Sorting in linear time

Chapter 8

Omit Section 8.1 and parts from other sections as stated in the slides

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 Sort

Comparison sort

Any comparison sort must make Ω(nlgn) comparisons.

(know this result but omit its proof in the following slides and section 8.1 of the text)

So nlgn is the most efficient they can be!

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

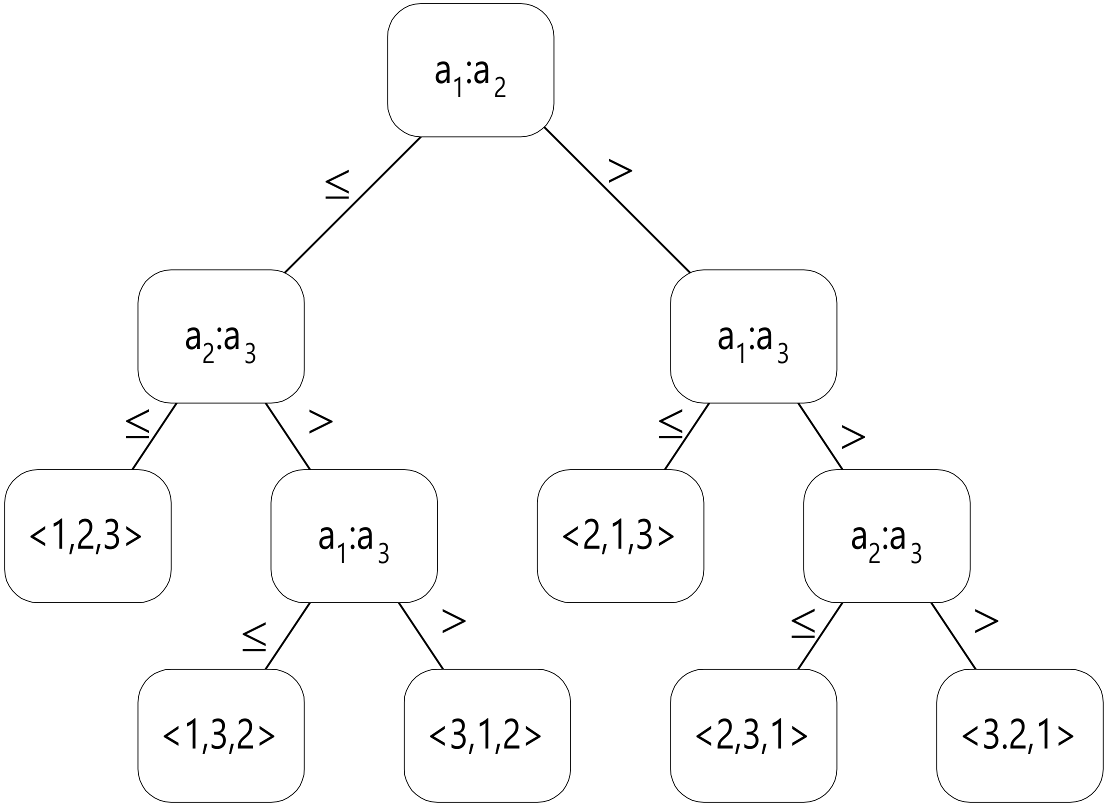
So they must use a strategy other than comparing pairs of numbers.

Decision Trees (omit)

- A representation of the comparisons a comparison sort algorithm makes
- A full binary tree
- Each node represents a comparison
- Each child represents one of two cases of that comparison
- Each leaf represents a sorted order
- An execution of the algorithm: a path from the root to a leaf

Insertion Sort Decision Tree (omit)

The decision tree model



Decision Tree (omit)

Any correct sorting algorithm must be able to produce any permutation of its input…why?

So if a comparison sort is correct each of the n! permutations must be a leaf of its decision tree AND each of these leaves must be reachable by a path corresponding to an actual execution of the algorithm

(omit)

Theorem 9.1. Any decision tree that sorts n elements has height $\Omega(n \log n)$.

Proof:

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

$$h \ge \log(n!) = \Omega(n \lg n).$$

Corollary 9.2 Heapsort and merge sort are asymptotically optimal comparisons.

8.2 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)

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

  $c[i] = 0$

for $j = 1$ **to** length[A]

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

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

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

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

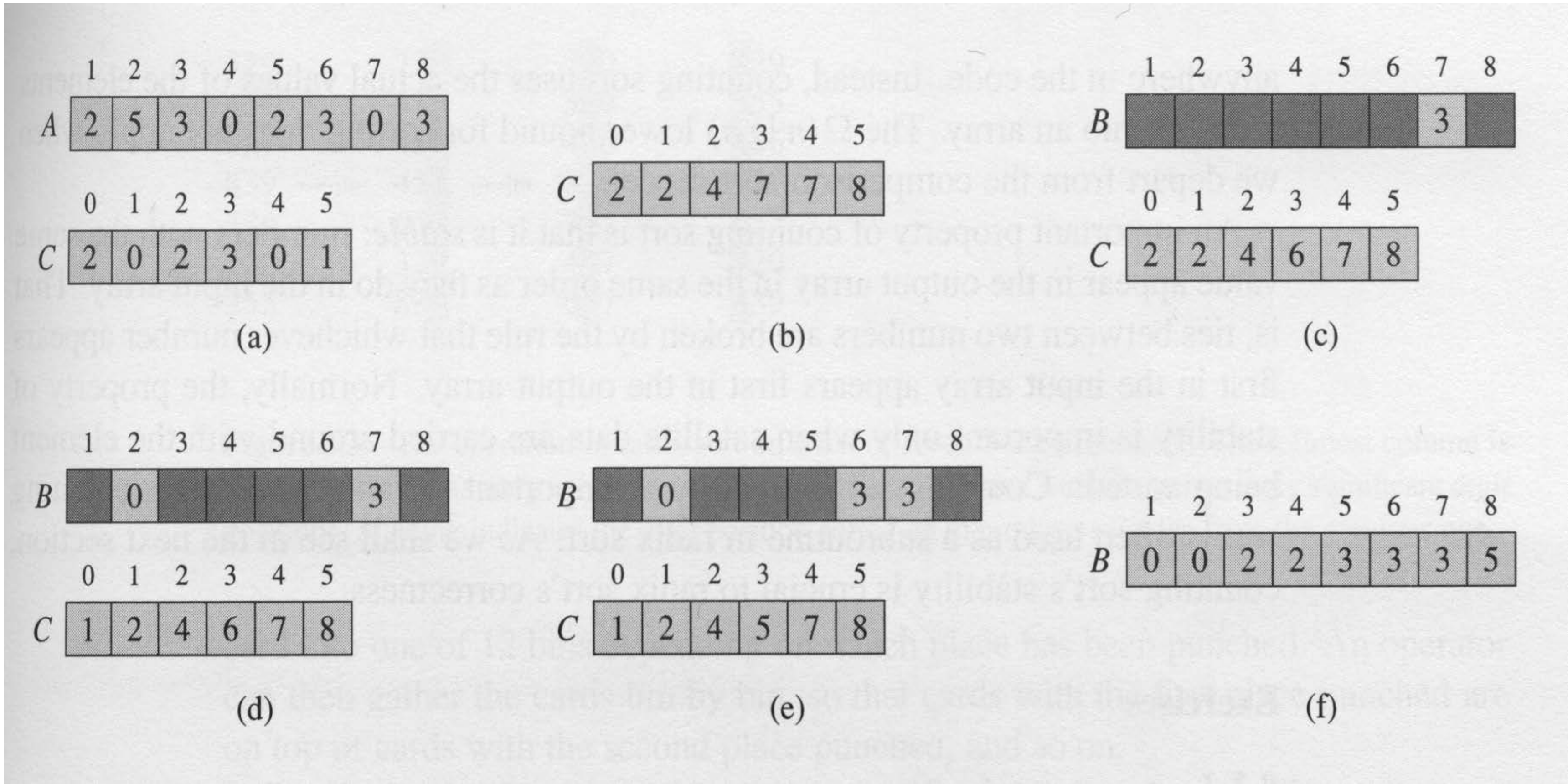
▶ $c[i]$ now contains the number of elements less than or equal to i

for $j = length[A]$ **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]



Analysis: $\Theta(k+n)$

Special case: $\Theta(n)$ when $k = O(n)$.

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.)

Thinking Assignment: Which other sorting algorithms that we have discussed are stable?

Counting Sort Thinking Assignments

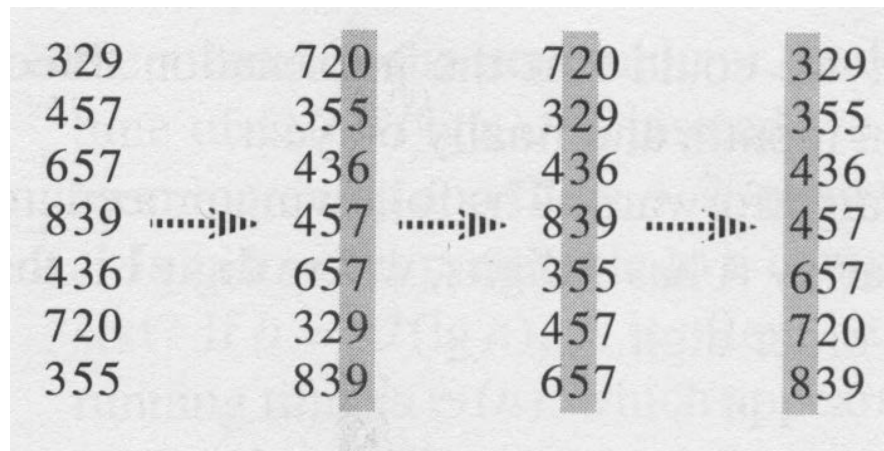
- Modify the algorithm to sort n integers in the range i-j, i<j, where i>0
- Modify the algorithm to sort n integers in the range i-j, i<j, where i!=0 and either or both of i and j may be negative
- Delete the loop in steps 6-7 and modify the rest of the algorithm appropriately so that it works correctly.

8.3 Radix sort

RADIX_SORT(A,d)

1 **for** i = 1 **to** d

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



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

Radix sort is **not** an ***in-place*** algorithm. Why?

Lemma 8.3

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.

Omit Lemma 8.4 (p.199)

Radix Sort Thinking Assignments

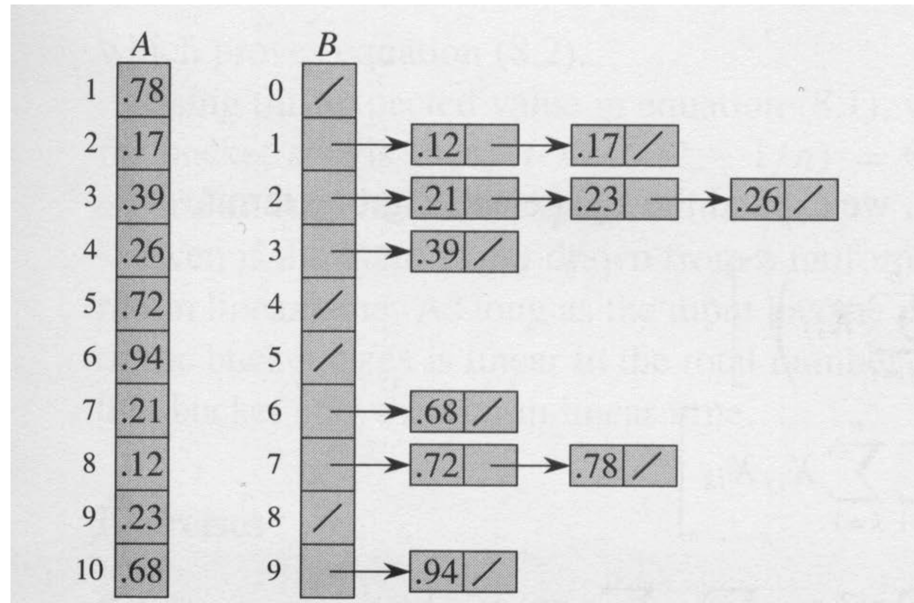
- Suppose you were to use the Counting Sort algorithm on slide #8 in step 2 of Radix Sort. Will Radix Sort work correctly for all negative integers? Why or why not? If not, can you modify Radix and/or Counting Sort to make this work?

- Suppose you were to use the Counting Sort algorithm on slide #8 in step 2 of Radix Sort. Will Radix Sort work correctly mixed +ve and –ve 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 on slide #8 in step 2 of Radix Sort. Will Radix Sort work correctly for finite decimal numbers? Why or why not? If not, can you modify Radix and/or Counting Sort to make this work?

8.4 Bucket sort

```
BUCKET_SORT(A)
    n = length[A]
    for i = 1 to n
        insert A[i] into
            list B[⌊nA[i]⌋]
    for i = 0 to n-1
        sort list B[i] with insertion sort
    concatenate B[0], B[1], ..., B[n-1] together in
      order
```



Analysis

The running time of bucket sort is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2).$$

Bucket sort is linear: $\Theta(n)$.

Omit the expected value computations in section 8.4 of text, p. 202-203

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?

- If length(A)=10 then numbers in the input array in the range [0,0.1) will all go to bucket 0. List the range of input numbers that will go to buckets 1…9.
- If length(A)=15 then list the range of input numbers that will go to buckets 0…14.
- If length(A)=n then list the range of input numbers that will go to buckets 0…(n-1).

- 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 3 and do away with steps 4 and 5. 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 step 3 of the algorithm with n buckets on slide 15 to accommodate real number input from some interval [0,p), p>1, not [0,1)?

- How will you modify step 3 of the algorithm with n buckets on slide 15 to accommodate real number input from some interval [i,j), not [0,1)?