Algorithms and Programming I
Spring 2015

Lecture8

Linear Sorting

Sorting by Comparisons

Up to this point, all the sorting algorithms we examined depended on comparing elements of a given set, in order to sort the set. All the algorithms we came up with, were either $O(n \lg n)$ or $\Theta(n \lg n)$ or $O(n^2)$.

One can ask: can we sort a set S, consisting of elements from a totally ordered universe, in time $O(|S|)$?

The answer, as we might expect, is "yes, but…"

First of all, the negative result: sorting by comparisons is (worst case) $\Omega(n \lg n)$.

Linear Sorting

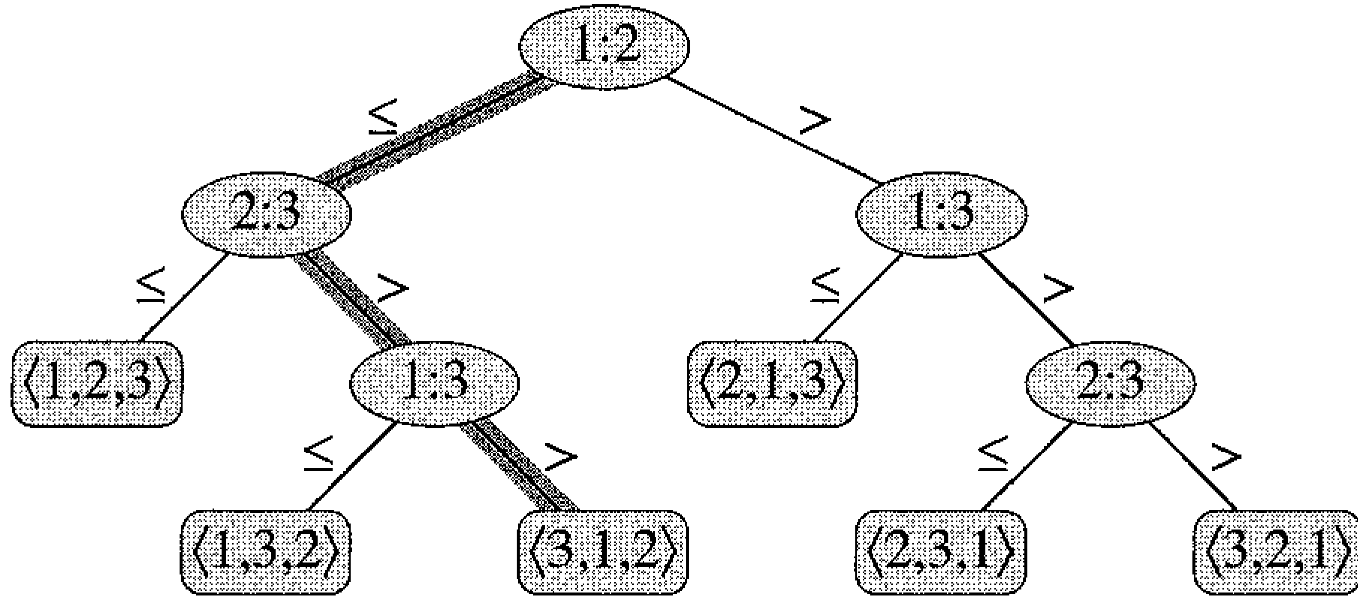

Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i:j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \le a_1 = 6 \le a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, and so the decision tree must have at least 6 leaves.

Linear Sorting

The main ideas are:

1. Every time you try to determine the relative positions of two element you must make a comparison (decision).
2. The input (n elements) can come in any order.
3. There are $n!$ ways in which n different elements can be arranged.
4. A “sort” is equivalent to finding (by a sequence of comparisons between two elements) the permutation of the input that leaves the input set ordered.
5. Each such permutation corresponds to a leaf of the “binary decision tree” generated by the comparisons.
6. The binary decision tree has $n!$ leaves.

Linear Sorting

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

Proof: by the previous discussion, the binary decision tree has at least $n!$ leaves, and height h. Since such a binary tree cannot have more than 2^h leaves, we have $n! \leq 2^h$. Taking the logarithm (base 2) of both sides:

$$h \geq \lg(n!) = \Omega(n \lg n),$$

This means that there is at least ONE path of length h connecting the root to a leaf.

Corollary : HEAPSORT and MERGESORT are asymptotically optimal comparison sorts.

Linear Sorting

Sorting NOT by comparisons

How do we do it?

- We must make some further assumptions.
- For example, we need to assume more than "the set to be sorted is a set of integers".
- More specifically, we assume the integers fall in some range, say $[1..k]$, where $k = O(n)$.
- This is the idea behind **Counting Sort**. How do we use it?

Linear Sorting

(a)

	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

(b)

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

(c)

	1	2	3	4	5	6	7	8
B							3	

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

(d)

	1	2	3	4	5	6	7	8
B		0					3	

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

(e)

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

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

(f)

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

Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

Ch.8 – Linear Sorting

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i - 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] - 1
```

Figure 8.2 illustrates counting sort. After the **for** loop of lines 2–3 initializes the array C to all zeros, the **for** loop of lines 4–5 inspects each input element. If the value of an input element is i, we increment $C[i]$. Thus, after line 5, $C[i]$ holds the number of input elements equal to i for each integer $i = 0, 1, \ldots, k$. Lines 7–8 determine for each $i = 0, 1, \ldots, k$ how many input elements are less than or equal to i by keeping a running sum of the array C.

Ch.8 – Linear Sorting

Counting Sort: Time Complexity

How much time?

- The **for** loop of l. 2-3 takes time $\Theta(k)$.
- The **for** loop of l. 4-5 takes time $\Theta(n)$.
- The **for** loop of l. 7-8 takes time $\Theta(k)$.
- The **for** loop of l. 10-12 takes time $\Theta(n)$.
- The overall time is $\Theta(k + n)$.
- The assumption on k gives that the overall time is $\Theta(n)$.

Ch.8 – Linear Sorting

Radix Sort I

What else can we use?

Assume all your integers (we are sorting sets of integers) have d or fewer digits. Pad the ones with fewer than d digits with leading 0s, for uniformity.

Assume the digits are on cards with 80 or so vertical columns, each column with room for 10 (or more) distinct holes (one for each of $0..9$).

Use columns $1..d$ to store each integer.

Take a deck of such cards (with integers) and sort it.

Ch.8 – Linear Sorting

Radix Sort

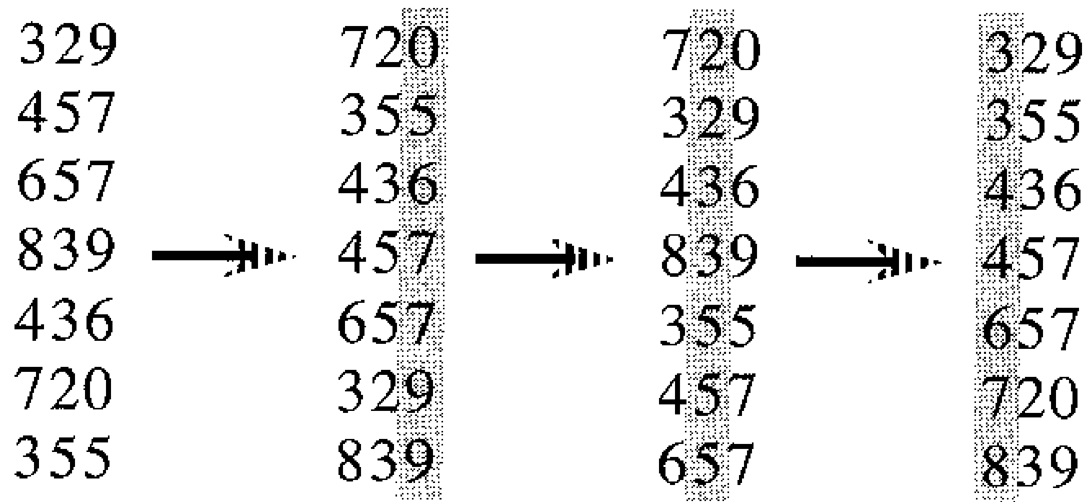

Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

RADIX-SORT(A, d)

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

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

Ch.8 – Linear Sorting

Radix Sort

Lemma 8.3. Given n d-digit numbers in which each digit can take up to k possible values, RADIXSORT correctly sorts these numbers in $\Theta(d(n + k))$ time if the stable sort it uses takes $\Theta(n + k)$ time.

Proof: the correctness follows by induction on the column being sorted. Sort the first column (using a stable sort). Assume the set is sorted on the first i columns (starting from the back); prove it remains sorted when we use a stable sort on the $i+1^{st}$ column (see Ex. 8.3-3).

COUNTINGSORT on each digit will give the result (details?).

Ch.8 – Linear Sorting

Bucket Sort

Assume all the values have equal (independent) probability of appearing as elements of [0, 1).

- Divide the interval [0, 1) into n equal sized subintervals (buckets) and distribute the n input numbers into the buckets.
- Sort the numbers in each bucket (your choice of sort).
- Go through the buckets in order, listing the contents.

Ch.8 – Linear Sorting

Bucket Sort

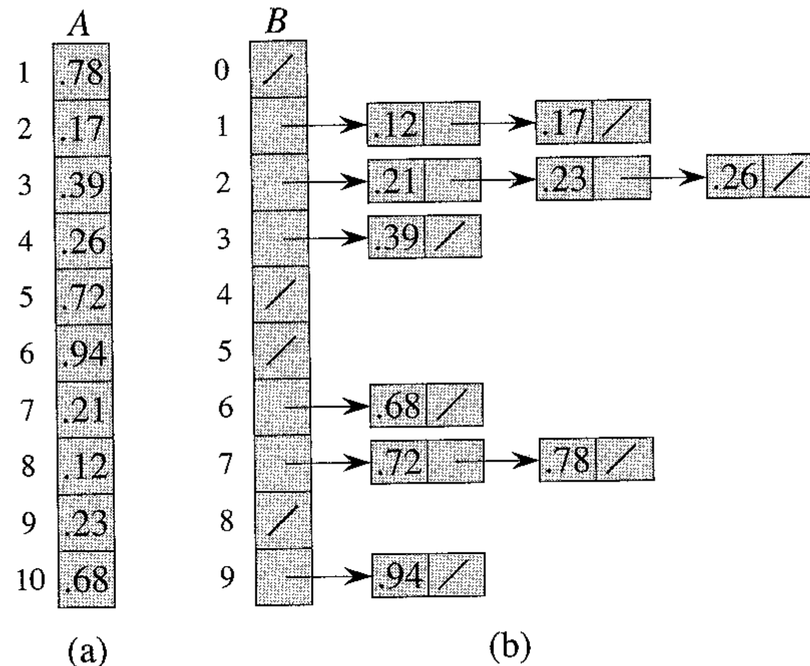

Figure 8.4 The operation of BUCKET-SORT for $n = 10$. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 8 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Ch.8 – Linear Sorting

Bucket Sort

```
BUCKET-SORT(A)
1  let B[0..n-1] be a new array
2  n = A.length
3  for i = 0 to n - 1
4      make B[i] an empty list
5  for i = 1 to n
6      insert A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n - 1
8      sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], ..., B[n-1] together in order
```

We observe that both of the **for** loops in l. 3-4 and 5-6 take time $O(n)$ to execute. We need to analyze the cost of the n calls to INSERTIONSORT on l. 8.