Chapter 8
Sorting in Linear-Time

Algorithm Analysis

School of CSEE

Sorting Algorithms in linear time

- Counting Sort
- Radix Sort
- Bucket Sort

Sorting So Far

- Insertion sort:
  - Easy to code
  - Fast on small inputs (less than ~50 elements)
  - Fast on nearly-sorted inputs
  - $\Theta(n^2)$ worst case
  - $\Theta(n^2)$ average (equally-likely inputs) case

Sorting So Far

- Merge sort:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort subarrays
    - Linear-time merge step
  - $\Theta(n \lg n)$
  - Doesn't sort in place

Sorting So Far

- Heap sort:
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children's keys
  - $\Theta(n \lg n)$ worst case
  - Sorts in place

Sorting So Far

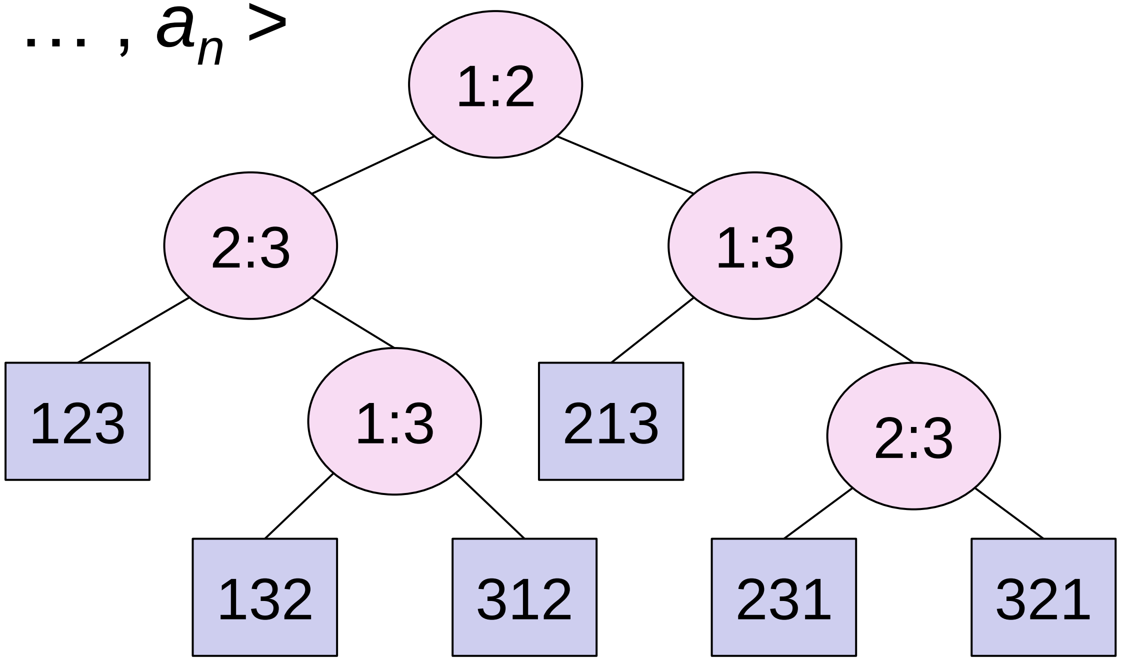
- Quick sort:
  - Divide-and-conquer:
    - Partition array into two subarrays, the recursive calling.
    - All of first subarray < all of second subarray
    - No merge step needed!
  - $\Theta(n \lg n)$ average case
  - Fast in practice
  - $\Theta(n^2)$ worst case

How Fast Can We Sort?

- We will provide a lower bound, then beat it
  - *How do you suppose we'll beat it?*
- First, an observation: all of the sorting algorithms so far are *comparison sorts*
  - The only operation used to gain ordering information about a sequence is the pairwise comparison of two elements
  - Theorem: all comparison sorts are $\Omega(n \lg n)$
    - Is this the best we can do?

Decision tree

- Sort <$a_1, a_2, \ldots, a_n$>



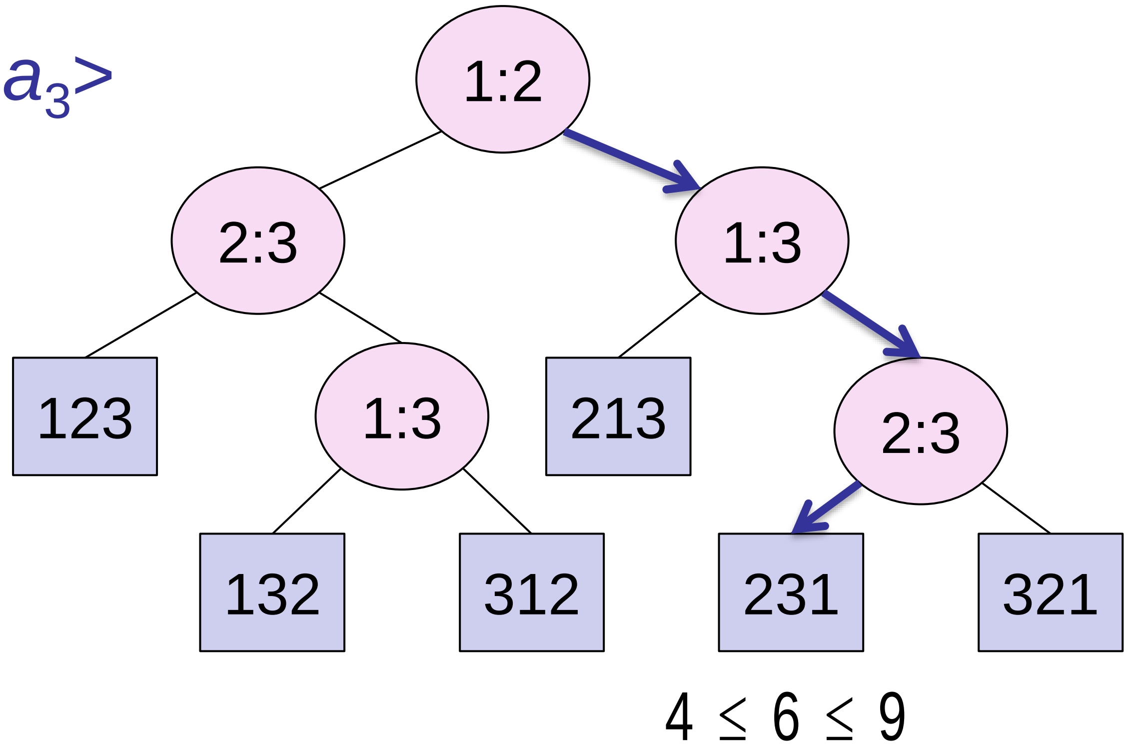
Each internal node is labeled i, j for $i, j \in \{1,2,\ldots,n\}$.

The left subtree shows subsequent comparisons if $a_i \le a_j$

The right subtree shows subsequent comparisons if $a_j \le a_i$

Decision tree example

- Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:



Each leaf contains a permutation $\{\pi(1), \pi(2), \ldots, \pi(n)\}$ to indicate that the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \ldots \le a_{\pi(n)}$ has been established

Decision tree model

A decision tree can model the execution of any comparison sort:

- One tree for each input size *n*.
- View the algorithm as slitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

Lower bound for decision tree sorting

- Theorem. Any decision tree that can sort n elements must have height $\Omega(n \lg n)$
- *Proof.* The tree must contain ≥ n! leaves, since there are n! possible permutations. A height-h binary tree has ≤ 2^h leaves. Thus, n! ≤ 2^h .

$$\therefore h \geq \lg(n!)$$

(lg is mono Increasing)

$$\geq \lg((n/e)^n)$$

(Stirling's formula)

$$= n \lg n - n \lg e$$

$$h = \Omega(n \lg n)$$

$$\sqrt{2\pi}\, n^{n+\frac{1}{2}} e^{-n} \leq n! \leq e\, n^{n+\frac{1}{2}} e^{-n}$$

Lower Bound For Comparison Sort

- Thus the time to sort n elements with comparison sort is $\Omega(n \lg n)$
- Corollary: Heapsort and Mergesort are asymptotically optimal comparison sorts
- But the name of this lecture is "Sorting in linear time"!
  - *How can we do better than $\Omega(n \lg n)$?*

What is Counting-Sort?

- Counting-Sort does not use comparisons.
- Counting-Sort uses counting to sort.
- We can sort an input array in $\Theta(n)$ time!!
- Constraints

  - Each of the n input elements should be an INTEGER.

  - Each of the n input elements should be in the range 0 to k, for some integer k.

  - It should be possible to represent that $k = O(n)$.

Sorting In Linear Time

- Input / Output:
  - Input: A[1..n], where A[j] $\in$ {1, 2, 3, …, k}
  - Output: B[1..n], sorted (notice: does not sort in place)
  - Also: Array C[1..k] for auxiliary storage is needed.
- Basic Idea
  - The basic idea of counting sort is to determine, for each input element x, the number of elements less than x.
  - This information can be used to place element x directly into its position in the output array.

Counting Sort

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

Line 1-2

```
1    for i ← 0 to k
2        do C[i] ← 0
```

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	0	0	0	0	0	0

Line 3-4

```
3  for j ← 1 to length[A]
4        do C[A[j]] ← C[A[j]] + 1
```

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C						

Line 3-4

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

▷ *C*[*i*] now contains the number of elements equal to *i*.

Line 5-6

```
5   for i ← 1 to k
6           do C[i] ← C[i] + C[i-1]
```

	0	1	2	3	4	5
C						
		2+0	2+2			

▷ *C*[*i*] now contains the number of elements less than or equal to *i*.

Line 7-9

```
7   for j ← length[A] downto 1
8       do B[C[A[j]]] ← A[j]
9           C[A[j]] ← C[A[j]] - 1
```

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3
B								
C	2	2	4	7	7	8		

Line 7-9 cont'd

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B								

	0	1	2	3	4	5
C	2	2	4	7	7	8

Analysis of counting sort

$\Theta(k)$ — for $i \leftarrow 0$ to k
  do $C[i] \leftarrow 0$

$\Theta(n)$ — for $i \leftarrow 1$ to n
  do $C[A[j]] \leftarrow C[A[j]] + 1$

$\Theta(k)$ — for $i \leftarrow 1$ to k
  do $C[i] \leftarrow C[i] + C[i - 1]$

$\Theta(n)$ — for $j \leftarrow n$ downto 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] + C[A[j]] - 1$

$$\Theta(n + k)$$

Running time

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

- But, sorting takes $\Omega(n \lg n)$ time!
- Where's the fallacy?

Answer:

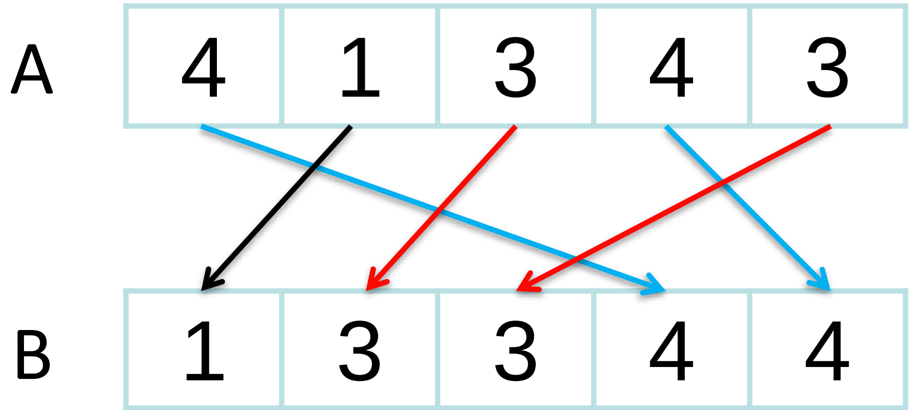
- *Comparison sorting* takes $\Omega(n \lg n)$ time.
- Counting sort is not a *comparison sort.*
- In fact, not a single comparison between elements occurs!

Counting Sort

- Cool! *Why don't we always use counting sort?*
- Because it depends on range k of elements.
- *Could we use counting sort to sort 32 bit integers? Why or why not?*
- Answer: no, k too large (2^{32} = 4,294,967,296)

Stable sorting

Counting sort is a *stable* sort: it preserves the input order among equal elements.



Exercise: What other sorts have this property?

Normally, the property of stability is important only when satellite data are carried around with the element being sorted. Counting-Sort is often used as a subroutine in radix sort.

Radix Sort

Number of keys : not one, but several keys… say *d* keys.

K^0, K^1, ... , K^{d-1}

Most significant

Least significant

Ex) Sorting students by class, math. score, height

K^0 K^1 K^2

Ex) Sorting deck of cards

Two keys – (1) suit : clover < diamond < heart < spade

(2) face value : 2 < 3 < 4 < . . . . < J < Q < K < A

After sorting,

2C, 3C, . . . KC, AC, 2D, 3D, ... < KS < AS

Radix Sort

[1] MSD : Most Significant Digit

Step 1 : Sort by suit

Step 2 : Sort by face value

[2] LSD : Least Significant Digit

Step 1 : Sort by face

Step 2 : Sort by suit

Radix Sort

- Intuitively, you might sort on the most significant digit, then the second msd, etc.
- Problem: lots of intermediate piles of cards 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
```

Example of Radix Sort

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

Radix Sort

- *Can we prove it will work?*
- Sketch of an inductive argument (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 two digits at position i are different, ordering numbers by that digit is correct (lower-order digits irrelevant)
    - If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order

Radix Sort

- *What sort will we use to sort on digits?*
- Counting sort is obvious choice:
  - Sort n numbers on digits that range from $1..k$
  - Time: $\Theta(n + k)$
- Each pass over n numbers with d digits takes time $\Theta(n+k)$, so total time $\Theta(dn+dk)$
  - When d is constant and $k=O(n)$, it takes $\Theta(n)$ time.

Radix Sort

- In general, radix sort based on counting sort is
  - Fast
  - Asymptotically fast (i.e., $\Theta(n)$)
  - Simple to code
  - A good choice

Bucket Sort

- Bucket sort
  - Assumption: input is *n* reals from [0, 1]
  - Basic idea:
    - Create *n* lists (*buckets*) to divide interval [0,1] into subintervals of size 1/*n*
    - Add each input element to appropriate bucket and sort buckets with insertion sort
  - Uniform input distribution → Θ(1) bucket size
    - Therefore the expected total time is Θ(*n*)
  - The idea as discussed in *hash table.*

Worst-Case Running Time

- Lemma: *In the worst case, Bucket Sort takes $O(n^2)$ time.*

- Using Merge Sort, we can get this down to $\Theta(n \lg n)$.

- But Insertion Sort is simpler.

Average-Case Running Time

- Lemma: *Given that the input sequence is drawn uniformly at random from* [0,1)*, the expected size of a bucket is* Θ(1)*.*

- Lemma: *Given that the input sequence is drawn uniformly at random from* [0,1)*, the average-case running time of Bucket Sort is* Θ(*n*)*.*

Summary

- Every *comparison-based sorting* algorithm has to take $\Omega(n \lg n)$ *time*.
- *Merge Sort*, *Heap Sort*, and *Quick Sort* are comparison-based and take $\Theta(n \lg n)$ time. Hence, they *are optimal*.
- Other *sorting algorithms can be faster* by exploiting assumptions made about the input.
- *Counting Sort* and *Radix Sort* take *linear time* for *integers in a bounded range*.
- *Bucket Sort* takes *linear average-case time* for *uniformly distributed* real numbers.