Chap7.Quick Sort

- What is quick sort?
- How does it work?
- Performance of quick sort
- Randomized version of quick sort

Quick Sort

- by C.A.R. Hoare(British computer scientist at age 26, 1960)
- Worst-case execution time : $\boldsymbol{\Theta(n^2)}$.
- Average-case execution time : $\boldsymbol{\Theta(n \log n)}$.

- Empirical and analytical studies show that quicksort can be ***expected*** to be twice as fast as its co mpetitors.

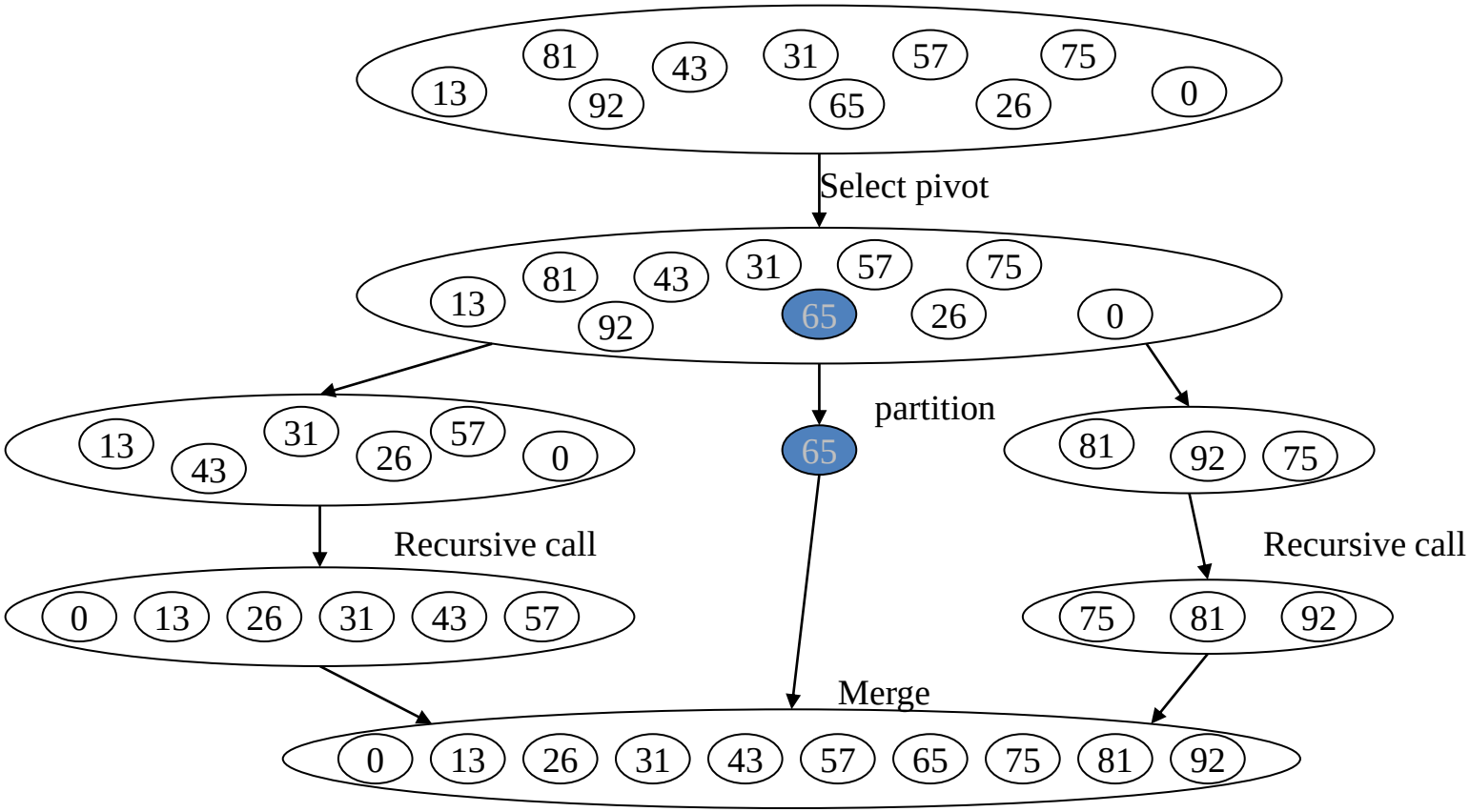
Quick Sort

- Divide-and-conquer approach to sorting
- Like Merge Sort, except
  - Don’t divide the array in half
  - Partition the array based elements being less than or greater than some element of the array (*the pivot*)
  - i.e., divide phase does all the work; merge phase is trivial.

Quicksort

- Quicksort is another divide-and-conquer algorithm.
- Basically, what we do is divide the array into two subarrays, so that all the values on the left are smaller than the values on the right.
- We repeat this process until our subarrays have only 1 element in them.
- When we return from the series of recursive calls, our array is sorted.

QuickSort Example



Quicksort

- **Divide:** Partition A[p..r] into two (possibly empty) subarrays A[p..q-1] and A[q+1 .. r] such each element of A[p..q-1] $\leq$ A[q] and A[q] $\leq$ each element of A[q+1..r]. Compute the index q as part of this partitioning procedure.
- **Conquer:** Sort the two subarrays by recursive calls to quicksort.
- **Combine:** Since the subarrays are sorted in place, no work is needed to combine them: A[p..r] is now sorted.

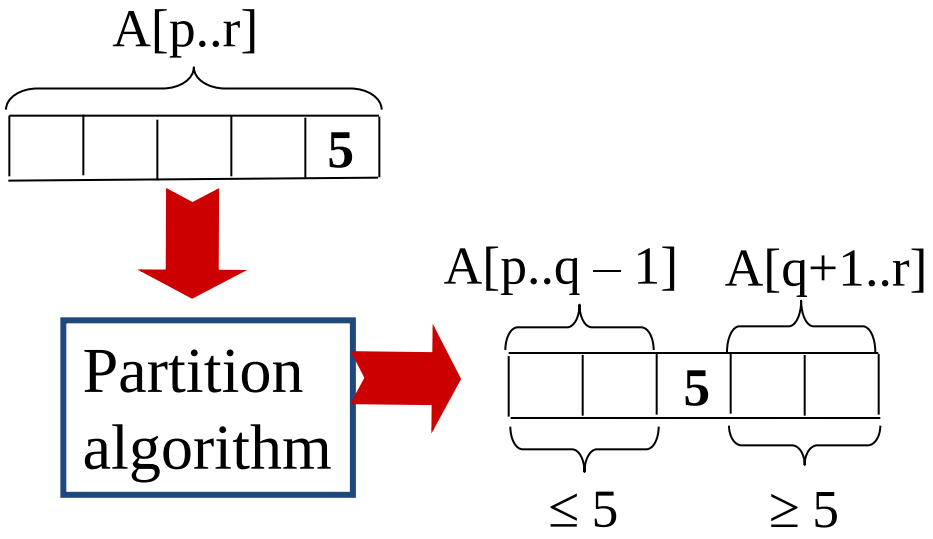
The Quicksort Algorithm

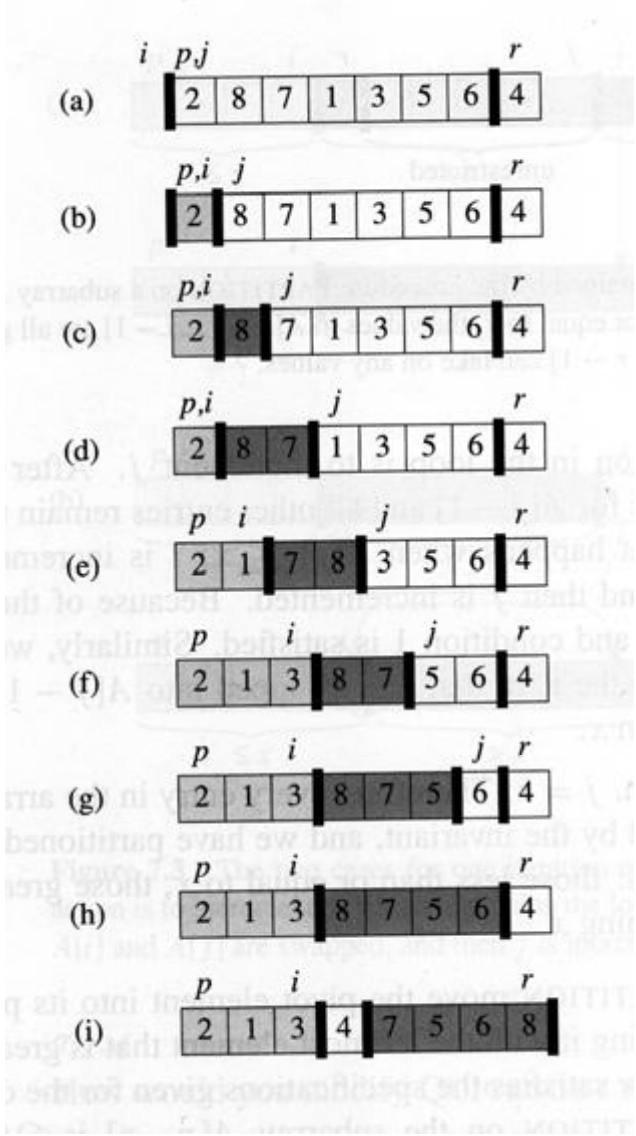
```
QUICKSORT(A,p,r)
1   if p < r
2       then q ← PARTITION(A,p,r)
3            QUICKSORT(A,p,q-1)
4            QUICKSORT(A,q+1,r)

Initial call:
QUICKSORT(A,1, length[A])
```

Partition Algorithm

```
PARTITION(A,p,r)
1    x ← A[r]
2    i  ← p - 1
3    for j ← p to r-1
4         do if A[j] ≤ x
5              then i  ← i + 1
6                   exchange A[i] ↔ A[j]
7    exchange A[i+1] ↔ A[r]
8    return i+1
```

Simple example






```
PARTITION(A,p,r)
1   x ← A[r]
2   i ← p - 1
3   for j ← p to r-1
4       do if A[j] ≤ x
5           then i ← i + 1
6                exchange A[i] ↔ A[j]
7   exchange A[i+1] ↔ A[r]
8   return i+1
```

Example

	p									r
initially:	2	5	8	3	9	4	1	7	10	**6**
	i	j								
next iteration:	2	5	8	3	9	4	1	7	10	**6**
		i	j							
next iteration:	2	5	8	3	9	4	1	7	10	**6**
		i	j							
next iteration:	2	5	8	3	9	4	1	7	10	**6**
		i		j						
next iteration:	2	5	3	8	9	4	1	7	10	**6**
			i		j					

note: pivot (x) = 6

```
PARTITION(A,p,r)
1   x ← A[r]
2   i ← p - 1
3   for j ← p to r-1
4       do if A[j] ≤ x
5           then i ← i + 1
6                exchange A[i] ↔ A[j]
7   exchange A[i+1] ↔ A[r]
8   return i+1
```

Example (Continued)

next iteration:	2	5	3	8	9	4	1	7	10	**6**
			i		j					
next iteration:	2	5	3	8	9	4	1	7	10	**6**
			i			j				
next iteration:	2	5	3	4	9	8	1	7	10	**6**
				i			j			
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i			j		
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i				j	
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i					j
after final swap:	2	5	3	4	1	**6**	9	7	10	8
					i					j

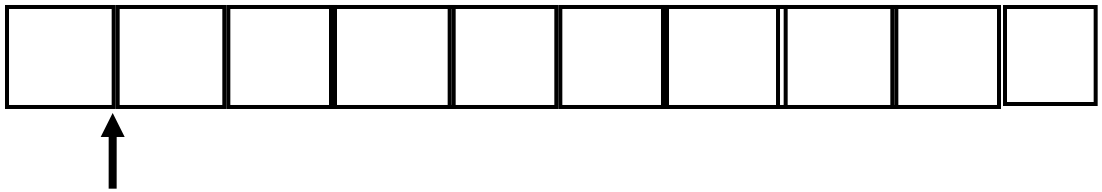
```
PARTITION(A,p,r)
1   x ← A[r]
2   i ← p - 1
3   for j ← p to r-1
4       do if A[j] ≤ x
5           then i ← i + 1
6                exchange A[i] ↔ A[j]
7   exchange A[i+1] ↔ A[r]
8   return i+1
```

Performance of Quicksort

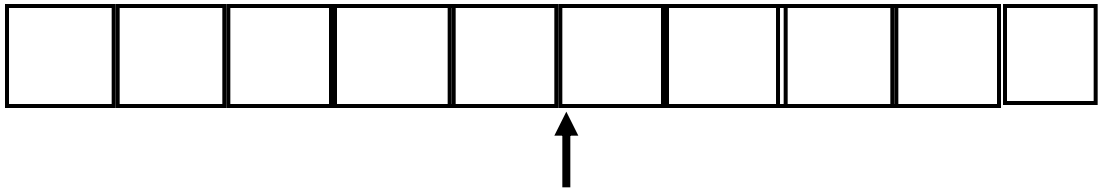
- Depends on whether the partition is balanced or unbalanced:
  - Balance of partition depends on selection of pivot
  - If balanced, runs as fast as Merge sort
  - If unbalanced, runs as slowly as Insertion sort

Worst/Best case partitioning

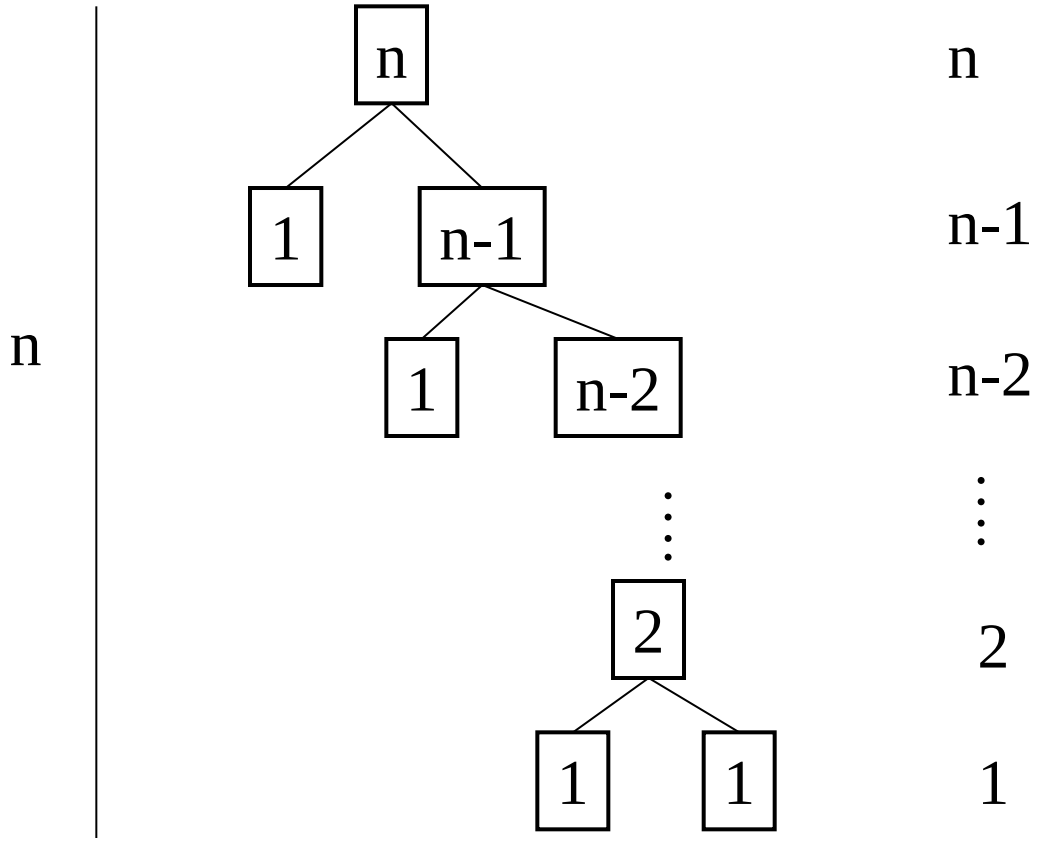
- **Worst case:**
  - One partition contains n – 1 elements
  - The other partition contains 1 element



- **Best case:**
  - Both partitions are of equal size



Worst case partitioning



Worst case performance

- Assume we have a maximally unbalanced partition at each step, splitting off just 1 element from the rest each time. This means we will have to call Partition n-1 times.
- The cost of Partition is: $\Theta(n)$
- So the recurrence for Quicksort is:

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= T(n-1) + \Theta(n)$$
$$= \Theta(n^2) \quad \text{by iterative substitution}$$

Best case performance

- Size of each subproblem $\leq n/2$.
  - One of the subproblems is of size $\lfloor n/2 \rfloor$
  - The other is of size $\lceil n/2 \rceil - 1$.
- Recurrence for running time
  - $T(n) \leq 2T(n/2) + \text{PartitionTime}(n)$

    $= 2T(n/2) + \Theta(n)$
- $\boldsymbol{T(n) = \Theta(n \log n)}$ by master theorem case2

Average Case

- Average case analysis is complex and difficult.
- However, we can observe that average-case performance is much closer to best-case than worst case.
- Suppose split is always 9-to-1
- Recurrence:

$$T(n) \leq T(9n/10) + T(n/10) + \Theta(n)$$

$$= T(9n/10) + T(n/10) + cn$$

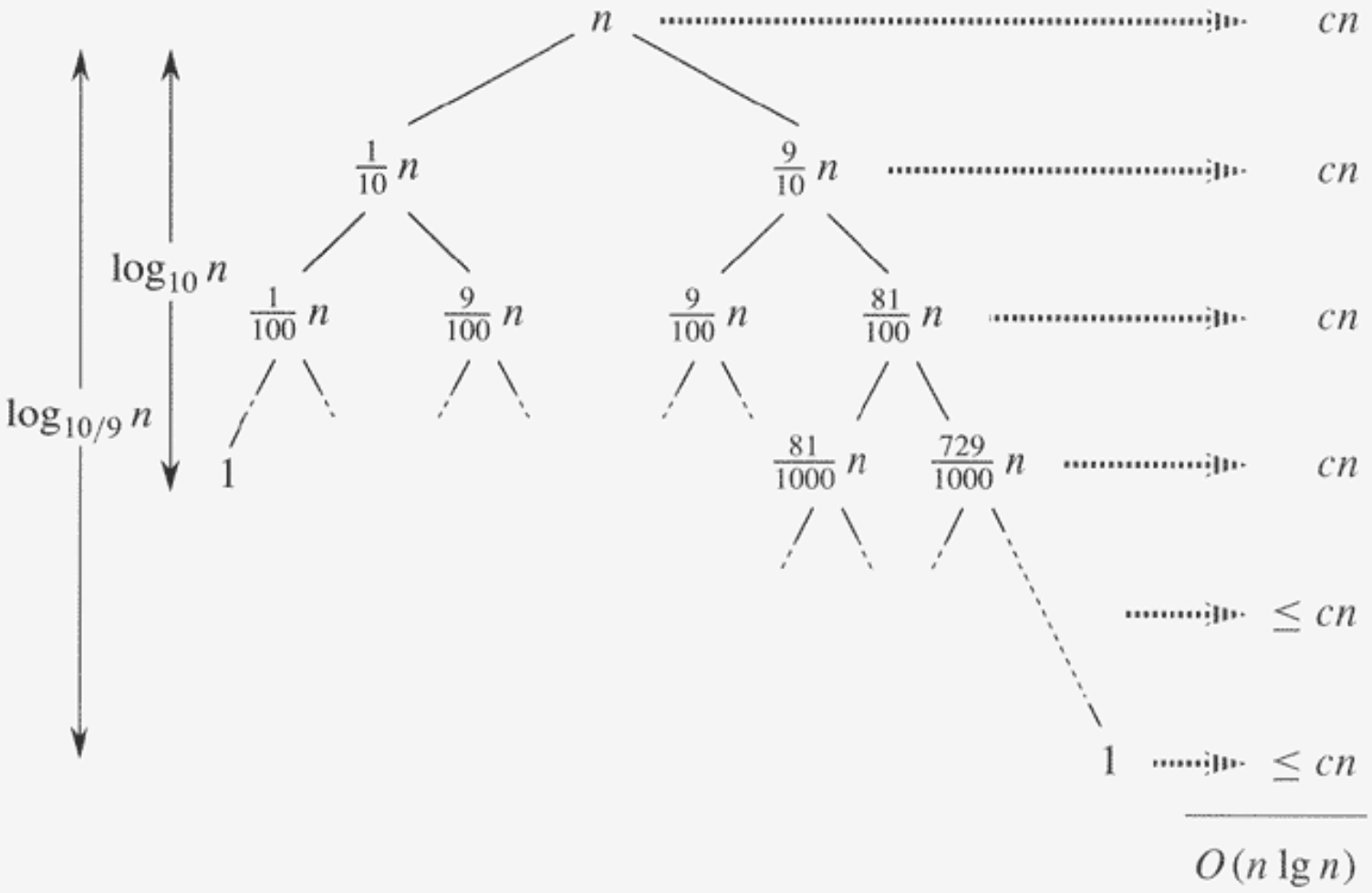


Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

Average case

- What if we have a 99-1 split?
  $T(n) \leq T(99n/100) + T(n/100) + \Theta(n)$
- We still have a running time of $O(n \log n)$
- As long as it's a constant, the base of the log doesn't matter in asymptotic notation.
- Any split of constant proportionality will yield a recursion tree of depth $\Theta(\log n)$.
- So whenever the split of constant proportionality, Quicksort performs on the order of $O(n \log n)$.

Picking the Pivot

- How would you pick one?
- Strategy 1: Pick the first element in **S**
  - Good for a randomly populated array
  - What if input **S** is sorted, or even mostly sorted?
    - All the remaining elements would go into either **S1** or **S2**!
- Strategy 2: Pick the pivot randomly
  - Good in practice if "truly random"
  - Still possible to get some bad choices
  - Requires execution of random number generator

Picking the Pivot (contd.)

- Strategy 3: Median-of-three Partitioning
  - *Ideally*, the pivot should be the median of input array **S**
    - Median = element in the middle of the sorted sequence
  - Would divide the input into two almost equal partitions
  - Unfortunately, its hard to calculate median quickly, without sorting first!
  - So find the approximate median
    - Pivot = median of the left-most, right-most and center element of the array **S**
    - Solves the problem of sorted input

Picking the Pivot (contd.)

- Example: Median-of-three Partitioning
  - Let input S = {6, 1, 4, 9, 0, 3, 5, 2, 7, 8}
  - left=0 and S[left] = 6
  - right=9 and S[right] = 8
  - center = (left+right)/2 = 4 and S[center] = 0
  - Pivot
    = Median of S[left], S[right], and S[center]
    = median of 6, 8, and 0
    = S[left] = 6

Quick Sort vs. Insertion Sort

- For small arrays ($N \leq 20$),
  - Insertion sort is faster than quicksort
- Quicksort is recursive
  - So it can spend a lot of time sorting small arrays
- Hybrid algorithm:
  - Switch to using insertion sort when problem size is small (say for $N < 20$)

QuickSort vs. MergeSort

- Main problem with quicksort:
  - QuickSort may end up dividing the input array into subproblems of size 1 and N-1 in the worst case, at every recursive step (unlike merge sort which always divides into two halves)
    - When can this happen?
    - Leading to $O(N^2)$ performance

  ⇨ Need to choose pivot wisely (but efficiently)
- MergeSort is typically implemented using a temporary array (for merge step) "not in place"
  - QuickSort can partition the array "in place"

Randomized Version of Quicksort

- When an algorithm has an average case performance and worst case performance that are very different, we can try to minimize the odds of encountering the worst case.
- We can:
  - Randomize the input
  - Randomize the algorithm

Randomized Version of Quicksort

- **Randomizing the input**

  With a given set of input numbers, there are very few permutations that produce the worstcase performance in Quicksort.

  We can randomly permute the numbers in an element array in O(n) time.

  For Quicksort, add an initial step to randomize the input array.

  Running time is now independent of input ordering.

Randomized Version of Quicksort

- **Randomizing the input**

  In standard Quicksort, the worst case is encountered when we choose a bad pivot.
  If the input array is already sorted (or inverse sorted), we will always pick a bad pivot.

  But if we pick our pivot randomly, we will rarely get a bad pivot.

  So, randomly choose a pivot element in A[p..r].

  Running time is now independent of input ordering.

Randomized Partition

```
RANDOMIZED-PARTITION (A, p, r)
1    i ← RANDOM (p, r)
2    exchange A[r] ↔ A[i]
3    return PARTITION (A, p, r)
```

Randomized Quicksort

```
RANDOMIZED-QUICKSORT (A, p, r)
1       if p < r
2       then q ← RANDOMIZED-PARTITION (A, p, r)
3               RANDOMIZED-QUICKSORT (A, p, q-1)
4               RANDOMIZED-QUICKSORT (A, q+1, r)
```

Conclusion

- Quicksort runs $O(n \log n)$ in the best and average case, but $O(n^2)$ in the worst case.
- Worst case scenarios for Quicksort occur when the array is already sorted, in either ascending or descending order.
- We can increase the probability of obtaining average-case performance from Quicksort by using Randomized-partition.

Chap8.Sorting in Linear Time

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

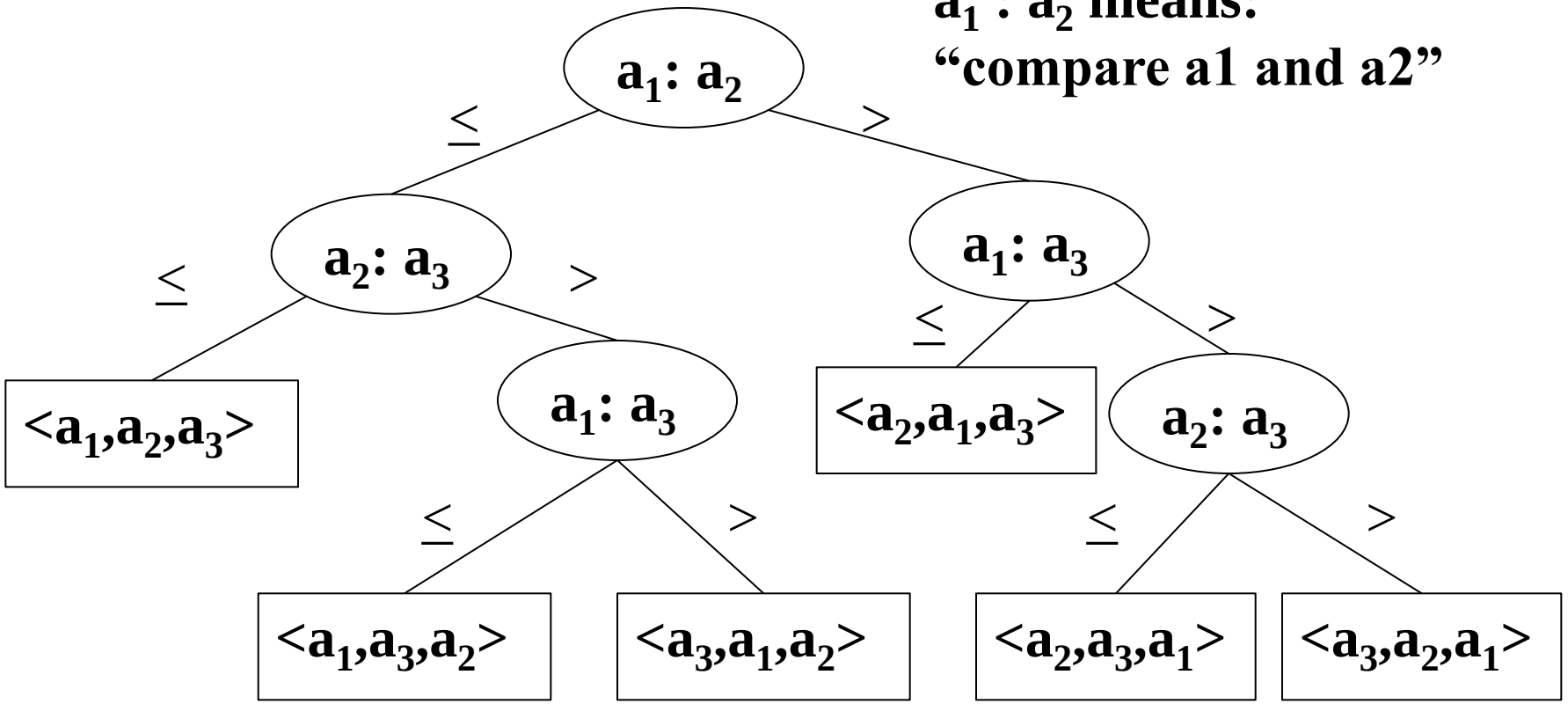
Lower Bounds for Sorting

- 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 $\Theta(n \log n)$
- This is a lower bound on sorting by key comparisons

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
- Assume:
  - All elements are distinct.
  - All comparisons are of the form ai ≤ aj

Decision Tree for Insertion Sort (n = 3)



< a_2, a_1,a_3 > means: a2 ≤ a1 ≤ a3

Lower Bounds for Comparison Sorts

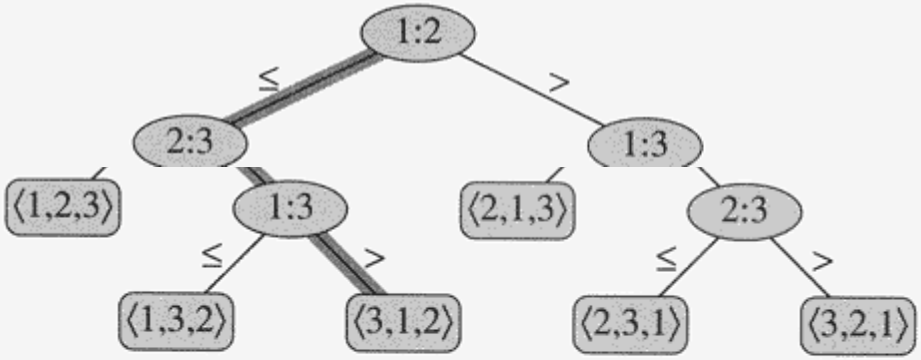


Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i:j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. 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

A Lower Bound for Worst Case

- ***Theorem* 8.1:**
  Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.
- **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 Lower Bound for Worst Case

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:

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

$\boldsymbol{n! > (n/e)^n}$ by using Stirling's approximation of n!

Hence, $h \geq \log(n!)$

$\geq \log(n/e)^n$

$= n \log n - n \log e = \Omega(n \log n)$

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

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. $C[A[j]] \leftarrow C[A[j]]-1$

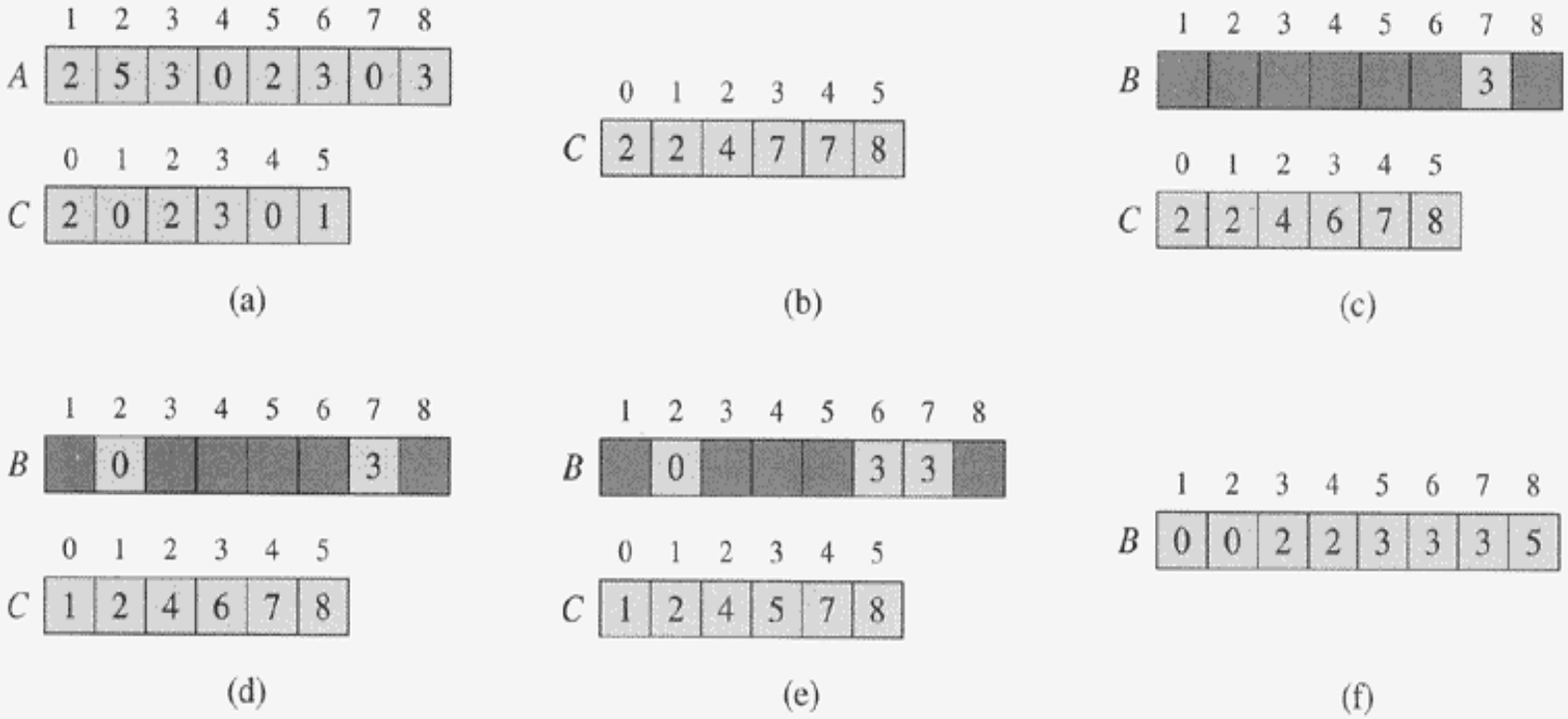


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after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

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

```
1. for i ← 0 to k                                   } Θ(k)
2.       do C[i] ← 0
3. for j ← 1 to length[A]                           } Θ(n)
4.       do C[A[j]] ← C[A[j]] + 1
5. for i ← 1 to k                                   } Θ(k)
6.       do C[i] ← C[i] + C[i –1]
7. for j ← length[A] downto 1                       } Θ(n)
8.       do B[C[A[ j ]]] ← A[j]
9.          C[A[j]] ← C[A[j]]–1
```

Total **Θ(n+k)**

Complexity of Counting Sort

- The overall time is $O(n+k)$.
  When we have $k=O(n)$, the worst case is $O(n)$.
- Beats the lower bound of (*n log n)* because it is not a comparison sort
- No comparisons made: it uses actual values of the ele ments to index into an array.

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.
- 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 becomes 051
  4 becomes 004

Running Time of 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.

Running Time of 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

- 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.

Bucket Sort Example

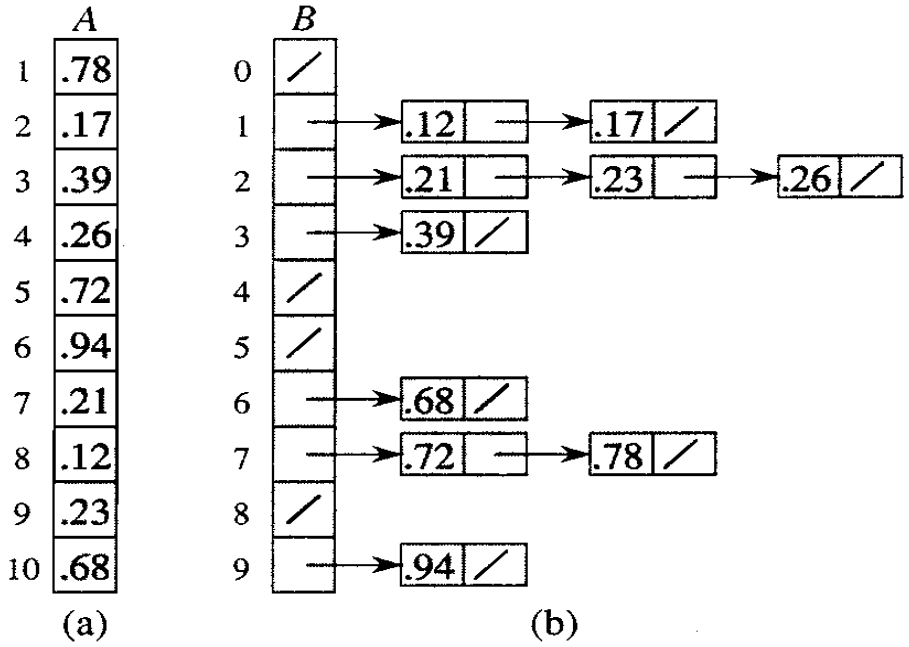


Figure 9.4 The operation of BUCKET-SORT. (**a**) The input array $A[1..10]$. (**b**) The array $B[0..9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1],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

BucketSort(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

Analysis of Bucket Sort

- Relies on no bucket getting too many values.
- All lines except insertion sorting in line 5 take $O(n)$
- 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 of Bucket Sort

- Random variable n_i = # 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 of Bucket Sort

$$
\begin{aligned}
E[n_i^2] &= \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \le j \le n} \sum_{\substack{1 \le k \le n \\ k \ne j}} \frac{1}{n^2} \\
&= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} \\
&= 1 + \frac{n-1}{n} \\
&= 2 - \frac{1}{n}.
\end{aligned}
$$

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