Chapter 8
Sorting in Linear Time

Chi-Yeh Chen
陳奇業
成功大學資訊工程學系

Overview

- Several algorithms that can sort n numbers in $O(n \lg n)$ time.
- Merge sort and heapsort achieve this upper bound in the worst case; quicksort achieves it on average.
- Each of these algorithms is run in $\Omega(n \lg n)$.

Overview

- These algorithms share an interesting property: the sorted order they determine is based only on comparisons between the input elements.

- We call such sorting algorithms comparison sorts.

Lower bounds for sorting (comparison sorts)

- Lower bounds
  - $\Omega(n)$ to examine all the input.
  - All sorts seen so far are $\Omega(n \lg n)$.
  - We' ll show that $\Omega(n \lg n)$ is a lower bound for comparison sorts.

Lower bounds for sorting (comparison sorts)

- In a comparison sort, we use only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$.

- That is, given two elements a_i and a_j, we perform one of the tests $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$ to determine their relative order.

Lower bounds for sorting (comparison sorts)

- Without loss of generality, we assume all the input elements are distinct. (comparisons of the form $a_i = a_j$ are useless)

- We also note that the comparisons $a_i < a_j$, $a_i \leq a_j$, $a_i \geq a_j$, and $a_i > a_j$ are all equivalent in that they yield identical information about the relative order of a_i and a_j.

Lower bounds for sorting (comparison sorts)

- We therefore assume that all comparisons have the form $a_i \leq a_j$.

The decision-tree model

- 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 that are performed by a particular sorting algorithm operating on an input of given size. (full binary tree: each node is either a leaf or has degree exactly 2)
- Control, data movement, and all other aspects of the algorithm are ignored.

- Decision tree
  - Abstraction of 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.
  - We’ re counting only comparisons.

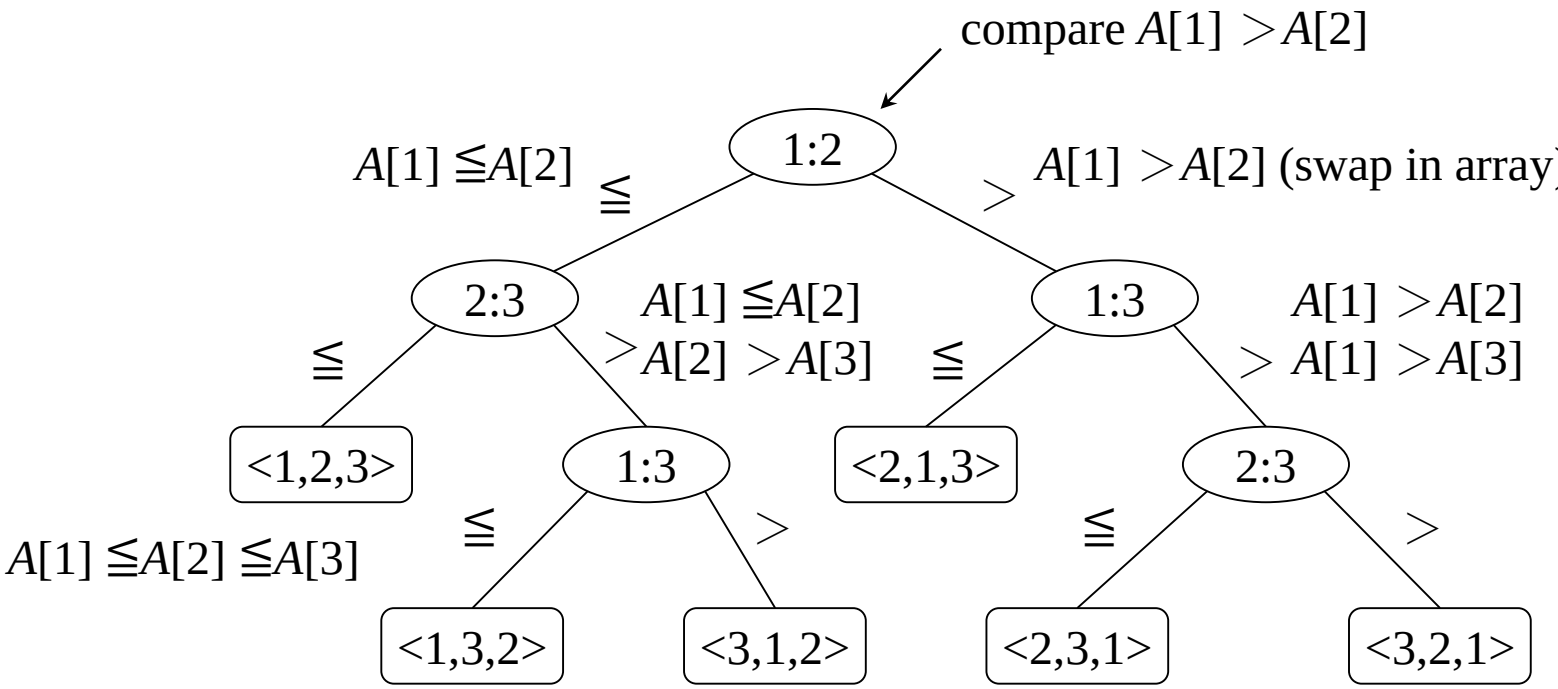
- In a decision tree, an internal node annotated by $i:j$ indicates a comparison between a_i and a_j.

- We also annotate each leaf by a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$.

Insertion-sort(A)

```
1 for j ← 1 to length[A] do
2     key ← A[j]
3     *Insert A[j] into the sorted sequence A[1, . . . , j − 1]
4     i ← j − 1
5     while i > 0 and A[i] > key do
6         A[i + 1] ← A[i]
7         i ← i − 1
8     A[i + 1] ← key
```

For insertion sort on 3 elements:



[Each internal node is labeled by indices of array elements from their original positions. Each leaf is labeled by the permutation of orders that the algorithm determines.]

- The execution of the sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf.
- Each internal node indicates a comparison $a_i \leq a_j$.
- The left subtree then dictates subsequent comparisons once we know that $a_i \leq a_j$, and the right subtree dictates subsequent comparisons knowing that $a_i > a_j$.

- When we come to a leaf, the sorting algorithm has established the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.

- How many leaves on the decision tree? There are $\geq n!$ leaves, because every permutation appears at least once.

- For any comparison sort,
  - one tree for each n.
  - View the tree as if the algorithm splits in two at each node, based on the information it has determined up to that point.
  - The tree models all possible execution traces.

- We shall consider only decision trees in which each permutation appears as a reachable leaf.

A lower bound for the worst case

- 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.

- The worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree.

What is the length of the longest path from root to leaf ?

- Depends on the algorithm
- Insertion sort: $\Theta(n^2)$
- Merge sort: $\Theta(n \lg n)$

Lemma

Any binary tree of height h has $\leq 2^h$ leaves.

In other words:

- $l = \#\text{ of leaves}$,
- $h = \text{height}$,
- Then $l \leq 2^h$.

Proof: By induction on h

Basis: $h = 0$. Tree is just one node, which is a leaf. $2^h = 1$.

Inductive step: Assume true for height$= h - 1$.

Extend tree of height $h - 1$ by making as many new leaves as possible. Each leaf becomes parent to two new leaves.

$$\begin{aligned} \#\text{ of leaves for height } h &= 2 \cdot (\#\text{of leaves for height } h-1) \\ &\leq 2 \cdot 2^{h-1} \qquad \text{(Inductive hypothesis)} \\ &\leq 2^h \end{aligned}$$

Theorem. Any decision tree that sorts n elements has height $\Omega(n \lg n)$

Proof

- $l \geq n!$ because each of the $n!$ permutations of the input appears as some leaf.
- By lemma, $n! \leq l \leq 2^h$ or $2^h \geq n!$
- Take logs: $h \geq \lg(n!)$
- Use Stirling's approximation: $n! > (n/e)^n$ (by equation (3.16))

$$
\begin{aligned}
h &\geq \lg(n/e)^n \\
&= n \lg(n/e) \\
&= n \lg n - n \lg e \\
&= \Omega(n \lg n)
\end{aligned}
$$

(e = Euler's number ≈ 2.71828)

■

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

Corollary

Heapsort and merge sort are asymptotically optimal comparison sort.

- The smallest possible depth of a leaf in a decision tree for a comparison sort is $n - 1$
  - Ex. an already sorted array for insertion sort

There is no comparison sort whose running time is linear for at least half of the $n!$ Inputs of length n.

Proof. If the sort runs in linear time for m input permutations, then the height h of the portion of the decision tree consisting of the m corresponding leaves and their ancestors is linear.

Use the same argument as in the proof of Theorem 8.1 to show that this is impossible for $m = n!/2$.

We have $2^h \geq m$, which gives us $h \geq \lg m$. For all the possible m' s given here, $\lg m = \Omega(n \lg n)$, hence $h = \Omega(n \lg n)$.

In particular,

$$\lg \frac{n!}{2} = \lg n! - 1 \geq n \lg n - n \lg e - 1$$

Moreover,

$$\lg \frac{n!}{n} = \lg n! - \lg n \geq n \lg n - n \lg e - \lg n$$

$$\lg \frac{n!}{2^n} = \lg n! - n \geq n \lg n - n \lg e - n$$

Sorting in linear time

Non-comparison sorts.

Counting sort

Depends on a key assumption: numbers to be sorted are integers in $\{0, 1, \ldots, k\}$.

- **Input**: $A[1 \ldots n]$, where $A[j] \in \{0, 1, \ldots, k\}$ for $j = 1, 2, \ldots, n$. Array A and values n and k are given as parameters.

- **Output**: $B[1 \ldots n]$, sorted. B is assumed to be already allocated and is given as a parameter.
- **Auxiliary storage**: $C[0 \ldots k]$

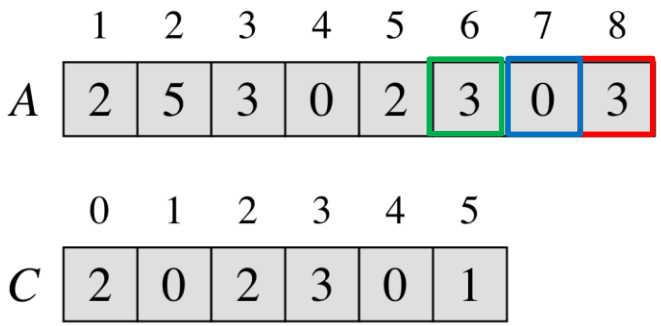
```
COUNTING-SORT(A, B, n, k)
1 for i ← 0 to k do
2     C[i] ← 0
3 for j ← 1 to n do
4     C[A[j]] ← C[A[j]] + 1
5 for i ← 1 to k do
6     C[i] ← C[i] + C[i − 1]
7 for j ← n downto 1 do
8     B[C[A[j]]] ← A[j]
9     C[A[j]] ← C[A[j]] − 1
```

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

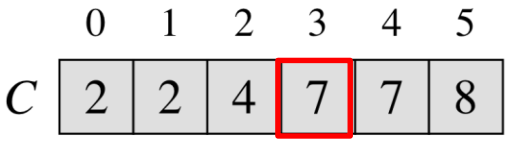
$C[i]$ now contains the number of elements less than or equal to i

- Do an example for $A = 2_1, 5_1, 3_1, 0_1, 2_2, 3_2, 0_2, 3_3$
- Counting sort is stable (keys with same value appear in same order in output as they did in input) because of how the last loop works.

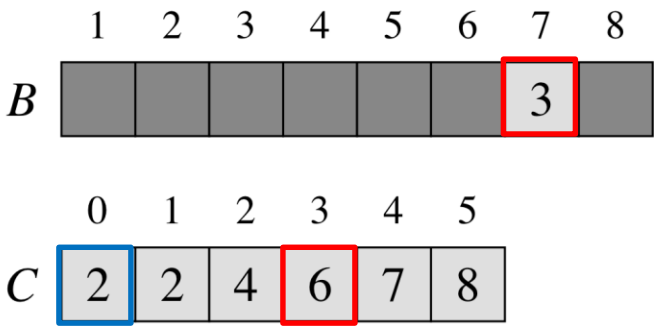
The operation of Counting-sort on an input array $A[1 \ldots 8]$



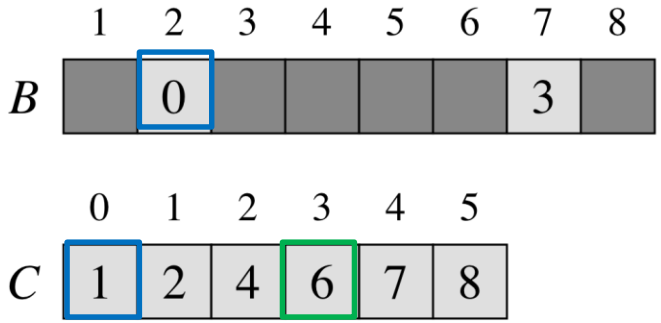
(a)



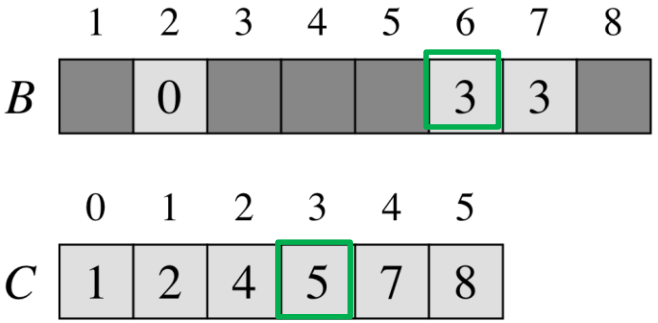
(b)



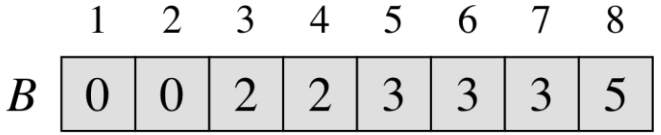
(c)



(d)



(e)



(f)

Analysis: $\Theta(n+k)$, which is $\Theta(n)$ if $k = \Theta(n)$.

How big a k is practical?

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

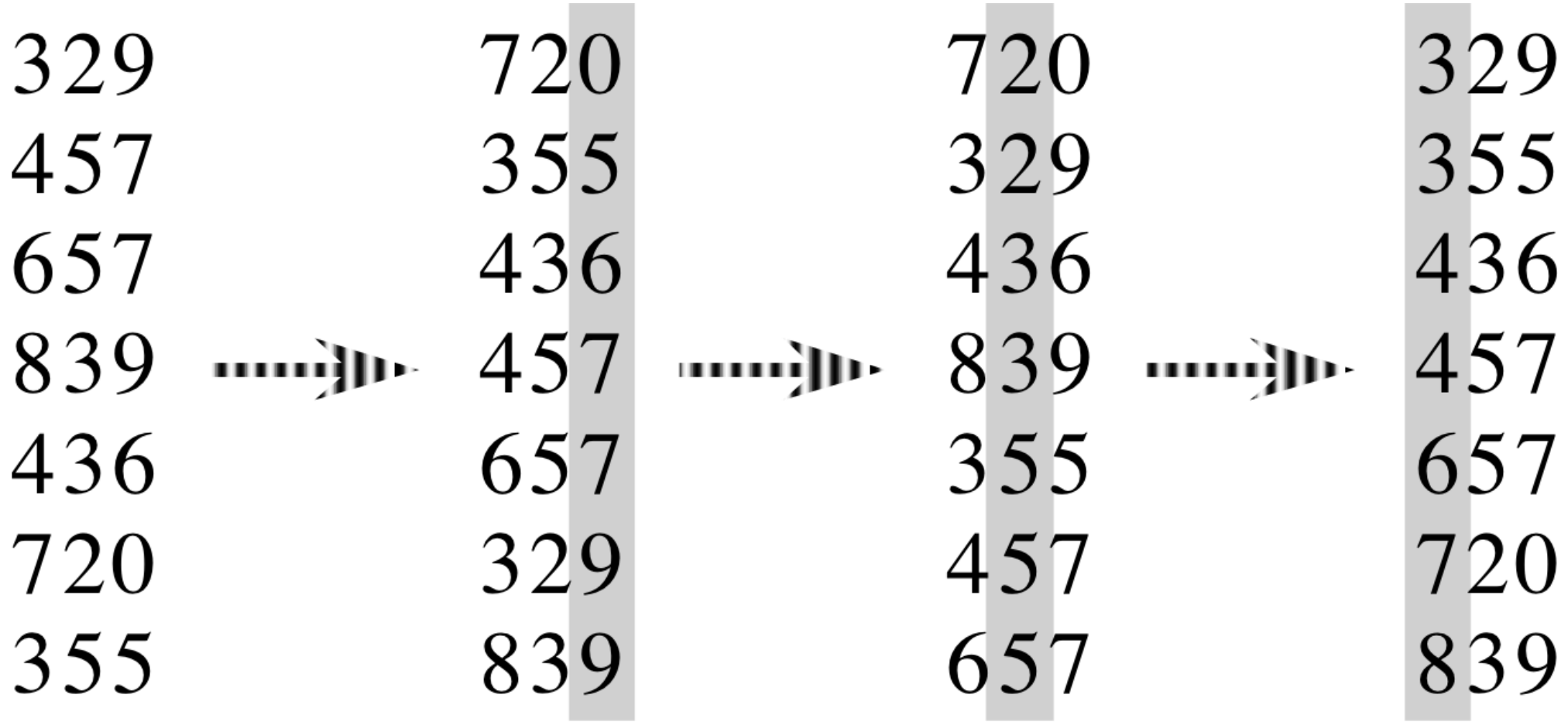
Counting sort will be used in radix sort. (since it is stable)

Radix sort

- **Key idea**: Sort least significant digits first.

To sort d digits:

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


329
457
657
839
436
720
355
720
355
436
457
329
720
329
436
839
355
457
329
355
436
457
657
720
839

Correctness:

- Induction on number of passes (i in pseudocode).
- Assume digits $1, 2, \ldots, i-1$ are sorted.
- Show that a stable sort on digit i leaves digits $1, \ldots, i$ sorted:
  - If 2 digits in position i are different, ordering by position i is correct, and positions $1, 2, \ldots, i-1$ are irrelevant.
  - If 2 digits in position i are equal, numbers are already in the right order (by inductive hypothesis). The stable sort on digit i leaves them in the right order.

This argument shows why it's so important to use a stable sort for intermediate sort.

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 $\Theta\big(d(n+k)\big)$ time if the stable sort it uses takes $\Theta(n+k)$ time.

Analysis: Assume that we use counting sort as the intermediate sort.

- $\Theta(n+k)$ per pass (digits in range $0, \ldots, k$)
- d passes
- $\Theta\big(d(n+k)\big)$ total
- If $k = O(n)$, time $= \Theta(dn)$.

Lemma 8.4

Given n b-bit numbers and any positive integer $r \leq b$,

RADIX-SORT correctly sorts these numbers in $\Theta\left(\frac{b}{r}(n +$

How to break each key into digits?

- n words
- b bits/word
- Break into r-bit digits. Have $d = \lceil b/r \rceil$
- Each digit is an integer in the range 0 to 2^r-1.
- Use counting sort, $k = 2^r$
  Example: 32-bit words, 8-bit digits. $b = 32, r = 8, d = \lceil 32/8 \rceil = 4, k = 2^8 = 256$
- Time $= \Theta\left(\frac{b}{r}(n + 2^r)\right)$

- How to choose r? Balance b/r and $n + 2^r$.
- If $b < \lfloor \lg n \rfloor$, then for any value of $r \leq b$, we have that $(n + 2^r) = \Theta(n)$. Thus, choosing $r = b$ yields a running time of $(b/b)(n + 2^b) = \Theta(n)$.
- If $b \geq \lfloor \lg n \rfloor$, then choosing $r = \lfloor \lg n \rfloor$ gives the best time to within a constant factor.

- Choosing $r = \lg n$ gives us $\Theta\left(\frac{b}{\lg n}(n+n)\right) = \Theta\left(\frac{bn}{\lg n}\right)$
- If we choose $r < \lg n$, then b/r term increases , and $n + 2^r$ term remains at $\Theta(n)$.
- If we choose $r > \lg n$, then $n + 2^r$ term gets big.
  Example: $r = 2\lg n \Rightarrow 2^r = 2^{2\lg n} = \left(2^{\lg n}\right)^2 = n^2$.
- So, to sort 2^{16} 32-bit numbers ($b \geq \lceil \lg n \rceil$), use $r = \lg 2^{16} = 16$. $\lceil 32/16 \rceil = 2$ passes.

Compare radix sort to merge sort and quicksort:

- (2^{20}) 32-bit integers.
- Radix sort: $\lceil 32/20 \rceil = 2$ passes.
- Merge sort/quicksort: $\lg n = 20$ passes.
- Although radix sort may make fewer passes than quicksort over the n keys, each pass of radix sort may take significantly longer.

- The version of radix sort that uses counting sort as the intermediate stable sort does not sort in place, which many of the $\Theta(n \lg n)$-time comparison sorts do.

- Thus, when primary memory storage is at a premium, we might prefer an in-place algorithm such as Insertion sort or Merge sort .

- Insertion sort is stable. When inserting $A[j]$ into the sorted sequence $A[1 \ldots j-1]$, we do it the following way: compare $A[j]$ to $A[i]$, starting with $i = j-1$ and going down to $i = 1$. continue at long as $A[j] < A[i]$.

- Merge sort as defined is stable, because when two elements compared are equal, the tie is broken by taking the element from array L which keeps them in the original order.

- Heapsort and quicksort are not stable.

Bucket sort

- Bucket sort assumes that the input is drawn from a uniform distribution and has an average-case running time of $O(n)$.
- Counting sort assumes that the input consists of integers in a small range.
- Bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval $[0, 1)$.

Bucket sort

- Assumes the input is generated by a random process that distributes 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 buckets in order, listing elements in each one.

- Assume $a, b \in \mathbb{R}$ and $a < b$

$$(a, b) = \{x | a < x < b\}$$ open interval

$$[a, b] = \{x | a \leq x \leq b\}$$ closed interval

$$[a, b) = \{x | a \leq x < b\}$$ Left-closed, right-open

$$(a, b] = \{x | a < x \leq b\}$$

$$(a, \infty) = \{x | x > a\}$$

$$[a, \infty) = \{x | x \geq a\}$$

$$(-\infty, b) = \{x | x < b\}$$

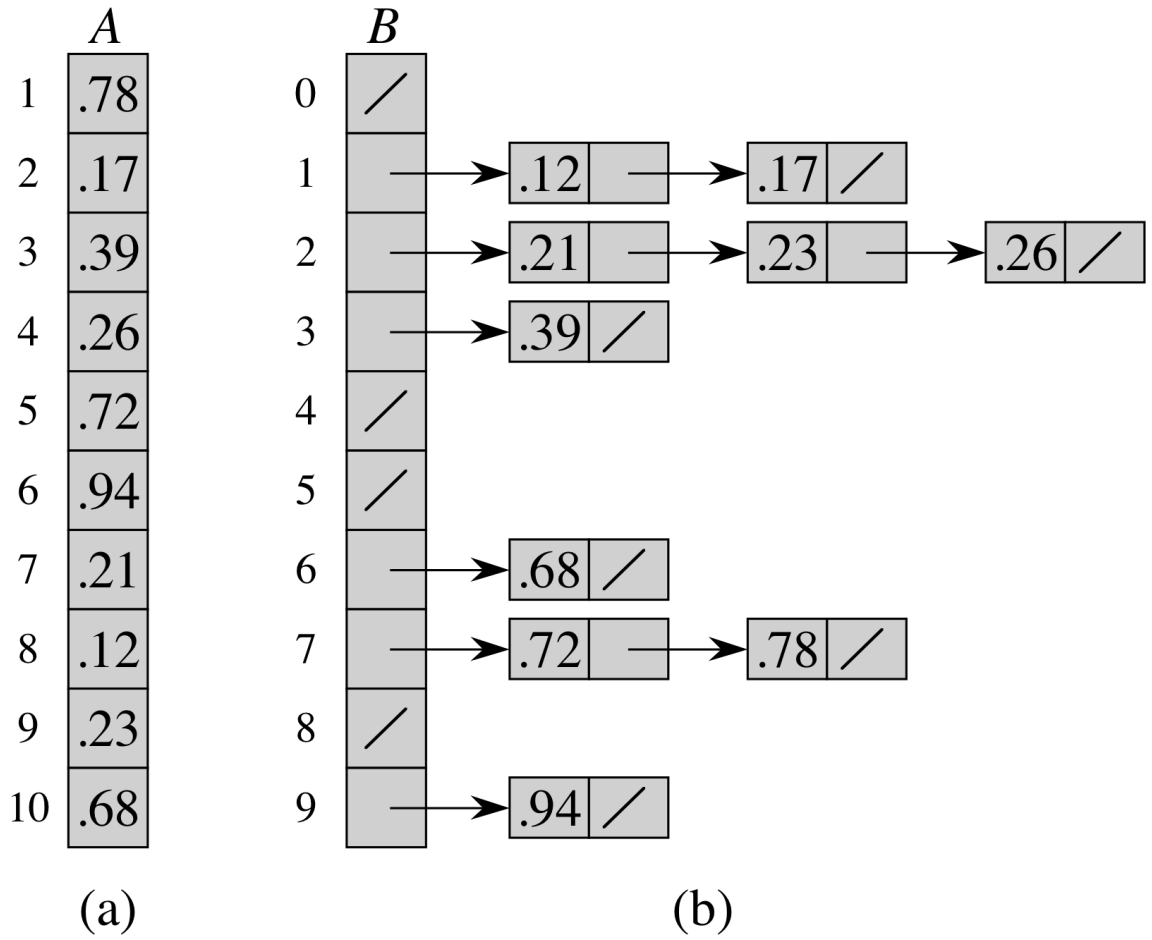
$$(-\infty, b] = \{x | x \leq b\}$$

$$(-\infty, \infty) = \mathbb{R}$$

$$[a, a] = \{a\}$$

BUCKET-SORT(A, n)

1 **Input:** $A[1 \ldots n]$, where $0 \leq A[i] < 1$ for all i.
2 **Auxiliary array:** $B[0 \ldots n-1]$ of linked lists, each list initially empty.
3 **for** $i \leftarrow 1$ ***to*** n **do**
4 insert $A[i]$ into list $B\,[\lfloor n \cdot A[i] \rfloor]$
5 **for** $i \leftarrow 0$ ***to*** $n-1$ **do**
6 sort list $B[i]$ with insertion sort
7 concatenate lists $B[0], B[1], \ldots, B[n-1]$ together in order
8 **return** the concatenated lists


A
1 .78
2 .17
3 .39
4 .26
5 .72
6 .94
7 .21
8 .12
9 .23
10 .68
B
0
1 .12 .17
2 .21 .23 .26
3 .39
4
5
6 .68
7 .72 .78
8
9 .94
(a)
(b)

Correctness: Consider $A[i], A[j]$. Assume without loss of generality that $A[i] \leq A[j]$. Then $\lfloor n \cdot A[i] \rfloor \leq \lfloor n \cdot A[j] \rfloor$. 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 of algorithm except insertion sorting take $\Theta(n)$ 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 insertion sort

Define a random variable:

- n_i = the number of elements placed in bucket $B[i]$.
- Because insertion sort runs in quadratic time, bucket sort time is $T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$
- Take expectations of both sides:

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

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

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

(Linearity of expectation)

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

$(E[aX] = aE[X])$

- Claim
- $\mathrm{E}[n_i^2] = 2 - (1/n)$ for $i = 0, \ldots, n-1$.

Proof of claim

Define indicator random variables:

- $X_{ij} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}$
- $\Pr\{A[j] \text{ falls in bucket } i\} = 1/n$
- $n_i = \sum_{j=1}^{n} X_{ij}$

Then

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

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

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

$$
\begin{aligned}
&\mathrm{E}\left[X_{ij}^2\right] \\
&= 0^2 \cdot \Pr\{A[j] \text{ doesn't fall in bucket } i\} + 1^2 \cdot \Pr\{A[j] \text{ falls in bucket } i\} \\
&= 0 \cdot \left(1 - \frac{1}{n}\right) + 1 \cdot \frac{1}{n} \\
&= \frac{1}{n}
\end{aligned}
$$

- $\mathrm{E}[X_{ij}X_{ik}]$ for $j \neq k$: Since $j \neq k$, X_{ij} and X_{ik} are independent random variables
- $\Rightarrow \mathrm{E}[X_{ij}X_{ik}] = \mathrm{E}[X_{ij}]\,\mathrm{E}[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}$

Therefore:

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

$$= \sum_{j=1}^{n} \frac{1}{n} + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} \frac{1}{n^2}$$

$$= n \cdot \frac{1}{n} + 2\binom{n}{2}\frac{1}{n^2}$$

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

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

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

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

■

Therefore:

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

$$= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{O}\left(2 - \frac{1}{n}\right)$$

$$= \Theta(n) + \mathrm{O}(n)$$

$$= \Theta(n)$$

- The worst-case running time for the bucket-sort algorithm occurs when the assumption of uniformly distributed input does not hold.

- If, for example, all the input ends up in the first bucket, then in the insertion sort phase it needs to sort all the input, which takes $O(n^2)$ time.

- Use a worst-case $O(n \lg n)$ time algorithm.

- Again, not a comparison sort. Used a function of key values to index into an array.

- This is a **probabilistic analysis**—we used probability to analyze an algorithm whose running time depends on the distribution of inputs.

- With bucket sort, if the input isn’ t drawn from a uniform distribution on [0,1), it may still run in linear time. As ling as the input has the property that the sum of the squares of the bucket sizes is linear in the total number of elements.

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