Chapter 7
Quicksort

1

Outlines: Quicksort

- Quicksort: Advantages and Disadvantages
- Quicksort:
  - **Quicksort Function**
  - **Partition Function**
  - **Choice Of Pivot**
    - **Rightmost or Leftmost**
    - **Randomly**
    - **Median-of-Three Rule**
  - **Partitioning using Additional Array**
  - **In-Place Partitioning**
  - **Runtime of Quicksort (Worst, Best, Average)**
- Randomized Quicksort

2

Quicksort

- Quicksort pros:
  - Sorts in place
  - Sorts $O(n \lg n)$ in the average case
  - Very efficient in practice
- Quicksort cons:
  - Sorts $O(n^2)$ in the worst case
  - not stable
    - does not preserve the relative order of elements with equal keys
    - Sorting algorithm is **stable** if 2 records with the same key stay in original order
  - But in practice, it's quick
  - And the worst case doesn't happen often … **sorted**

3

Quicksort

- Another divide-and-conquer algorithm:
- *Divide*: $A[p…r]$ is partitioned (rearranged) into two nonempty subarrays $A[p…q\text{-}1]$ and $A[q\text{+}1…r]$ s.t. each element of $A[p…q\text{-}1]$ is less than or equal to each element of $A[q\text{+}1…r]$. Index q is computed here, called **pivot**.
- *Conquer*: two subarrays are sorted by recursive calls to quicksort.
- *Combine*: unlike merge sort, no work needed since the subarrays are sorted in place already.

4

Quicksort

- The basic algorithm to sort an array A consists of the following four easy steps:
  - If the number of elements in A is 0 or 1, then return
  - Pick any element $\boldsymbol{v}$ in A. This is called the ***pivot***
  - Partition A-{$\boldsymbol{v}$} (the remaining elements in A) into two disjoint groups:
    - $A_1 = \{\boldsymbol{x} \in A\text{-}\{\boldsymbol{v}\} \mid \boldsymbol{x} \leq \boldsymbol{v}\}$, and
    - $A_2 = \{\boldsymbol{x} \in A\text{-}\{\boldsymbol{v}\} \mid \boldsymbol{x} \geq \boldsymbol{v}\}$
  - return
    - {quicksort(A_1) followed by $\boldsymbol{v}$ followed by quicksort(A_2)}

5

Quicksort

- Small instance has $n \leq 1$
  - Every small instance is a sorted instance
- To sort a large instance:
  - select a pivot element from out of the $\boldsymbol{n}$ elements
- Partition the $\boldsymbol{n}$ elements into 3 groups left, middle and right
  - The middle group contains only the pivot element
  - All elements in the left group are ≤ pivot
  - All elements in the right group are ≥ pivot
- Sort left and right groups recursively
- Answer is sorted left group, followed by middle group followed by sorted right group

6

Example

6	2	8	5	11	10	4	1	9	7	3

Use 6 as the pivot

2	5	4	1	3	6	7	9	10	11	8

Sort left and right groups recursively

7

Quicksort Code

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r)
        Quicksort(A, p, q-1)
        Quicksort(A, q+1, r)
    }
}
```

❖ Initial call is **Quicksort**(*A*, 1, *n*), where *n* is the length of *A*

8

Partition

- Clearly, all the action takes place in the **`partition()`** function
  - Rearranges the subarray in place
  - End result:
    - Two subarrays
    - All values in first subarray ≤ all values in second
  - Returns the **index** of the "pivot" element separating the two subarrays

9

Partition Code

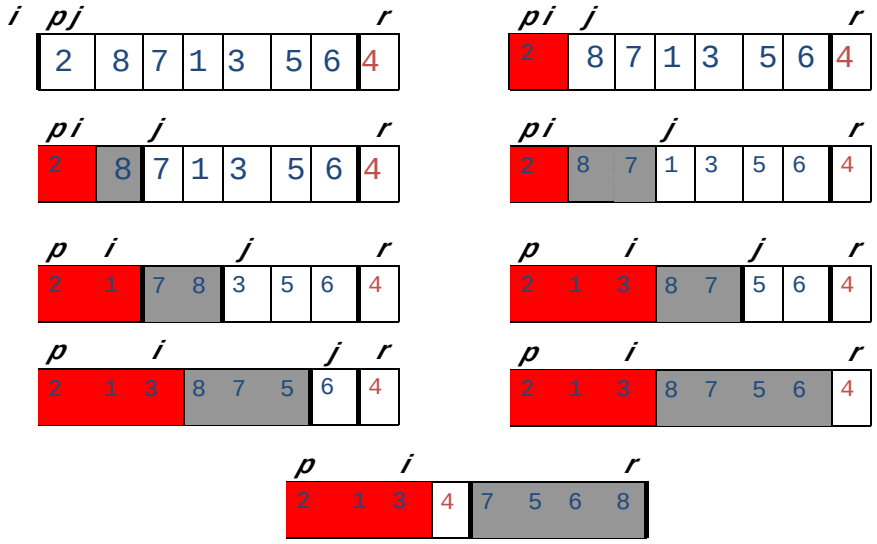
```
Partition(A, p, r)
{
    x = A[r]          // x is pivot
    i = p - 1
     for j = p to r – 1
     {
        do if A[j] <= x
              then
              {
                i = i + 1
                exchange A[i] ↔ A[j]
              }
     }
    exchange A[i+1] ↔ A[r]
    return i+1
}
```

`partition()` *runs in O(n) time*

10

Partition Example

$A = \{2, 8, 7, 1, 3, 5, 6, 4\}$



11

Partition Example Explanation

- Red shaded elements are in the first partition with values $\leq x$ (pivot)
- Gray shaded elements are in the second partition with values $\geq x$ (pivot)
- The unshaded elements have not yet been put in one of the first two partitions
- The final white element is the pivot

12

Choice Of Pivot

- Pivot is **rightmost** element in list that is to be sorted
  - When sorting *A*[6:20], use *A*[20] as the pivot
  - Textbook implementation does this
- **Randomly** select one of the elements to be sorted as the pivot
  - When sorting *A*[6:20], generate a random number *r* in the range [6, 20]
  - Use *A*[*r*] as the pivot

13

Choice Of Pivot

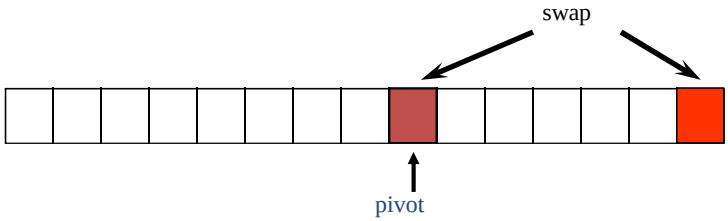
- **Median-of-Three** rule - from the leftmost, middle, and rightmost elements of the list to be sorted, select the one with median key as the pivot
  - When sorting *A*[6:20], examine *A*[6], *A*[13] ((6+20)/2), and *A*[20]
  - Select the element with median (i.e., middle) key
  - If *A*[6].key = 30, *A*[13].key = 2, and *A*[20].key = 10, *A*[20] becomes the pivot
  - If *A*[6].key = 3, *A*[13].key = 2, and *A*[20].key = 10, *A*[6] becomes the pivot

14

Choice Of Pivot

- If *A*[6].key = 30, *A*[13].key = 25, and *A*[20].key = 10, *A*[13] becomes the pivot

❖ When the pivot is picked at random or when the median-of-three rule is used, we can use the quicksort code of the *textbook provided* we first swap the rightmost element and the chosen pivot.



15

Partitioning Into Three Groups

❖ Sort *A* = [6, 2, 8, 5, 11, 10, 4, 1, 9, 7, 3].

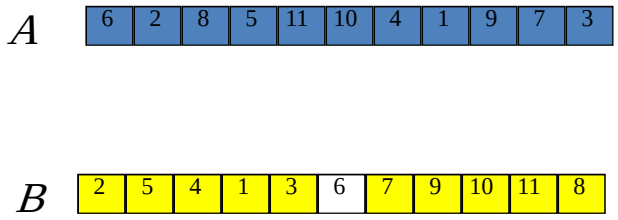
❖ **Leftmost** element (6) is the pivot

❖ When **another array *B*** is available:

- Scan *A* from left to right (omit the pivot in this scan), placing elements ≤ pivot at the left end of *B* and the remaining elements at the right end of *B*
- The pivot is placed at the remaining position of the *B*

16

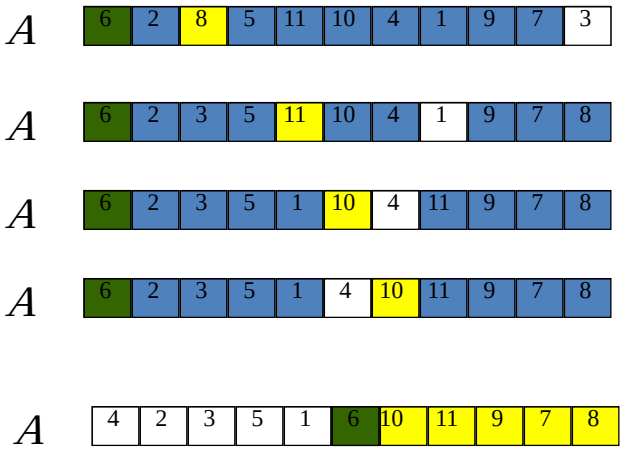
Partitioning Example Using Additional Array



Sort left and right groups recursively.

17

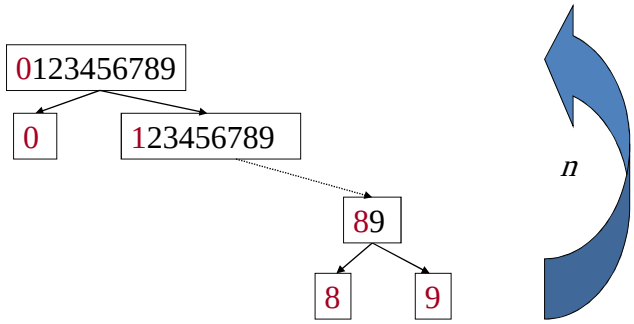
In-Place Partitioning Example



18

Runtime of Quicksort

- Worst case:
  - every time nothing to move
  - pivot = left (right) end of subarray
  - $\Theta(n^2)$



19

Worst Case Partitioning

- The running time of quicksort depends on whether the partitioning is **balanced** or not.
- $\Theta(n)$ time to partition an array of n elements
- Let $T(n)$ be the time needed to sort n elements
- $T(0) = T(1) = c$, where c is a constant
- When $n > 1$,
  - $T(n) = T(|\text{left}|) + T(|\text{right}|) + \Theta(n)$
- $T(n)$ is maximum (worst-case) when either $|\text{left}| = 0$ or $|\text{right}| = 0$ following each partitioning

20

Worst Case Partitioning

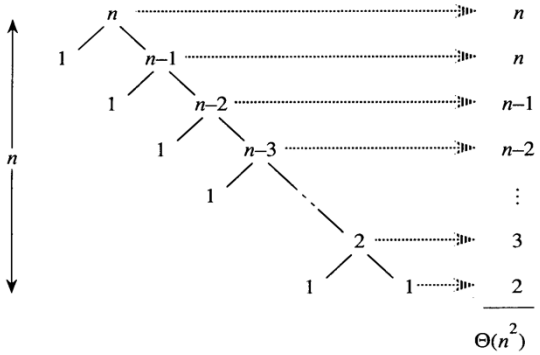


Figure 8.2 A recursion tree for QUICKSORT in which the PARTITION procedure always puts only a single element on one side of the partition (the worst case). The resulting running time is $\Theta(n^2)$.

21

Worst Case Partitioning

- **Worst-Case** Performance (unbalanced):
  - $T(n) = T(1) + T(n\text{-}1) + \Theta(n)$
    - partitioning takes $\Theta(n)$
  - $= 2 + 3 + 4 + \ldots + n\text{-}1 + n + n =$
  - $= \sum_{k = 2 \text{ to } n} \Theta(k) + n = \Theta(\sum_{k = 2 \text{ to } n} k) + n = \Theta(n^2)$

- This occurs when
  - the input is **completely sorted**
- or when
  - the pivot is always the **smallest** (or **largest**) element

Worst-Case Analysis

- $T(n) = \max_{0 \le q \le n-1} (T(q) + T(n - q - 1)) + \Theta(n)$
  - where q ranges from 0 to n-1 since the procedure **PARTITION** produces two sub-problems with *total* size n-1
- **Substitution method**: Guess $T(n) \le cn^2$

$$T(n) \le \max_{0 \le q \le n-1} (cq^2 + c(n - q - 1)^2) + \Theta(n)$$

$$= c \cdot \max_{0 \le q \le n-1} (q^2 + (n - q - 1)^2) + \Theta(n)$$

- Take derivatives to get maximum at $q = 0, n-1$:

$$T(n) \le c(n - 1)^2 + \Theta(n) \le cn^2 - 2c(n - 1) + 1 + \Theta(n) \le cn^2$$

- Therefore, the worst case running time is $\Theta(n^2)$

23

Best Case Partition

- When the partitioning procedure produces two regions of size ***n*/2**, we get the a **balanced** partition with **best case** performance:
  - $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$
- **Average** complexity is also $\Theta(n \lg n)$

24

Best Case Partitioning

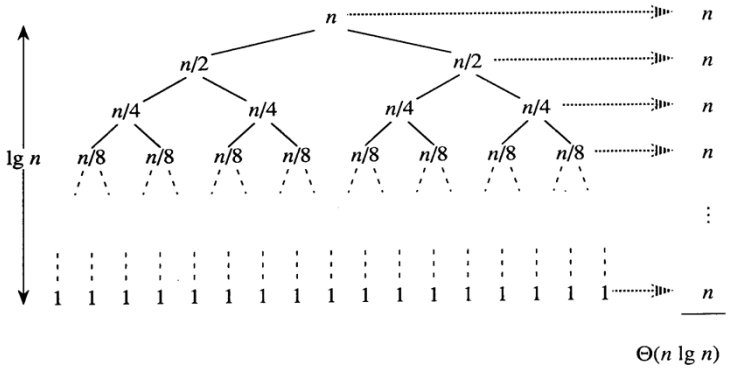


Figure 8.3 A recursion tree for QUICKSORT in which PARTITION always balances the two sides of the partition equally (the best case). The resulting running time is $\Theta(n \lg n)$.

25

Average Case

- Assuming random input, average-case running time is much closer to $\Theta(n \lg n)$ than $\Theta(n^2)$
- First, a more intuitive explanation/example:
  - Suppose that **partition()** always produces a 1-to-9 **proportional** split. This looks quite unbalanced!
  - The recurrence is thus:

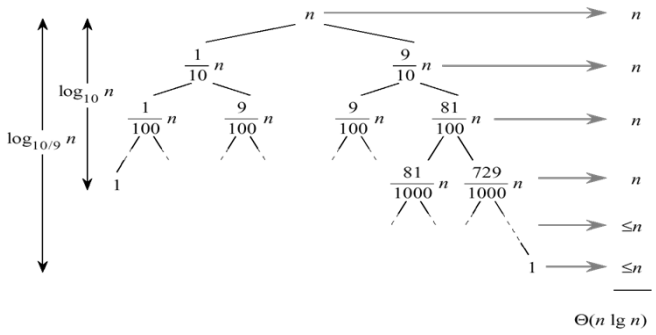
    $T(n) = T(n/10) + T(9n/10) + \Theta(n) = \Theta(n \lg n)$!

  - *How deep will the recursion go?*

26

Average Case

$$T(n) = T(n/10) + T(9n/10) + \Theta(n) = \Theta(n \log n)!$$



For a split of proportionality α, where $0 \le \alpha \le ½$, the minimum depth of the tree is $-\lg n / \lg \alpha$ & the maximum depth is $-\lg n / \lg(1-\alpha)$.
$\log_2 n = \log_{10} n / \log_{10} 2$

27

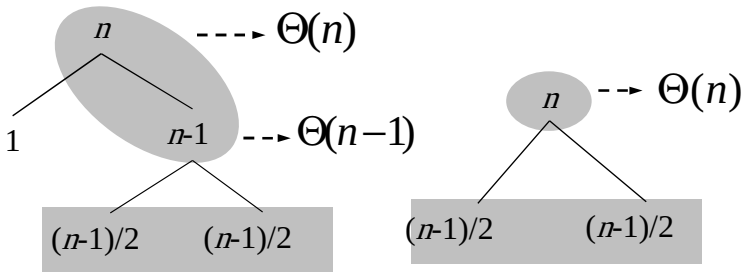
Average Case

- Intuitively, a real-life run of quicksort will produce a mix of "bad" and "good" splits
  - Randomly distributed among the recursion tree
  - Pretend for intuition that they alternate between best-case ($n/2$:$n/2$) and worst-case (n-1:1)

28

Average Case

- What happens if we bad-split root node, then good-split the resulting size (n-1) node?
  - We end up with three subarrays, size
    - 1, (n-1)/2, (n-1)/2
  - Combined cost of splits = $n + n$-1 = $2n$ -1 = $\Theta(n)$
  - No worse than if we had good-split the root node!



Intuition for the Average Case

❖ Suppose, we alternate lucky and unlucky cases to get an average behavior

$$L(n) = 2U(n/2) + \Theta(n) \quad \text{lucky}$$
$$U(n) = L(n-1) + \Theta(n) \quad \text{unlucky}$$

we consequently get

$$L(n) = 2(L(n/2-1) + \Theta(n/2)) + \Theta(n)$$
$$= 2L(n/2-1) + \Theta(n)$$
$$= \Theta(n \log n)$$

The combination of good and bad splits would result in $T(n) = \Theta(n \lg n)$, but with slightly larger constant hidden by the Θ-notation.

30

Randomized Quicksort

- An algorithm is *randomized* if its behavior is determined not only by the input but also by values produced by a *random-number generator*.
- Exchange $A[r]$ with an element chosen at random from $A[p…r]$ in **Partition**.
- This ensures that the pivot element is equally likely to be any of input elements.

31

Randomized Quicksort

Randomized-Partition(*A, p, r*)

1. $i \leftarrow Random(p, r)$
2. exchange $A[r] \leftrightarrow A[i]$
3. **return Partition**(A, p, r)

Randomized-Quicksort(*A, p, r*)

1. **if** $p < r$
2. **then** $q \leftarrow$ **Randomized-Partition**(A, p, r)
3. **Randomized-Quicksort**$(A, p, q\text{-}1)$
4. **Randomized-Quicksort**$(A, q\text{+}1, r)$

32

Review: Analyzing Quicksort

- *What will be the worst case for the algorithm?*
  - Partition is always unbalanced
- *What will be the best case for the algorithm?*
  - Partition is balanced
- *Which is more likely?*
  - The latter, by far, except...
- *Will any particular input elicit the worst case?*
  - Yes: Already-sorted input

33