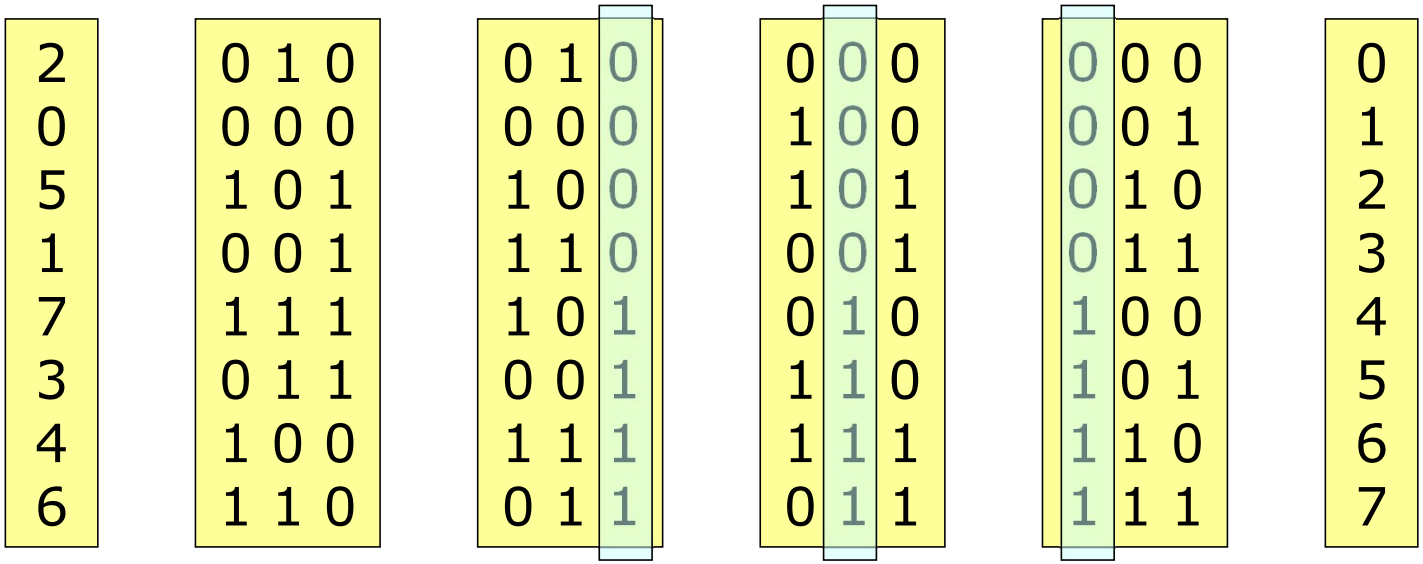
Bin	0	1	2	3	4	5	6	7	8	9
Content	0	1 81	-	-	64 4	25	36 16	-	-	9 49

Second Buckets:

Bin	0	1	2	3	4	5	6	7	8	9
Content	0 1 4 9	16	25	36	49	-	64	-	81	-

Radix sort

- Example:



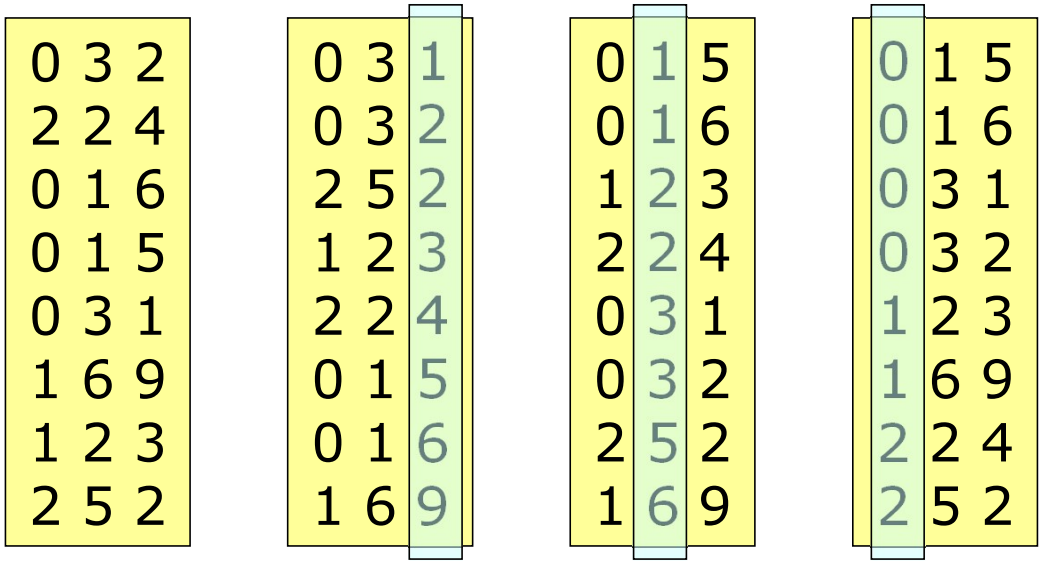
Each sorting step must be stable.

Radix sort characteristics

- Each sorting step can be performed via bucket sort, and is thus $O(N)$.
- If the numbers are all b bits long, then there are b sorting steps.
- Hence, radix sort is $O(bN)$.

What about non-binary?

- Radix sort can be used for decimal numbers and alphanumeric strings.



Radix Sort Animation

- http://www.cs.auckland.ac.nz/software/AlgAnim/radixsort.html

More Sorting Algorithms

- Summary – Sorting Algorithms

Algorithm	Runtime	Techniques	Usefulness
Insertionsort	$O(N^2)$	Insertion	Very small arrays
Selectionsort	$O(N^2)$	Selection	Very small arrays
Bubblesort	$O(N^2)$	Two-way passes, restricting bounds of interest	Very small arrays, mostly sorted arrays
Heapsort	O(N logN)	Heaps, storing complete trees in an array	Large arrays with unknown distribution
Quicksort	O(N logN) expected, $O(N^2)$ worst case	Divide-and-conquer, swapping items into position, randomization to avoid worst-case behaviour	Large arrays without too many duplicates, parallel sorting
Mergesort	O(N logN)	Divide-and-conquer, merging, external sorting	Large arrays with unknown distribution, parallel sorting
Coutingsort	O(N+M)	Counting	Large arrays of integers with a limited range of values
Bucketsort	O(N + M)	Buckets	Large arrays with reasonably uniform value distribution