Chapter 8
Sorting in Linear Time

The slides for this course are based on the course textbook: Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 2nd edition, The MIT Press, McGraw-Hill, 2001.

• Many of the slides were provided by the publisher for use with the textbook. They are copyrighted, 2001.

• These slides are for classroom use only, and may be used only by students in this specific course this semester. They are NOT a substitute for reading the textbook!

Chapter 8 Topics

- Lower bounds for sorting
- Counting sort
- Radix sort
- Bucket sort

Lower Bounds for Sorting

- All the sorts we have examined so far work by “key comparisons”. Elements are put in the correct place by comparing the values of the key used for sorting.
- Mergesort and Heapsort both have running time $\Theta(n \lg n)$
- This is a lower bound on sorting by key comparisons

Sorting by Key Comparisons

- Input sequence

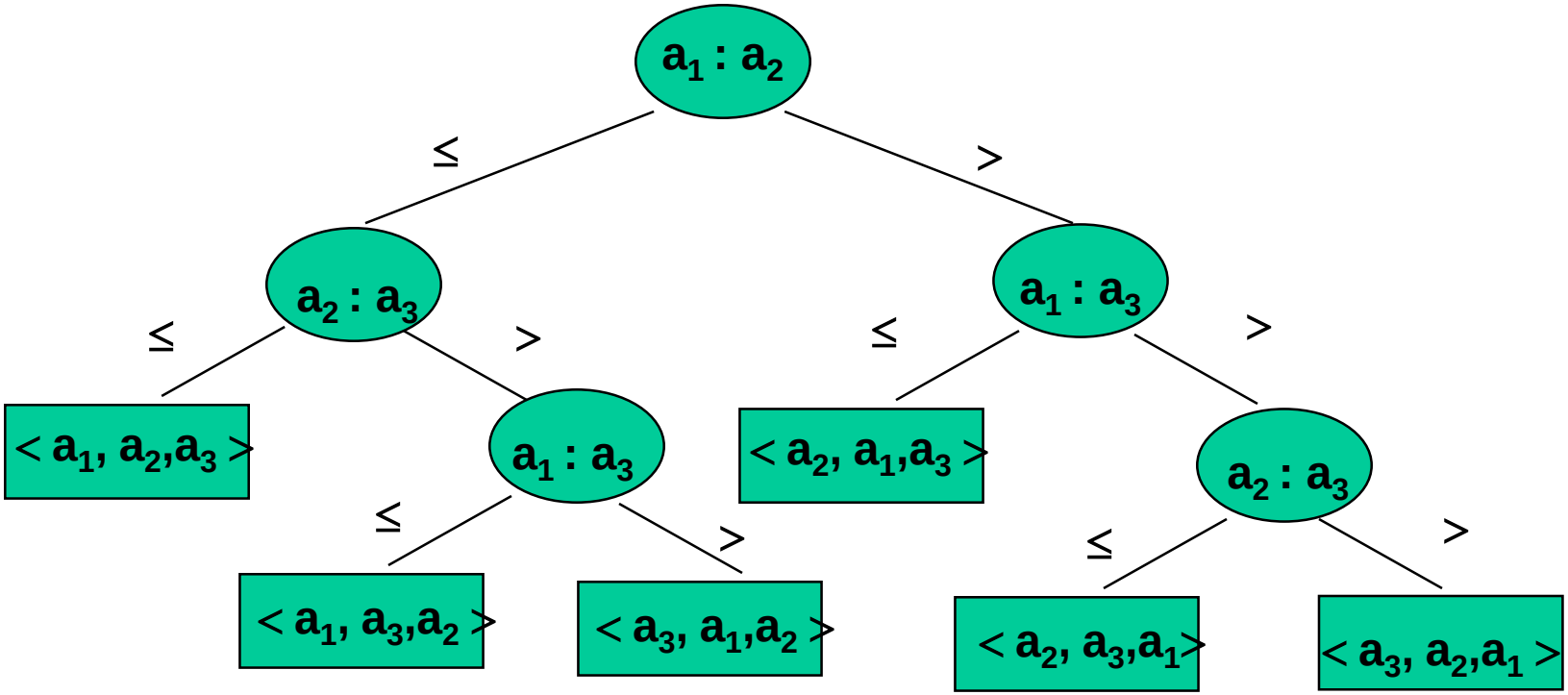
  $<a_1, a_2, \ldots a_n,>$
- Possible tests of a_i between a_j

  $< \quad > \quad \leq \quad \geq \quad =$
- If assume unique elements,
  - test for equality is unnecessary (=)
  - $< \quad > \quad \leq \quad \geq$ all yield the same information about the relative order of a_i and a_j
- So, use only $a_i <= a_j$

Decision Tree Model

- We can view a comparison sort abstractly by using a *decision tree*.
- The decision tree represents all possible comparisons made when sorting a list using a particular sorting algorithm
- Control, data movement, and other aspects of the algorithm are ignored
- Assume:
  - All elements are distinct.
  - All comparisons are of the form ai ≤ aj

Decision Tree for Insertion Sort (n = 3)



$a_1 : a_2$ means: "compare a_1 and a_2"

$< a_2, a_1, a_3 >$ means: $a_2 \leq a_1 \leq a_3$

Lower Bounds for Comparison Sorts

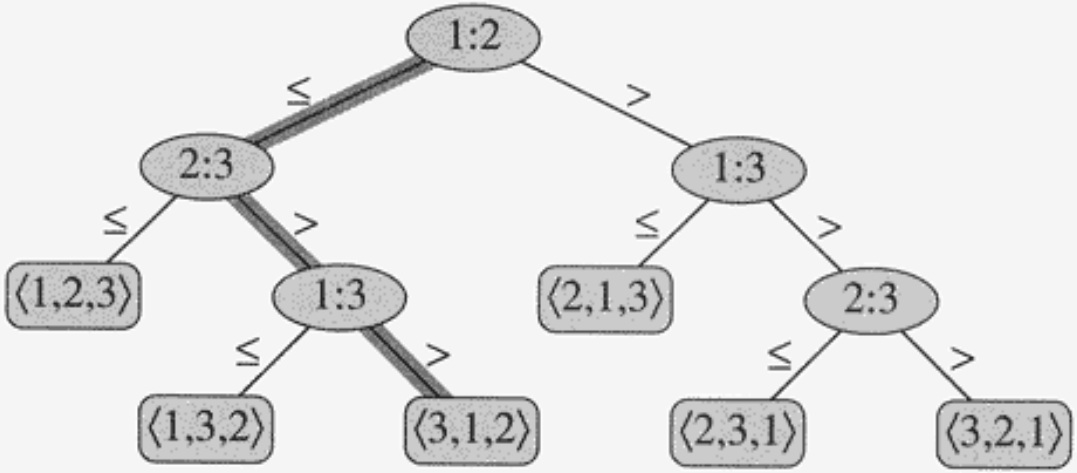


Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i{:}j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots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

Permutations of n elements

- There are n! permutations of n elements
- That means that the decision tree which results in all possible permutations of elements must have n! leaves
- The longest path from the root to a leaf represents the worst case performance of the algorithm
- So the worst case performance is the height of the decision tree

Theorem 8.a

Any decision tree that sorts n elements has height Ω (n lg n).

Proof:

Consider a decision tree of height h with l leaves that sorts n elements.

There are $n!$ permutations of n elements.

The tree must have at least $n!$ leaves since each permutation of input must be a leaf.

A binary tree of height h has no more than 2^h leaves.

Therefore the decision tree has no more than 2^h leaves.

Thus: $n! \leq l \leq 2^h$

Therefore: $n! \leq 2^h$

Take the logarithm of both sides:

$\lg(n!) \leq h$, or, equivalently, $h \geq \lg(n!)$

Using Stirling's approximation of n! we have:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$$

Since $\sqrt{2\pi n}$ and $\left(1+\Theta\left(\frac{1}{n}\right)\right)$ are > 1,

$$n! > \left(\frac{n}{e}\right)^n$$

Since $h \geq \lg(n!)$ and $n! > \left(\frac{n}{e}\right)^n$

we have: $h \geq \lg\left(\frac{n}{e}\right)^n$

$$h = n\lg n - n\lg e$$

$$h = \Omega(n\lg n)$$

Lower Bound for Worst Case

Theorem 8.1:

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

Lower Bound for Worst Case

Proof:

By the previous theorem we know that any decision tree that sorts n elements has height $\Omega(n \lg n)$. Since the decision tree explicitly models the comparison process, no comparison sort algorithm can guarantee any fewer than $\Omega(n \lg n)$ comparisons in the worst case.

Lower Bound for Worst Case

Corollary:

Heapsort and merge sort are asymptotically optimal comparison sorts.

Proof:

The $O(n \lg n)$ upper bounds on the running times for heapsort and merge sort match the $\Omega(n \lg n)$ worst-case lower bound from the theorem.

Counting Sort

- Assumes each of the *n* input elements is an integer in the range 0 to *k*, for some integer *k*
- For each element x, determine the number of values $\leq$ x.
- Requires three arrays
  - An input array A[1..*n*]
  - An array B[1..*n*] for the sorted output
  - An array C[0..*k*] for counting the number of times each element occurs (temporary working storage)

Counting Sort

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

Counting Sort

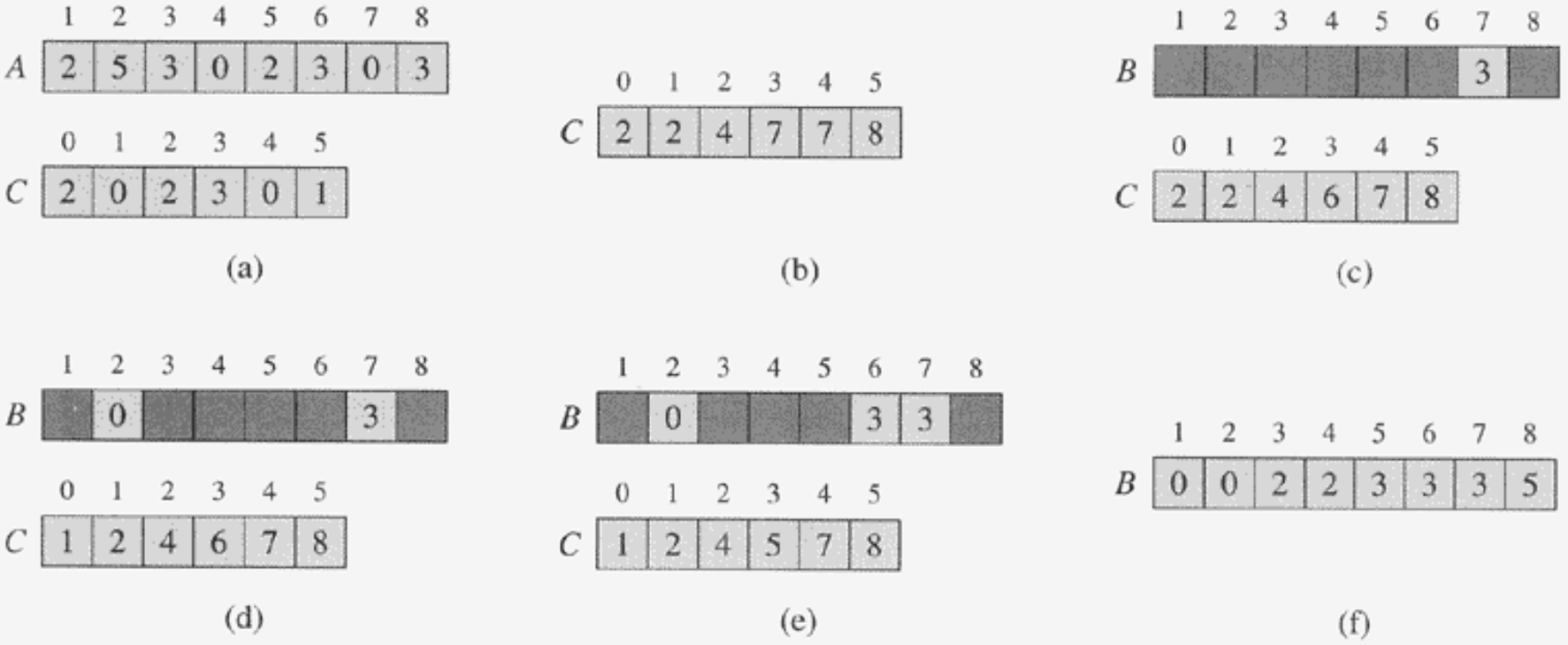


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after 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.

Complexity of Counting Sort

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

Complexity of Counting Sort

Lines 1-2	$\Theta(k)$
Lines 3-4	$\Theta(n)$
Lines 6-7	$\Theta(k)$
Lines 9-11	$\Theta(n)$
Total	$\Theta(n+k)$

Counting Sort

- Beats the lower bound of $\Omega(n \lg n)$ because it is not a comparison sort
- Makes assumptions about the input data
- Is a **stable** sort: numbers with the same value appear in the output array in the same order as they do in the input array

Radix Sort

Radix sort assumes that each element in an array A with *n* elements consists of a number with *d* digits, where digit 1 is the lowest-order digit and digit *d* is the highest-order digit.

Radix Sort

```
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

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

One method of implementing the radix sort involves the use of bins, or auxiliary arrays. We will need a number of bins equal to the base (or *radix*) of our numbering system. For decimal (base 10) numbers we will need 10 extra bins one for each of the numbers 0 through 9.

To implement radix sort, imagine the array is a deck of cards with numbers on them.

Radix Sort

Dealing from the bottom of the deck, take each card, look at the digit in its 1's column, and put it face-up into the appropriate bin (0 through 9).

Without disturbing the order of the cards within a bin, pick up the cards in the 0's bin, then pick up the cards in the 1's bin and put them on top of the cards you are already holding, then do the same with the 2's bin, etc., finishing with the 9's bin.

Repeat this process for the 10's column, the 100's column, etc.

Your deck is now sorted from bottom to top.

Radix Sort

What do we do if we have some numbers with more digits than others?

Just add (or pretend to add) extra zeros on the left so that all numbers have the same number of digits.

So `307` becomes `307`

`51` `051`

`4` `004`

Radix Sort

Will this work with real numbers?

Yes. You will have to pad both the integer part and the decimal part with zeros so that all of the numbers have the same number of digits in the integer part and all numbers have the same number of digits in the decimal part.

002.000
123.456
003.142
027.200

Radix Sort

What is the running time of this sort?

If d is the number of digits in the number with the greatest number of digits, then the running time is O($d * n$), or just O(n).

Proof:

We have to go through the deck d times. We have n cards and we have to handle each card once each time we go through the deck. So the total amount of work involved is O($d * n$). For large values of n, n is the dominant term.

Radix Sort

How much space does radix sort use?

It depends on how line 2 of the algorithm is implemented. If we use bins, as in our example, the sort is very inefficient in its use of space.

The worst case is if we have an array of *n* identical numbers (e.g., 9876543210). Then we will need 10 bins of size n.

In the best case, we still need 10 bins each of size n/10.

Bucket Sort

Bucket sort also makes an assumption about the elements it is sorting. It assumes that the elements are generated by a random process that distributes the elements uniformly over the interval [0,1).

Bucket sort works by dividing the interval [0,1) into *n* equal-sized subintervals (or *buckets*), and then dealing the *n* input numbers into the appropriate buckets.

Bucket Sort

Each bucket will *probably* end up with only 1 or two elements in it - a few elements at most, we expect.

We can use any old sort, even Insertion sort, to sort these items in the buckets.

Then we recombine the elements in the buckets (subintervals) back into a single array.

Bucket Sort

```
BUCKET-SORT(A)
1  n ← length[A]
2  for i ← 1 to n do
3     insert A[i] into list B[⌊nA[i]⌋]
4  for i ← 0 to n-1 do
5     sort list B[i] with insertion sort
6  concatenate lists B[0], B[1], …, B[n-
   1] together in order
```

Bucket Sort

Given array A = <.49, .37, .26, .81, .65>, let's see how BucketSort will work.

First, create *n* buckets. In this case, *n* = 5.

.80 - .99 

.60 - .79 

.40 - .59 

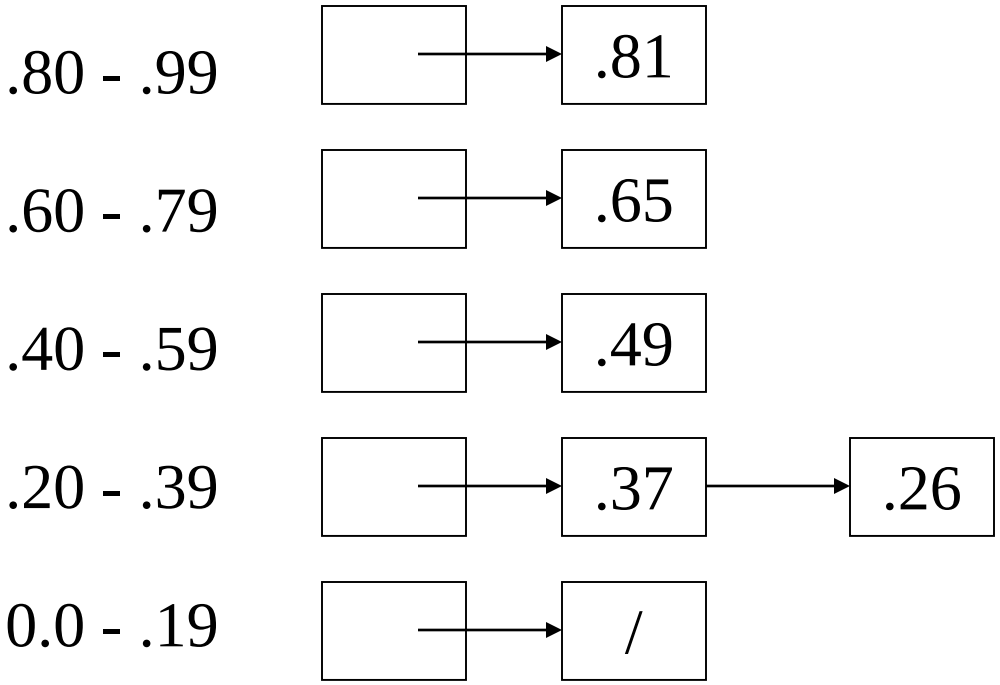
.20 - .39 

0.0 - .19 

Each of the buckets will represent 1/n of the interval [0,1).

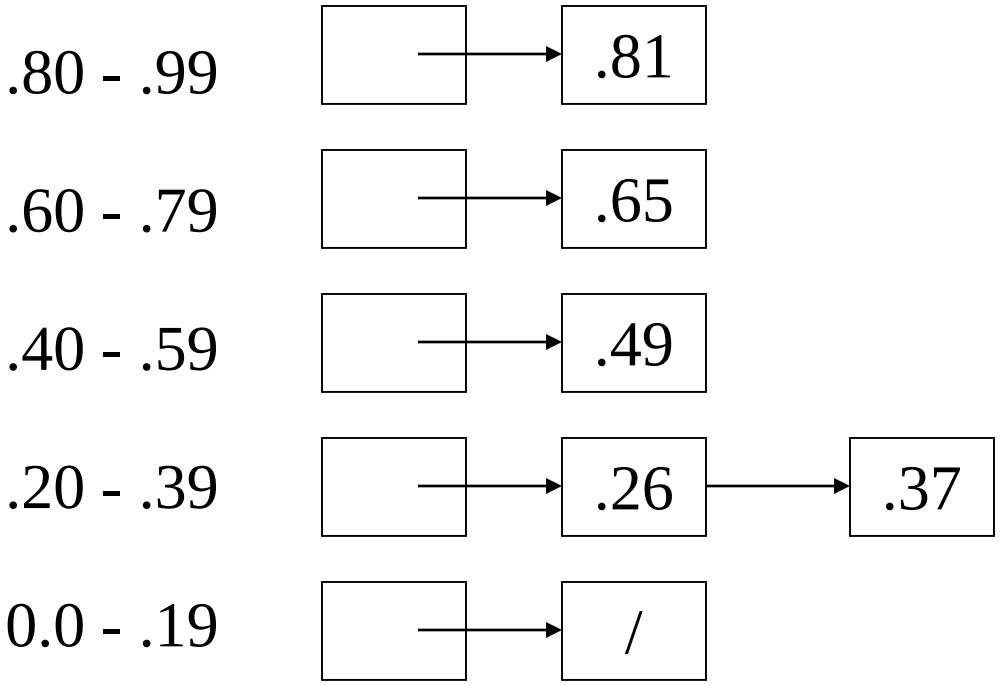
Bucket Sort

Now take array A = <.49, .37, .26, .81, .65> and distritbute its elements into the buckets.



Bucket Sort

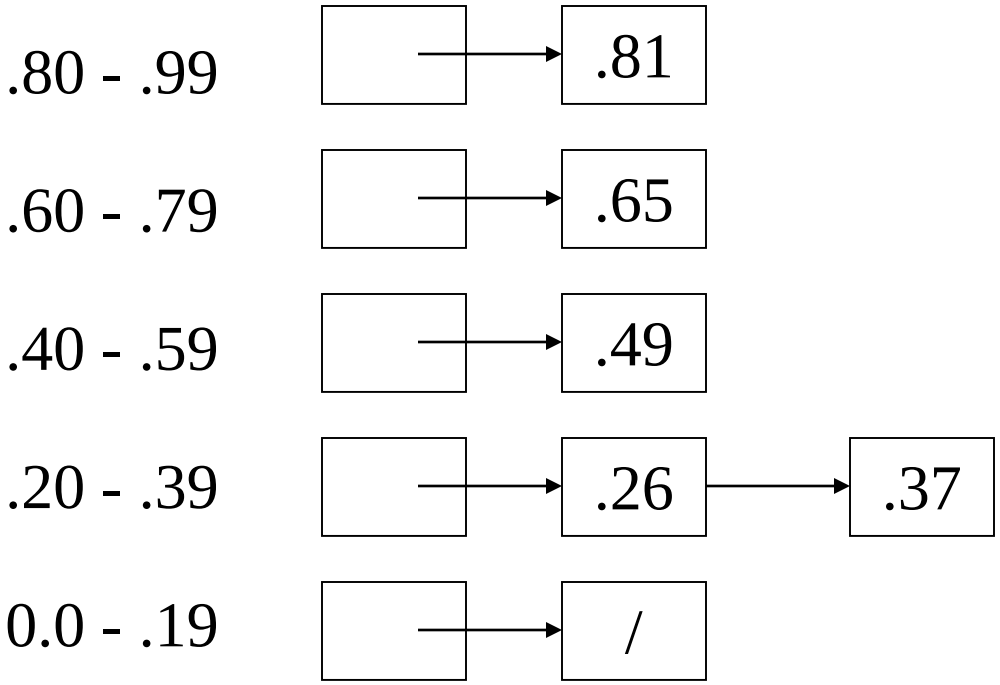
Now sort each bucket as needed. Here, only one bucket will require sorting, and it has only two elements in it, so we can use any old sort.



Bucket Sort

Finally, concatenate the buckets together to create the sorted array:

<.26, .37., .49, .65, .81>



Bucket Sort

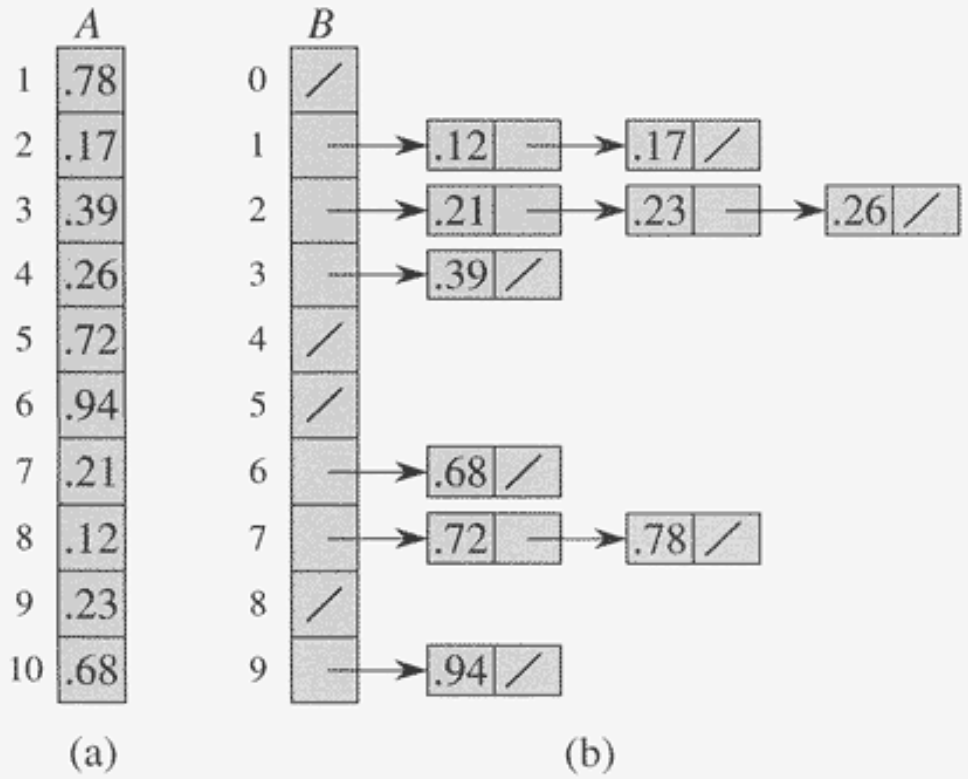


Figure 8.4 The operation of BUCKET-SORT. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

Analysis of Bucket sort

The only line of the Bucket sort algorithms that should bother us is line 5, where we sort each bucket with Insertion sort; won't that make the whole sort $O(n^2)$?

Well, no. If n = 10,000 and the maximum number of items in any bucket is, say, 3, then line 3 will cost us at most 9 each time we hit it. If all buckets have 3 items, our cost will still be only O(9*n) = O(n).

Analysis of Bucket sort

Remember that Bucket sort assumes that the elements of the array to be sorted are generated by a random process that distributes the elements uniformly over the interval [0,1).

If this truly is the case, Bucket sort will run O(n).

If our assumption is false, all bets are off!

Conclusion

We can sort in O(n) time *if and only if* we know something about the nature of the items we are sorting that enables us to use a specialized algorithm.

But, under normal circumstances, O(n lg n) is a provable lower bound on sorting.

A final word about sorts …

Sorting is one of the central problems of computer science. Many other sorting techniques have been developed, including:

Pancake sort

Bead sort

Pigeonhole sort

“Perfect hashing” sort

. . .