Sorting in Linear Time:
fastest (but special purpose) sorting algorithms

Debswapna Bhattacharya, PhD
Assistant Professor
Computer Science and Software Engineering
Auburn University

Comparison Sorts

- Sorting by comparing pairs of numbers
- Algorithms that sort n numbers in $O(n^2)$ time
  - Insertion, Selection, Bubble
- Algorithms that sort n numbers in O(nlgn) time
  - Merge, Heap and Quick

Comparison Sorts

Any comparison sort must make **Ω(nlgn)** comparisons.

So O(nlgn) is the most efficient they can be!

Now we will talk about three sorting algorithms – counting, radix and bucket – that are linear

So they must use a strategy other than comparing pairs of numbers.

Counting Sort

- ***Assume that each of the n input elements is an integer in the range 0 to k for some integer k.***

COUNTING_SORT(A,B,k)

let C[0…k] be a new array

for $i = 0$ **to** k

$c[i] = 0$

for $j = 1$ **to** A.length

$c[A[j]] = c[A[j]] + 1$

// $c[i]$ now contains the number of elements equal to i

for $i = 1$ **to** k

$c[i] = c[i] + c[i-1]$

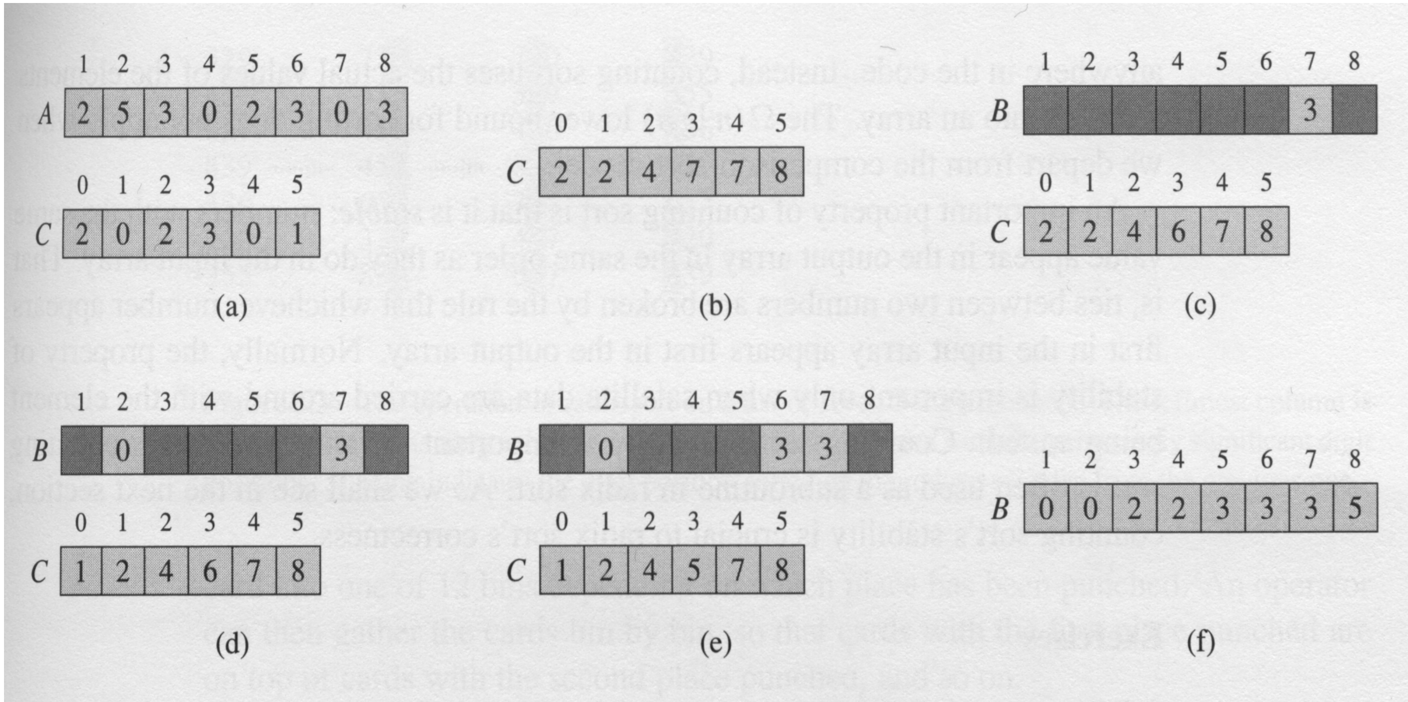
// $c[i]$ now contains the number of elements less than or equal to i

for $j = A.length$ **downto** 1

$B[c[A[j]]] = A[j]$

$c[A[j]] = c[A[j]] - 1$

The operation of Counting-sort on an input array A[1..8]



COUNTING_SORT(A,B,k)

let C[0…k] be a new array

for $i = 0$ **to** k

$c[i] = 0$

for $j = 1$ **to** A.length

$c[A[j]] = c[A[j]] + 1$

// $c[i]$ now contains the number of elements equal to i

for $i = 1$ **to** k

$c[i] = c[i] + c[i-1]$

// $c[i]$ now contains the number of elements less than or equal to i

for $j = A.length$ **downto** 1

$B[c[A[j]]] = A[j]$

$c[A[j]] = c[A[j]] - 1$

Approximate Analysis: $\Theta(k+n)$

$\Theta(n)$ when $k < n$

$\Theta(k)$ when $k > n$

What if k ≫ n?

Counting sort is *not* an *in-place* algorithm.

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.)

Counting Sort Thinking Assignments

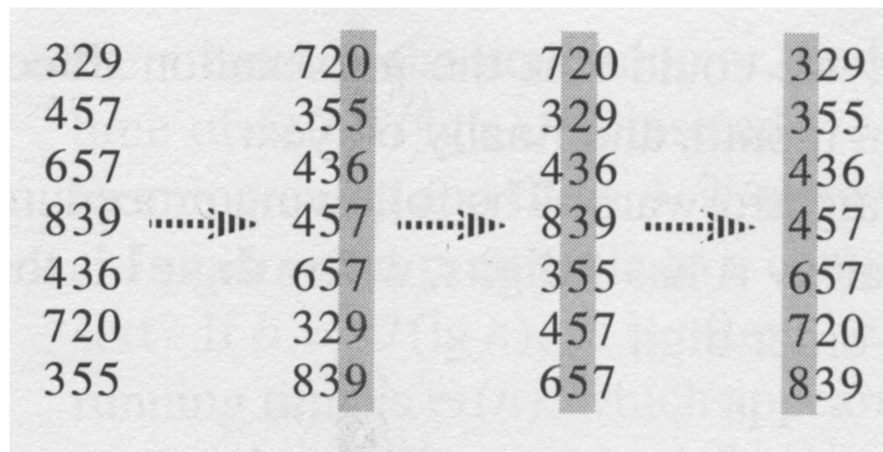
- Modify the algorithm to sort n integers in the range i-j, i<j, where i>0
- Modify the algorithm to sort n integers in the range i-j, i<j, where i!=0 and either or both of i and j may be negative

Radix Sort

RADIX_SORT(A, d)

for i = 1 **to** d

    use a stable sort to sort array A on digit i



Given n d-digit numbers in which each digit can take on up to k possible values, Radix sort correctly sorts these number in $\Theta(d(n + k))$ time if Counting Sort is used.

k = 10 so for large n, n+k can be approximated as n, Radix sort runs in $\Theta(dn) = \Theta(n)$ time because d is also a small constant.

Radix Sort Thinking Assignments

- Why is it important that the sorting algorithm used by Radix sort be stable?

- Radix sort is not an in-place algorithm…why?

Radix Sort Thinking Assignments

- Suppose you were to use the Counting Sort algorithm in step 2 of Radix Sort. Will Radix Sort work correctly for all negative integers? Why or why not? If not, can you modify Radix and/or Counting Sort to make this work?

- Suppose you were to use the Counting Sort algorithm in step 2 of Radix Sort. Will Radix Sort work correctly for mixed +ve and –ve integers? Why or why not? If not, can you modify Radix and/or Counting Sort to make this work?

Bucket Sort

BUCKET_SORT(*A*)

$n = A.length$

let B[0…n-1] be a new array

for $i = 0$ **to** *n-1*

    make B[i] an empty list

for $i = 1$ **to** *n*

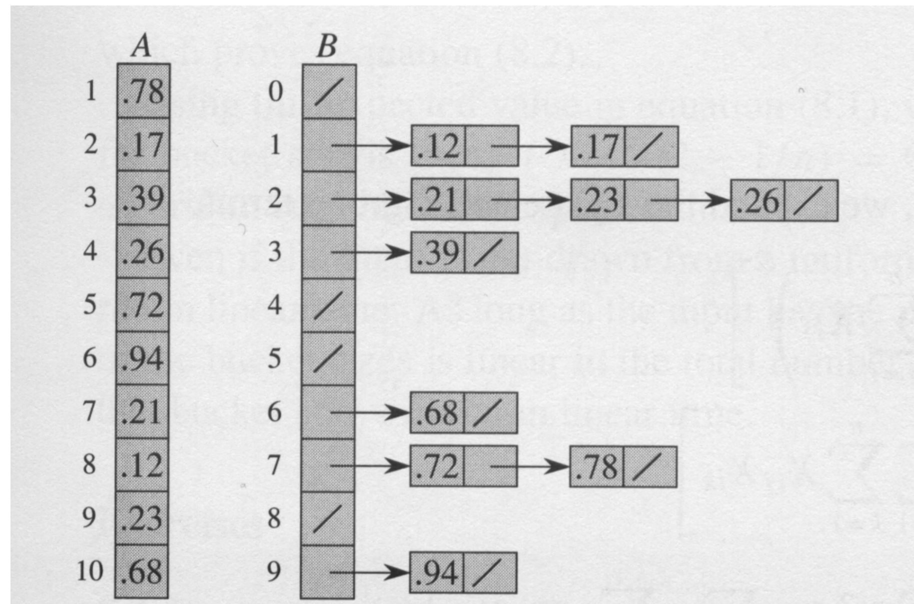
    insert $A[i]$ into

        list $B\lfloor nA[i] \rfloor$

for $i = 0$ **to** *n-1*

    sort list $B[i]$ with insertion sort

concatenate $B[0], B[1], \ldots, B[n-1]$ in order



Bucket Sort Complexity

Bucket sort is linear, O(n), because if the numbers are distributed uniformly across each of the n buckets, each bucket will only have <u>on average</u> **one** number in it.

Bucket Sort Thinking Assignments

- The algorithm requires an input of real numbers uniformly distributed over the interval [0,1). Why is it important that the input to Bucket sort be uniformly distributed over this interval?

- Instead of simply adding new numbers to the head of the linked lists and later sorting the lists using Insertion Sort, you can maintain sorted linked lists by adding a new number in its correct sorted location in step 6 and do away with steps 7 and 8. Will this change the complexity of the algorithm? If so, how? If not, why not?

Bucket Sort Thinking Assignments

- The algorithm uses n buckets for n input numbers in the range [0,1) and some buckets may be empty while others contain more than one number and need to be sorted. But if we restrict the input to n integers in a range [0,k] as was the case for Counting Sort, and if you use k+1 buckets, you can avoid having to use Insertion sort. Why?

- How will you modify the Bucket Sort algorithm to accomplish this? Compare this modified algorithm with Counting Sort in terms of space and time efficiency.

- How will you modify the algorithm step 6 to accommodate real number input from some interval [0,p), p>1, not [0,1)?

- How will you modify the algorithm step 6 to accommodate real number input from some interval [i,j), not [0,1)?

Ch. 8 Reading Assignments

Omit 8.1

Read 8.2

Read 8.3 (omit lemmas and proofs)

Read 8.4 (omit average case analysis p.202-203)

Ch. 8 Thinking Assignments

Do Exercises:

8.2-1:8.2-4

8.3-1:8.3-3

8.4-1:8.4-2

Why study sorting?

A universal problem! Many applications incorporate sorting.

There is a wide variety of sorting algorithms, and they use a rich set of techniques. So sorting provides a good case study of a variety of algorithm design techniques resulting in algorithms of differing complexities.

Popular Sorting Algorithms

Quadratic Sorts $O(n^2)$

Bubble, Insertion, Selection

A sort that improved on this (of historical interest only)

Shell Sort $O(n^{1.5})$

Efficient Sorts $O(n \lg n)$

Merge, Heap, Quick

Linear (specialized) sorts $O(n)$

Counting, Radix, Bucket

For a lager list, see https://en.wikipedia.org/wiki/Sorting_algorithm

Four important properties

- Time Efficiency
- Space Efficiency: in-place
- Stability
- Recursive/Non-recursive

Thinking Assignments

- Which of the sorting algorithms we discussed are in-place?
- Which of the sorting algorithms we discussed are stable?