Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Quick Sort

Dr. Shylaja S S

Quick Sort

Quicksort is another important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input's elements according to their position in the array, quicksort divides them according to their value. Specifically, it rearranges elements of a given array A[0 .. n - 1] to achieve its partition, a situation where all the elements before some position s are smaller than or equal to A[s] and all the elements after positions are greater than or equal to A[s]:

$$\underbrace{A[0]\dots A[s-1]}_{\text{all are } \le A[s]}\quad A[s]\quad \underbrace{A[s+1]\dots A[n-1]}_{\text{all are } \ge A[s]}$$

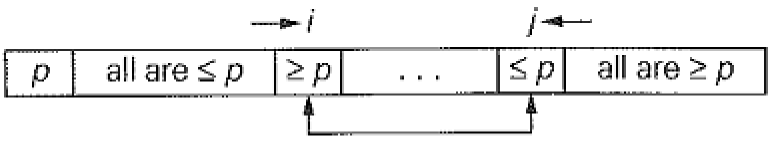
Obviously, after a partition has been achieved, A[s] will be in its final position in the sorted array, and we can continue sorting the two sub arrays of the elements preceding and following A[s] independently (e.g., by the same method).

```
ALGORITHM  Quicksort(A[l .. r])
// Sorts a subarray by quicksort
// Input: A subarray A[l … r] of A[0 .. n  -1], defined by its left and right indices l
//and r
// Output: Subarray A[l .. r]  sorted in non decreasing order
if l  <  r
        s $\leftarrow$Partition(A[l .. r])        //s is a split position
        Quicksort(A[l  .. s - 1])
        Quicksort(A[s + 1 .. r])
```

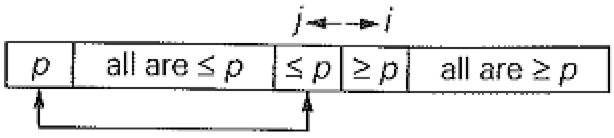
A partition of A [0 ... n- 1] and, more generally, of its subarray A[l .. r] ($0 \le l < r \le n - 1$) can be achieved by the following algorithm. First, we select an element with respect to whose value we are going to divide the subarray. Because of its guiding role, we call this element the pivot. There are several different strategies for selecting a pivot. For now, we use the simplest strategy of selecting the subarray's first element: p = A[l].

There are also several alternative procedures for rearranging elements to achieve a partition. Here we use an efficient method based on two scans of

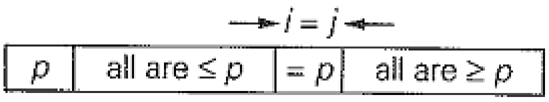
the subarray: one is left-to-right and the other right-to-left, each comparing the sub-array's elements with the pivot. The left-to-right scan, denoted below by index i, starts with the second element. Since we want elements smaller than the pivot to be in the first part of the subarray, this scan skips over elements that are smaller than the pivot and stops on encountering the first element greater than or equal to the pivot. The right-to-left scan, denoted below by index j, starts with the last element of the subarray. Since we want elements larger than the pivot to be in the second part of the sub array, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.
After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed. If scanning indices i and j have not crossed, i.e., i < j, we simply exchange A[i] and A[j] and resume the scans by incrementing i and decrementing j, respectively:



If the scanning indices have crossed over, i.e., i > j, we will have partitioned the array after exchanging the pivot with A[j]:



Finally, if the scanning indices stop while pointing to the same element, i.e., i = j, the value they are pointing to must be equal top (why?). Thus, we have the array partitioned, with the split positions s = i = j:



We can combine the last case with the case of crossed-over indices (i > j) by exchanging the pivot with A[j] whenever i ≥j.

Here is a pseudocode implementing this partitioning procedure.

```
ALGORITHM  Partition(A[l .. r])
// Partitions a subarray by using its first element as a pivot
// Input: A  subarray A[l..r] of A[0 .. n  - 1],  defined by its left and right  indices l
// and r  (l  <  r)
// Output: A partition of A[l .. r ], with the split position returned as this
//function's value
p ← A[l]
i ← l; j ← r + 1
repeat
        repeat i ← i + 1 until A[i] ≥ p
        repeat j ← j  - 1 until A[j] ≤ p
        swap(A[i],  A[j])
until i ≥ j
swap(A[i],  A[j])      //undo last swap when i ≥ j
swap(A[i],  A[j])
return  j
```

Note that index i can go out of the subarray bounds in this pseudocode. Rather than checking for this possibility every time index i is incremented, we can append to array A[0 .. n- 1] a "sentinel" that would prevent index i from advancing beyond position n. The more sophisticated method of pivot selection mentioned at the end of the section makes such a sentinel unnecessary.

An example of sorting an array by quicksort is given in Fig. 1.

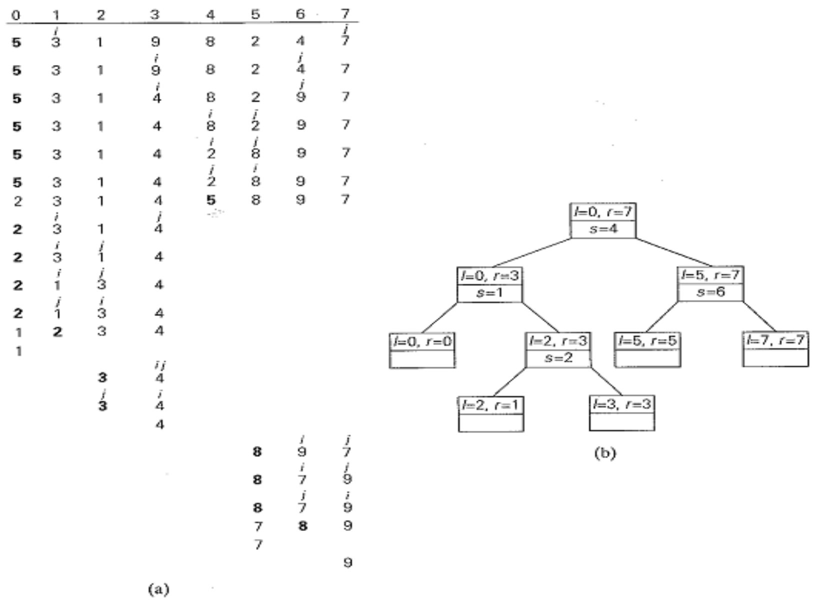


Fig. 1: Example of Quicksort operation. (a) The array's transformations with pivots shown in bold. (b) The tree of recursive calls to Quicksort with input values l and r of subarray bounds and split positions of a partition obtained.

Quick sort Analysis:

Best Case: The number of comparisons in the best case satisfies the recurrence:

$C_{best}(n) = 2C_{best}(n/2) + n$ for $n > 1$, $C_{best}(1) = 0$

According to Master Theorem

$$C_{best}(n) \in \Theta(n \log_2 n)$$

Worst Case: The number of comparisons in the worst case satisfies the recurrence:

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \theta(n^2)$$

Average Case: Let $C_{avg}(n)$ be the number of key comparisons made by Quick Sort on a randomly ordered array of size n.

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \; for \; n > 1$$

The solution for the above recurrence is:

$$C_{avg}(n) \approx 2n \ln n \approx 1.38n \log_2 n$$