Introduction to Algorithms

Chapter 8 : Sorting in Linear Time

Xiang-Yang Li and Haisheng Tan

School of Computer Science and Technology
University of Science and Technology of China (USTC)

Fall Semester 2024

Outline of Topics

- Lower Bounds for Sorting
- Counting Sort
- Radix Sort
- Bucket Sort

Lower Bounds for Sorting

comparison sorts:

We have introduced several algorithms that can sort n numbers in $\Omega(n \lg n)$ time, which share an interesting property: the sorted order they determine is based only on comparisons between the input elements.

- Section 8.1, any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort n elements.
- Sections 8.2, 8.3, and 8.4 examine three sorting algorithms —counting sort, radix sort, and bucket sort—that run in linear time.

Lower Bounds for Sorting

In a comparison sort, without loss of generality, we assume that all of the input elements are distinct, so all comparisons of two elements a_i and a_j will have the form $a_i < a_j$ or $a_i > a_j$.

We can view comparison sorts abstractly in terms of decision trees. A decision tree is a full binary tree that represents the comparisons between elements.

Decision Tree

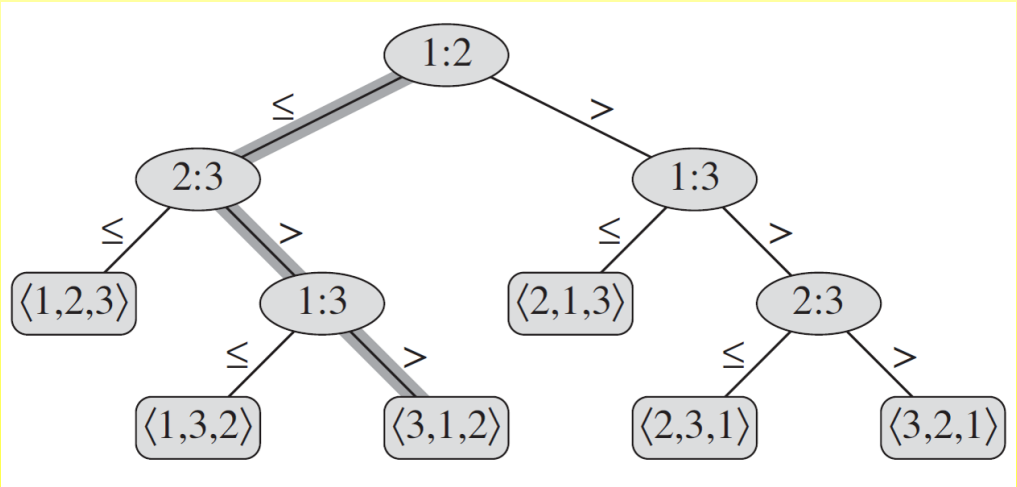


Figure: The decision tree for insertion sort operating on three elements. The shaded path indicates the decisions made when sorting the input sequence $< a_1 = 6;\ a_2 = 8;\ a_3 = 5 >$.

Lower Bounds for Sorting

The length of the longest simple path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs.

A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm.

Lower Bounds for Sorting

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

- Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements;
- Each of the n! permutations of the input appears as some leaf, so $n! \leq l$
- A binary tree of height h has no more than 2^h leaves, so

$$n! \leq l \leq 2^h \tag{1}$$

$$h \geq \log(n!) = \Omega(n \log n) \tag{2}$$

Counting Sort

Assumption:

Each of the n input elements is an integer in the range 0 to k, for some integer k.

Basic idea:

For each input element x, count the number of elements $\leq$ x.

In the counting sort, the input is an array A[1...n] and two other arrays are required: the array B[1...n] holds the sorted output, and the array C[1...n] provides temporary working storage

Counting Sort - Analysis

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

Counting Sort - Example

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

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

Counting Sort - Example

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

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

Counting Sort - Example

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

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

Counting Sort - Example

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

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

Counting Sort - Example

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

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

Counting Sort - Example

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

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

Counting Sort - Example

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

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

Counting Sort - Example

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

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

Counting Sort - Example

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

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

Counting Sort - Example

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

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

Counting Sort - Analysis

COUNTING-SORT(A, B, k)

1: for i = 0 to k do
2: C[i] = 0 //Θ(k)
3: for j = 1 to A.length do
4: C[A[j]] = C[A[j]] + 1 //Θ(n)
5: //C[i] now contains the number of elements equal to i.
6: for i = 1 to k do
7: C[i]=C[i]+C[i-1] //Θ(k)
8: //C[i] now contains the number of elements less than or equal to i.
9: for j = A.length to 1 do
10: B[C[A[j]]] = A[j]
11: C[A[j]] = C[A[j]] − 1 //Θ(n)
12: Overall Time: Θ(n + k); Stable

Radix Sort

Basic idea:

The algorithm used by the card-sorting machines. It sorts n cards on a d-digit number;

Radix sort sorts on the least significant digit first and are then combined into a single deck, then the entire deck is sorted again on the second-least significant digit and recombined in a like manner;

The process continues until the cards have been sorted on all d digits.

Radix Sort - Example

3	2	9
4	5	7
6	5	7
8	3	9
4	3	6
7	2	0
3	5	5

Radix Sort - Example

7	2	0
3	5	5
4	3	6
4	5	7
6	5	7
3	2	9
8	3	9

Radix Sort - Example

7	2	0
3	2	9
4	3	6
8	3	9
3	5	5
4	5	7
6	5	7

Radix Sort - Example

3	2	9
3	5	5
4	3	6
4	5	7
6	5	7
7	2	0
8	3	9

Radix Sort - Analysis

Assumption:

Each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

RADIX-SORT(A, d)
1: for i = 1 to d do
2: use a stable sort to sort array A on digit i

Radix Sort - Analysis

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 numbers in $\Theta(d(n+k))$ time.

Proof:

- Each pass over n d-digit numbers takes time $\Theta(n+k)$
- There are d passes, so the total time for radix sort is $\Theta(d(n+k))$

Radix Sort - Analysis

Lemma 8.4:

Given n b-bit numbers and any positive integer $r \leq b$, RADIX-SORT correctly sorts these numbers in $\Theta((b/r)(n+2^r))$ time.

Proof:

- For a value $r \leq b$, each key was viewed as having $d = \lceil b/r \rceil$ digits of r bits each;
- Each digit is an integer in the range 0 to $2^r - 1$, so that we can use counting sort with $k = 2^r - 1$;
- Each pass of counting sort takes time $\Theta(n+k) = \Theta(n+2^r)$, and there are d passes.
- So the total running time is $\Theta((b/r)(n+2^r))$.

Radix Sort - Analysis

For given values of n and b, how to choose the value of r, with $r \leq b$, that minimizes the expression $(b/r)(n+2^r)$.

- If $b < \lfloor \log n \rfloor$, choosing $r = b$ yields a running time of $\Theta((b/b)(n+2^b)) = \Theta(n)$.
- If $b \geq \lfloor \log n \rfloor$, then choosing $r = \lfloor \log n \rfloor$, the running time is $\Theta(bn/\log n)$.
  Increasing r above $\lfloor \log n \rfloor$ yields a running time of $\Omega(bn/\log n)$.
  If decreasing r below $\lfloor \log n \rfloor$, then the b/r term increases and the $(n+2^r)$ term remains at $\Theta(n)$.

Selection of Sorting Algorithms

Is radix sort preferable to a comparison-based sorting algorithm, such as quick-sort?

- If $b = O(\log n)$ and $r \approx \log n$, then radix sort's running time is $\Theta(n)$, which is better than quicksort's average-case running time of $\Theta(n \log n)$.
- Although radix sort may make fewer passes than quicksort over the n keys, each pass of radix sort may take longer time.

Selection of Sorting Algorithms

Which sorting algorithm is preferred depends on the characteristics of the implementations, the underlying machine and the input data.

For example, Radix sort that uses counting sort does not sort in place, while many comparison sorts do. Thus, when primary memory storage is at a premium, an in-place algorithm such as quicksort may be preferable.

Bucket Sort

Assumption:

The input is drawn from a uniform distribution over the interval $[0, 1)$.

Basic idea:

Bucket sort divides the interval $[0, 1)$ into n equal-sized subintervals, or buckets, and then distributes the n input numbers into the buckets.

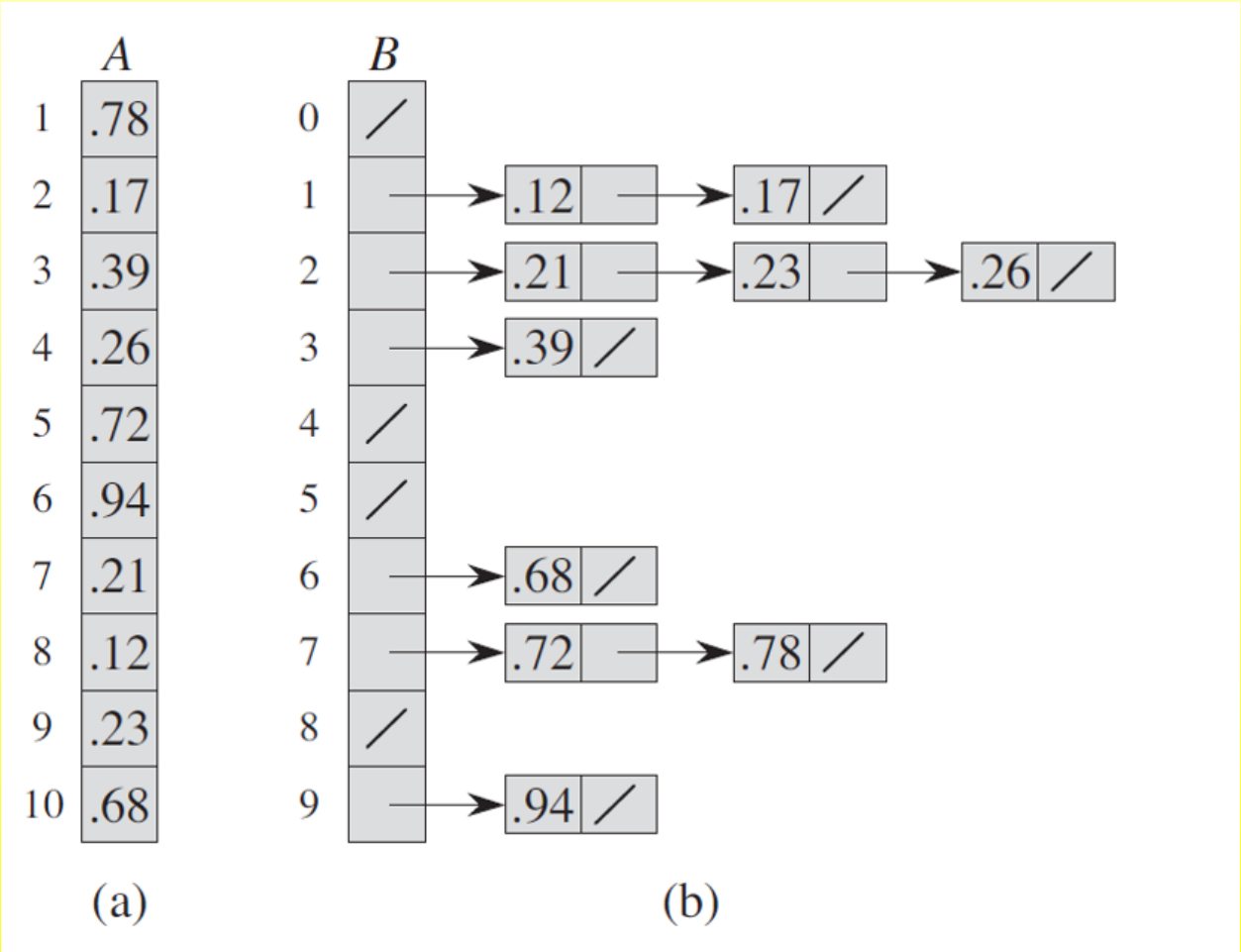
Finally, sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

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 do
4: make B[i] an empty list
5: for i = 1 to n do
6: insert A[i] into list B[⌊n∗A[i]⌋]
7: for i = 0 to n−1 do
8: sort list B[i] with insertion sort
9: concatenate the lists B[0],B[1],...,B[n−1] together in order

Bucket Sort - Example



Bucket Sort - Analysis

Running time: let n_i be the random variable denoting the number of elements placed in bucket B[i], the running time of bucket sort is

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

Taking expectations of both sides

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

Bucket Sort - Analysis

We claim that $E[n_i^2] = 2 - 1/n$.
Define indicator random variables. For $i = 0, 1, \ldots, n-1$ and $j = 1, 2, \ldots, n$.

$$X_{ij} = I\{A[j] \text{ falls in bucket } i\}$$

Thus, $n_i = \sum_{j=1}^{n} X_{ij}$

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] = E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right]$$
$$= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{1\le j\le n}\sum_{1\le k\le n, j\neq k} E[X_{ij}X_{ik}]$$

Bucket Sort - Analysis

Indicator random variable X_{ij} is 1 with probability $1/n$ and 0 otherwise, and therefore

$$E[X_{ij}^2] = 1^2 \times \frac{1}{n} + 0^2 \times (1 - \frac{1}{n}) = \frac{1}{n}$$

when $k \neq j$, the variables X_{ij} and X_{ik} are independent, and hence

$$E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}] = \frac{1}{n^2}$$

Bucket Sort - Analysis

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

Using this expected value, we conclude that the expected time for bucket sort is $\Theta(n) + nO(2 - \frac{1}{n}) = \Theta(n)$