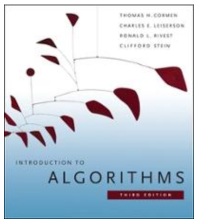
CS146 Data Structures and Algorithms



Chapter 8: Sorting in Linear Time

L8.1

Sorting So Far

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

Sorting So Far

- Merge sort:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort subarrays
    - Linear-time merge step
  - O(n lg n) worst case
  - 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
  - O(n lg n) worst case
  - Sorts in place
  - Fair amount of shuffling memory around

Sorting So Far

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two subarrays, recursively sort
    - All of first subarray < all of second subarray
    - No merge step needed!
  - O(n lg n) average case
  - Fast in practice
  - O(n^2) worst case
    - Naïve implementation: worst case on sorted input
    - Address this with randomized quicksort

Comparison-based Sorting

- **Comparison sort**
  - Only comparison of pairs of elements may be used to gain order information about a sequence.
  - Hence, a lower bound on the number of comparisons will be a lower bound on the complexity of any comparison-based sorting algorithm.
- The best worst-case complexity so far is $\Theta(n \lg n)$ (merge sort and heapsort).
- We prove a lower bound of $\Omega(n \lg n)$ for any comparison sort: merge sort and heapsort are optimal.
- The idea is simple: there are $n!$ outcomes, so we need a tree with $n!$ leaves, and therefore $\lg(n!) = \Theta(?)$

How Fast Can We Sort?

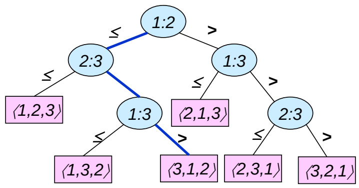
- We will provide a lower bound, then beat it
  - *How do you suppose we'll beat it?*
- First, an observation: all of the sorting algorithms so far are *comparison sorts*
  - The only operation used to gain ordering information about a sequence is the pairwise comparison of two elements
  - Theorem: all comparison sorts are Ω(n lg n)
    - A comparison sort must do O(n) comparisons (*why?*)
    - What about the gap between O(n) and O(n lg n)

Decision Tree

- Binary-tree abstraction for any comparison sort.
- Represents comparisons made by
  - a specific sorting algorithm
  - on inputs of a given size.
- Abstracts away everything else – control and data movement – counting only comparisons.
- Each internal node is annotated by $i{:}j$, which are indices of array elements from their original positions.
- Each leaf is annotated by a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ of orders that the algorithm determines.

Decision Tree

For insertion sort operating on three elements.



Simply unroll all loops for all possible inputs.

Node i:j means compare A[i] to A[j].

Leaves show outputs;

No two paths go to same leaf!

The blue path indicates the sorting decision for input array [6,8,5]

Contains 3! = 6 leaves = possible permutations of the input elements.

Decision Tree (Contd.)

- Execution of sorting algorithm corresponds to tracing a path from root to leaf.
- The tree modelsAt each internal node, a comparison $a_i \le a_j$ is made.
  - If $a_i \le a_j$, follow left subtree, else follow right subtree.
  - View the tree as if the algorithm splits in two at each node, based on information it has determined up to that point.
- When we come to a leaf, ordering $a_{\pi(1)} \le a_{\pi(2)} \le \ldots \le a_{\pi(n)}$ is established.
- A correct sorting algorithm must be able to produce any permutation of its input.
  - Hence, each of the $n!$ permutations must appear at one or more of the leaves of the decision tree.
- all possible execution traces.

A Lower Bound for Worst Case

- Worst case no. of comparisons for a sorting algorithm is
  - Length of the longest path from root to any of the leaves in the decision tree for the algorithm.
    - Which is the height of its decision tree.
- A lower bound on the running time of any comparison sort is given by
  - A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf.

A Lower Bound for Worst Case

Theorem 8.1:
Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof:

- From previous discussion, suffices to determine the height of a decision tree.
- h – height, l – no. of reachable leaves in a decision tree.
- In a decision tree for n elements, $l \ge n!$. **Why?**
- In a binary tree of height h, no. of leaves $l \le 2^h$. **Prove it.**
- Hence, $\mathbf{n! \le l \le 2^h}$.

A Lower Bound for Worst Case

- Decision trees can model comparison sorts. For a given algorithm:
  - One tree for each *n*
  - Tree paths are all possible execution traces
  - *What's the longest path in a decision tree for insertion sort? For merge sort?*
- *What is the asymptotic height of any decision tree for sorting n elements?*
- Answer: $\Omega(n \lg n)$ (now let's prove it…)

A Lower Bound for Worst Case

- So we have… $n! \le l \le 2^h$
  $n! \le 2^h$
- Taking logarithms:
  $\lg (n!) \le h$
- Stirling's approximation tells us:
  $$n! > \left(\frac{n}{e}\right)^n$$
- Thus: $h \ge \lg\left(\frac{n}{e}\right)^n$

A Lower Bound for Worst Case

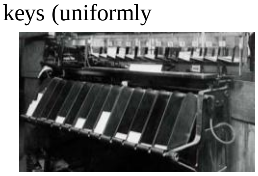
- So we have

$$\begin{aligned} h &\ge \lg\left(\frac{n}{e}\right)^n \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n) \end{aligned}$$

- Thus the minimum height of a decision tree is $\Omega(n \lg n)$

Beating the lower bound

- *How can we do better than $\Omega(n \lg n)$?*
- We can beat the lower bound if we don't base our sort on comparisons:
  - Counting sort for keys in [0..k], k = O(n)
  - Radix sort for keys with a fixed number of "digits"
  - Bucket sort for random keys (uniformly distributed)



Sorting In Linear Time

- Counting sort
  - No comparisons between elements!
  - ***But***…depends on assumption about the numbers being sorted
    - We assume numbers are in the range *1.. k*
  - The algorithm:
    - Input: A[1..*n*], where A[j] ∈ {1, 2, 3, …, *k*}
    - Output: B[1..*n*], sorted (notice: not sorting in place)
    - Also: Array C[1..*k*] for auxiliary storage

Counting Sort

- **Assumption**: we sort integers in {0, 1, 2, …, *k*}.
- **Input:** $A[1..n] \subset \{0, 1, 2, \ldots, k\}^n$.
  Array *A* and values *n* and *k* are given.
- **Output:** *B*[1..*n*] sorted. Assume *B* is already allocated and given as a parameter.
- **Auxiliary Storage:** *C*[0..*k*] counts
- Runs in linear time if ***k = O(n)***.

Counting-Sort (*A, B, k*)

```
CountingSort(A, B, k)
1.  Let C[0…k] ne a new array
2.  for i ← 0 to k
3.      do C[i] ← 0
4.  for j ← 1 to A.length
5.      do C[A[j]] ← C[A[j]] + 1
// C[i] now contains the number of elements
   equal to i.
6. for i ← 1 to k
7.      do C[i] ← C[i] + C[i –1]
// C[i] now contains the number of elements
   less than or equal to i.
8. for j ← A.length downto 1
9.      do B[C[A[ j ]]] ← A[j]
10.        C[A[j]] ← C[A[j]]–1
```

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

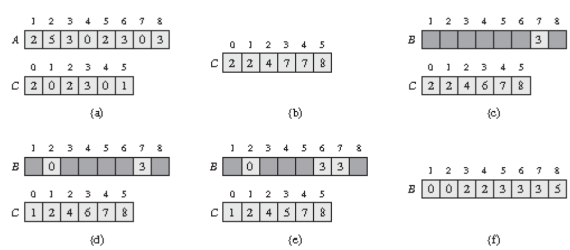
O(k) init counts (lines 2–3)

O(n) counts (lines 4–5)

O(k) prefix sum (lines 6–7)

O(n) reorder (lines 8–10)

Counting-Sort (*A, B, k*)



- (a): Array *A* and the auxiliary array *C* after line 5
- (b): Array after line 7
- (c) ~ (e): The change values of array *B* and auxiliary array *C* after iteration, 1, 2, and 3.
- (f): The final sorted output array *B*.

Algorithm Analysis

- The overall time is $O(n+k)$. When we have $k=O(n)$, the worst case is $O(n)$.
  - for-loop of lines 2-3 takes time $O(k)$
  - for-loop of lines 4-5 takes time $O(n)$
  - for-loop of lines 6-7 takes time $O(k)$
  - for-loop of lines 8-10 takes time $O(n)$
- Stable, but not in place.
- No comparisons made: it uses actual values of the elements to index into an array.

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)
- 16-bit?
  - Probably not.
- 8-bit?
  - Maybe, depending on n.
- **4-bit?**
  - Probably, (unless n is really small).

Pop Quiz Solution – Counting Sort

```
CountingSort(A, B, k)
1.  Let C[0…k] ne a new array
2.  for i ← 0 to k
3.      do C[i] ← 0
4.  for j ← 1 to A.length
5.      do C[A[j]] ← C[A[j]] + 1
// C[i] now contains the number of elements
    equal to i.
6. for i ← 1 to k
7.      do C[i] ← C[i] + C[i –1]
// C[i] now contains the number of elements
    less than or equal to i.
8. for j ← A.length downto 1
9.      do B[C[A[ j ]]] ← A[j]
10.         C[A[j]] ← C[A[j]]–1
```

L7.23

Pop Quiz Solution – Counting Sort

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

	0	1	2	3	4	5	6
C	2	2	2	2	1	0	2

After Line 4, 5

L7.24

Pop Quiz Solution – Counting Sort

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

	0	1	2	3	4	5	6
C	2	2	2	2	1	0	2

After Line 4, 5

	0	1	2	3	4	5	6
C	2	4	6	8	9	9	11

After Line 6, 7

Pop Quiz Solution – Counting Sort

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

	0	1	2	3	4	5	6
C	2	2	2	2	1	0	2

After Line 4, 5

	0	1	2	3	4	5	6
C	2	4	6	8	9	9	11

After Line 6, 7

	1	2	3	4	5	6	7	8	9	10	11
B						2					

Line 8, 9, 10 Loop 1

	0	1	2	3	4	5	6
C	2	4	5	8	9	9	11

Pop Quiz Solution – Counting Sort

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

	0	1	2	3	4	5	6
C	2	4	5	8	9	9	11

	1	2	3	4	5	6	7	8	9	10	11
B						2		3			

Line 8, 9, 10 Loop 2

Pop Quiz Solution – Counting Sort

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

	0	1	2	3	4	5	6
C	2	4	5	7	9	9	11

	1	2	3	4	5	6	7	8	9	10	11
B				1		2		3			

Line 8, 9, 10 Loop 3

Pop Quiz Solution – Counting Sort

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

	0	1	2	3	4	5	6
C	2	3	5	7	9	9	11

	1	2	3	4	5	6	7	8	9	10	11
B				1		2		3			6

Line 8, 9, 10 Loop 4

Pop Quiz Solution – Counting Sort

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

	0	1	2	3	4	5	6
C	2	3	5	7	9	9	10

	1	2	3	4	5	6	7	8	9	10	11
B				1		2		3	4		6

Line 8, 9, 10 Loop 5

Pop Quiz Solution – Counting Sort

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

	0	1	2	3	4	5	6
C	2	3	5	7	8	9	10

	1	2	3	4	5	6	7	8	9	10	11
B				1		2	3	3	4		6

Line 8, 9, 10 Loop 6

Pop Quiz Solution – Counting Sort

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

	0	1	2	3	4	5	6
C	2	3	5	6	8	9	10

	1	2	3	4	5	6	7	8	9	10	11
B			1	1		2	3	3	4		6

Line 8, 9, 10 Loop 7

Pop Quiz Solution – Counting Sort

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

	0	1	2	3	4	5	6
C	2	2	5	6	8	9	10

	1	2	3	4	5	6	7	8	9	10	11
B		0	1	1		2	3	3	4		6

Line 8, 9, 10 Loop 8

Pop Quiz Solution – Counting Sort

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

	0	1	2	3	4	5	6
C	1	2	5	6	8	9	10

	1	2	3	4	5	6	7	8	9	10	11
B		0	1	1	2	2	3	3	4		6

Line 8, 9, 10 Loop 9

Pop Quiz Solution – Counting Sort

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

	0	1	2	3	4	5	6
C	1	2	4	6	8	9	10

	1	2	3	4	5	6	7	8	9	10	11
B	0	0	1	1	2	2	3	3	4		6

Line 8, 9, 10 Loop 10

Pop Quiz Solution – Counting Sort

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

	0	1	2	3	4	5	6
C	0	2	4	6	8	9	10

	1	2	3	4	5	6	7	8	9	10	11
B	0	0	1	1	2	2	3	3	4	6	6

Line 8, 9, 10 Loop 11

Done