Linear Time Sorting Algorithms

Dr. Navjot Singh

Design and Analysis of Algorithms

Comparison-based Sorting

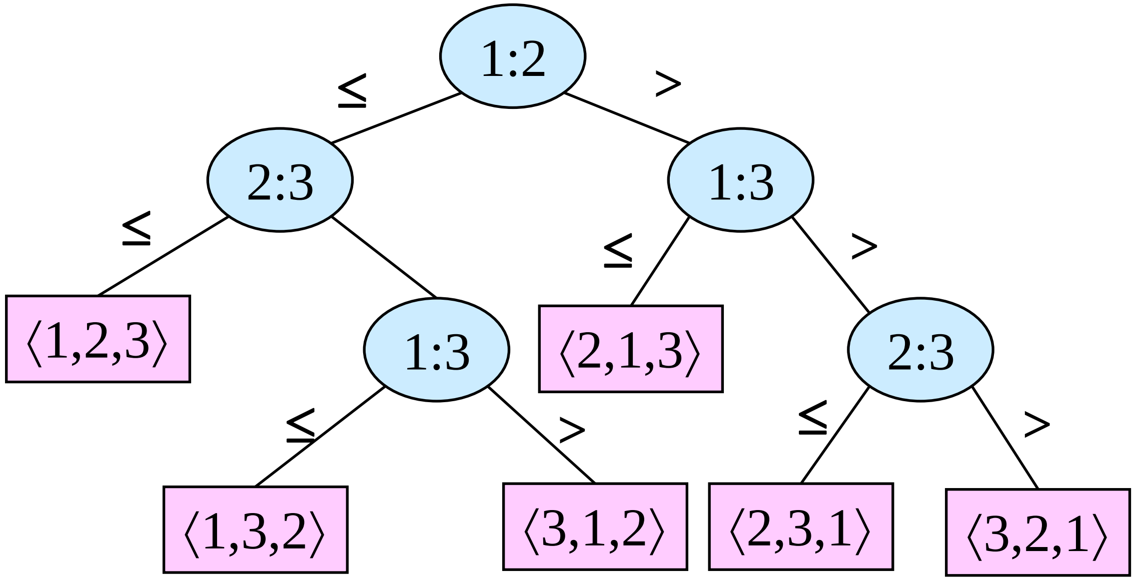
- **Comparison sort**
  - Only comparison of pairs of elements may be used to gain order information about a sequence.
  - Hence, a lower bound on the number of comparisons will be a lower bound on the complexity of any comparison-based sorting algorithm.
- All our sorts have been comparison sorts
- The best worst-case complexity so far is $\Theta(n \lg n)$ (merge sort and heapsort).
- We prove a lower bound of $n \lg n$, (or $\Omega(n \lg n)$) for any comparison sort, implying that merge sort and heapsort are optimal.

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 – Example

For insertion sort operating on three elements.



Contains 3! = 6 leaves.

Decision Tree (Contd.)

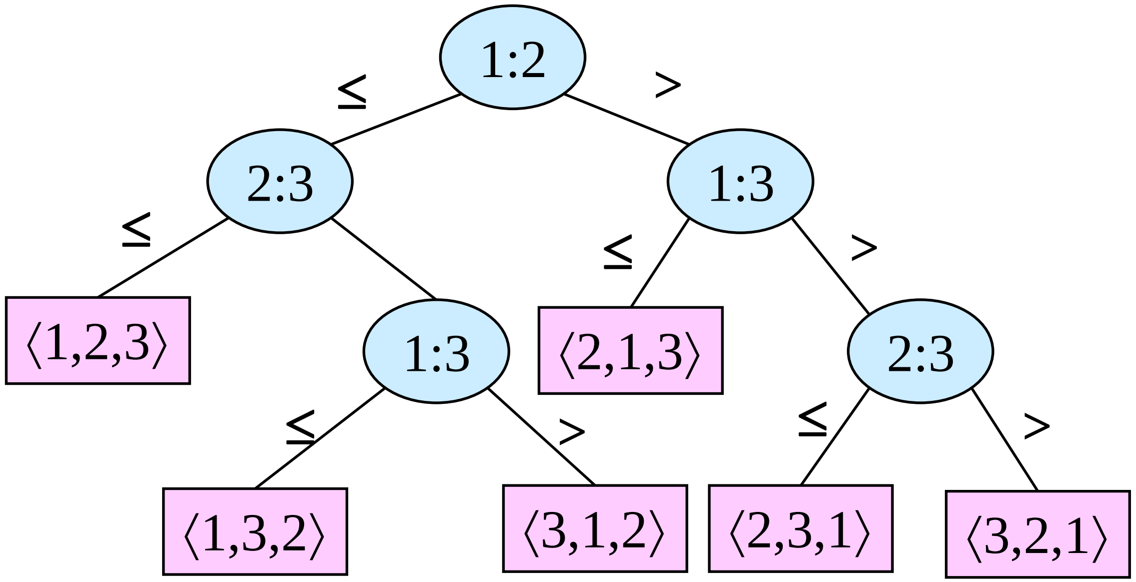
- Execution of sorting algorithm corresponds to tracing a path from root to leaf.
- The tree models all possible execution traces.
- At each internal node, a comparison $a_i \leq a_j$ is made.
  - If $a_i \leq a_j$, follow left subtree, else follow right subtree.
  - View the tree as if the algorithm splits in two at each node, based on information it has determined up to that point.
- When we come to a leaf, ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$ 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.

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.

Optimal sorting for three elements

Any sort of three elements has 5 internal nodes.



There must be a wost-case path of length ≥ 3.

A Lower Bound for Worst Case

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

Proof:

- The number of leaves is at least $n!$ (# outputs)
- The number of internal nodes ≥ $n!–1$
- The height is at least $\lg(n!–1)$
- ***h*** – height, ***l*** – no. of reachable leaves in a decision tree.
- In a decision tree for n elements, $l \geq n!$. **Why?**
- In a binary tree of height h, no. of leaves $l \leq 2^h$. **Prove it.**
- Hence, $\mathbf{n! \leq l \leq 2^h}$.

Proof – Contd.

- $n! \leq l \leq 2^h$ or $2^h \geq n!$
- Taking logarithms, $h \geq \lg(n!)$.
- $n! > (n/e)^n$. (Stirling's approximation)
- Hence, $h \geq \lg(n!)$

$$\geq \lg(n/e)^n$$
$$= n \lg n - n \lg e$$
$$= \Omega(n \lg n)$$

Non-comparison Sorts: Counting Sort

- Depends on a **key *assumption***: numbers to be sorted are integers in $\{0, 1, 2, \ldots, k\}$.
- **Input:** $A[1..n]$, where $A[j] \in \{0, 1, 2, \ldots, k\}$ for $j = 1, 2, \ldots, n$. Array A and values n and k are given as parameters.
- **Output:** $B[1..n]$ sorted. B is assumed to be already allocated and is given as a parameter.
- **Auxiliary Storage:** $C[0..k]$
- Runs in linear time if $k = O(n)$.

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

CountingSort(*A*, *B*, *k*)

1. **for** $i \leftarrow 0$ to k
2. **do** $C[i] \leftarrow 0$
3. **for** $j \leftarrow 1$ to $length[A]$
4. **do** $C[A[j]] \leftarrow C[A[j]] + 1$
5. **for** $i \leftarrow 1$ to k
6. **do** $C[i] \leftarrow C[i] + C[i-1]$
7. **for** $j \leftarrow length[A]$ **downto** 1
8. **do** $B[C[A[j]]] \leftarrow A[j]$
9. $C[A[j]] \leftarrow C[A[j]]-1$

Lines 1–2: $O(k)$

Lines 3–4: $O(n)$

Lines 5–6: $O(k)$

Lines 7–9: $O(n)$

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

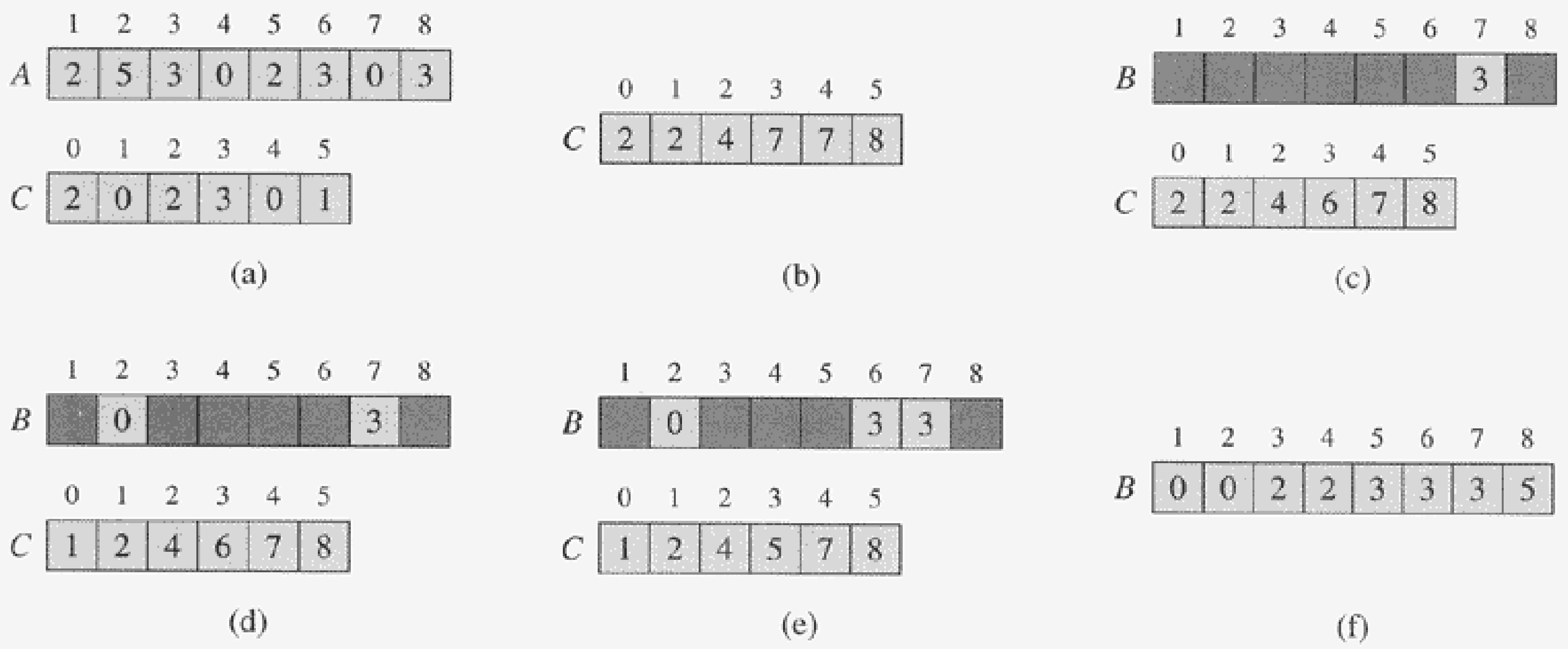


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 9–11, respectively. Only the lightly shaded elements of array B have been filled in. **(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 1-2 takes time $O(k)$
  - for-loop of lines 3-4 takes time $O(n)$
  - for-loop of lines 5-6 takes time $O(k)$
  - for-loop of lines 7-9 takes time $O(n)$
- Stable, but not in place.
- No comparisons made: it uses actual values of the elements to index into an array.

What values of *k* are practical?

- Good for sorting 32-bit values? No. **Why?**
- 16-bit? Probably not.
- 8-bit? Maybe, depending on n.
- 4-bit? Probably, (unless n is really small).

- Counting sort will be used in radix sort.

Radix Sort

- It was used by the card-sorting machines.
- Card sorters worked on one column at a time.
- It is the algorithm for using the machine that extends the technique to multi-column sorting.
- The human operator was part of the algorithm!
- ***Key idea:*** sort on the “least significant digit” first and on the remaining digits in sequential order. The sorting method used to sort each digit must be “stable”.
  - If we start with the “most significant digit”, we’ll need extra storage.

An Example

Input		After sorting on LSD		After sorting on middle digit		After sorting on MSD
392		631		928		356
356		392		631		392
446		532		532		446
928	$\Rightarrow$	495	$\Rightarrow$	446	$\Rightarrow$	495
631		356		356		532
532		446		392		631
495		928		495		928
		$\uparrow$		$\uparrow$		$\uparrow$

Radix-Sort(*A*, *d*)

RadixSort(*A*, *d*)

1. for $i \leftarrow 1$ to d
2. do *use a stable sort to sort array A on digit i*

Correctness of Radix Sort

By induction on the number of digits sorted.

Assume that radix sort works for $d - 1$ digits.

Show that it works for d digits.

Radix sort of d digits $\equiv$ radix sort of the low-order $d - 1$ digits followed by a sort on digit d .

Correctness of Radix Sort

By induction hypothesis, the sort of the low-order $d - 1$ digits works, so just before the sort on digit d , the elements are in order according to their low-order $d - 1$ digits. The sort on digit d will order the elements by their d^{th} digit.

Consider two elements, a and b, with d^{th} digits a_d and b_d:

- If $a_d < b_d$, the sort will place a before b, since $a < b$ regardless of the low-order digits.
- If $a_d > b_d$, the sort will place a after b, since $a > b$ regardless of the low-order digits.
- If $a_d = b_d$, the sort will leave a and b in the same order, since the sort is stable. But that order is already correct, since the correct order of is determined by the low-order digits when their d^{th} digits are equal.

Algorithm Analysis

- Each pass over n d-digit numbers then takes time $\Theta(n+k)$. (Assuming counting sort is used for each pass.)
- There are d passes, so the total time for radix sort is $\Theta(d\,(n+k))$.
- When d is a constant and $k = O(n)$, radix sort runs in linear time.
- Radix sort, if uses counting sort as the intermediate stable sort, does not sort in place.
  - If primary memory storage is an issue, quicksort or other sorting methods may be preferable.

Bucket Sort

- Assumes input is generated by a random process that distributes the elements uniformly over [0, 1).
- **Idea:**
  - Divide [0, 1) into *n* equal-sized buckets.
  - Distribute the *n* input values into the buckets.
  - Sort each bucket.
  - Then go through the buckets in order, listing elements in each one.

An Example

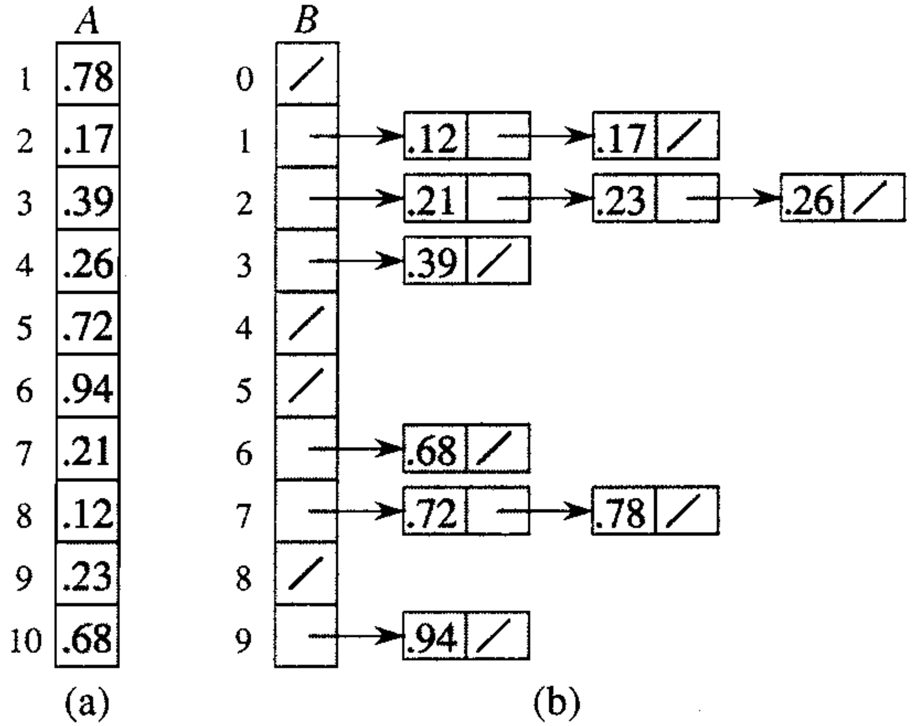


Figure 9.4 The operation of BUCKET-SORT. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Bucket-Sort (*A*)

Input: $A[1..n]$, where $0 \le A[i] < 1$ for all i.
Auxiliary array: $B[0..n-1]$ of linked lists, each list initially empty.

BucketSort(*A*)

1. $n \leftarrow length[A]$
2. **for** $i \leftarrow 1$ to n
3. **do** insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i \leftarrow 0$ **to** $n - 1$
5. **do** sort list $B[i]$ with insertion sort
6. concatenate the lists $B[i]$s together in order
7. **return** the concatenated lists

Correctness of BucketSort

- Consider $A[i]$, $A[j]$. Assume w.o.l.o.g, $A[i] \leq A[j]$.
- Then, $\lfloor n \times A[i] \rfloor \leq \lfloor n \times A[j] \rfloor$.
- So, $A[i]$ is placed into the same bucket as $A[j]$ or into a bucket with a lower index.
  - If same bucket, insertion sort fixes up.
  - If earlier bucket, concatenation of lists fixes up.

Analysis

- Relies on no bucket getting too many values.
- All lines except insertion sorting in line 5 take $O(n)$ altogether.
- Intuitively, if each bucket gets a constant number of elements, it takes $O(1)$ time to sort each bucket $\Rightarrow$ $O(n)$ sort time for all buckets.
- We “expect” each bucket to have few elements, since the average is 1 element per bucket.
- But we need to do a careful analysis.

Analysis – Contd.

- RV n_i = no. of elements placed in bucket $B[i]$.
- Insertion sort runs in quadratic time. Hence, time for bucket sort is:

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

Taking expectations of both sides and using linearity of expectation, we have

$$E[T(n)] = E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right]$$

$$= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \qquad \text{(by linearity of expectation)}$$

$$= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \qquad (E[aX] = aE[X])$$

Analysis – Contd.

- **Claim:** $E[n_i^2] = 2 - 1/n$.
- **Proof:**
- Define indicator random variables.
  - $X_{ij} = I\{A[j] \text{ falls in bucket } i\}$
  - $\Pr\{A[j] \text{ falls in bucket } i\} = 1/n$.
  - $n_i = \sum_{j=1}^{n} X_{ij}$

Analysis – Contd.

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right]$$

$$= E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right]$$

$$= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ j\ne k}} X_{ij}X_{ik}\right]$$

$$= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ j\ne k}} E[X_{ij}X_{ik}] \text{ , by linearity of expectation.}$$

Analysis – Contd.

$$E[X_{ij}^2] = 0^2 \cdot \Pr\{A[j] \text{ doesn't fall in bucket } i\} +$$

$$1^2 \cdot \Pr\{A[j] \text{ falls in bucket } i\}$$

$$= 0 \cdot \left(1 - \frac{1}{n}\right) + 1 \cdot \frac{1}{n}$$

$$= \frac{1}{n}$$

$E[X_{ij}X_{ik}]$ for $j \neq k$:

Since $j \neq k$, X_{ij} and X_{ik} are independent random variables.

$$\Rightarrow E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}]$$

$$= \frac{1}{n} \cdot \frac{1}{n}$$

$$= \frac{1}{n^2}$$

Analysis – Contd.

$$E[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \le j \le n} \sum_{\substack{1 \le k \le n \\ k \ne j}} \frac{1}{n^2}$$

$$= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2}$$

$$= 1 + \frac{n-1}{n}$$

$$= 2 - \frac{1}{n}.$$

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n)$$

$$= \Theta(n) + O(n)$$

$$= \Theta(n)$$

Acknowledgements

- Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., Introduction to algorithms. MIT press, 2009
- Dr. David Kauchak, Pomona College
- Prof. David Plaisted, The University of North Carolina at Chapel Hill