Chapter 7: Quicksort

Introduction

- Quicksort is based on divide-and-conquer.
  For an array A[p..r]:

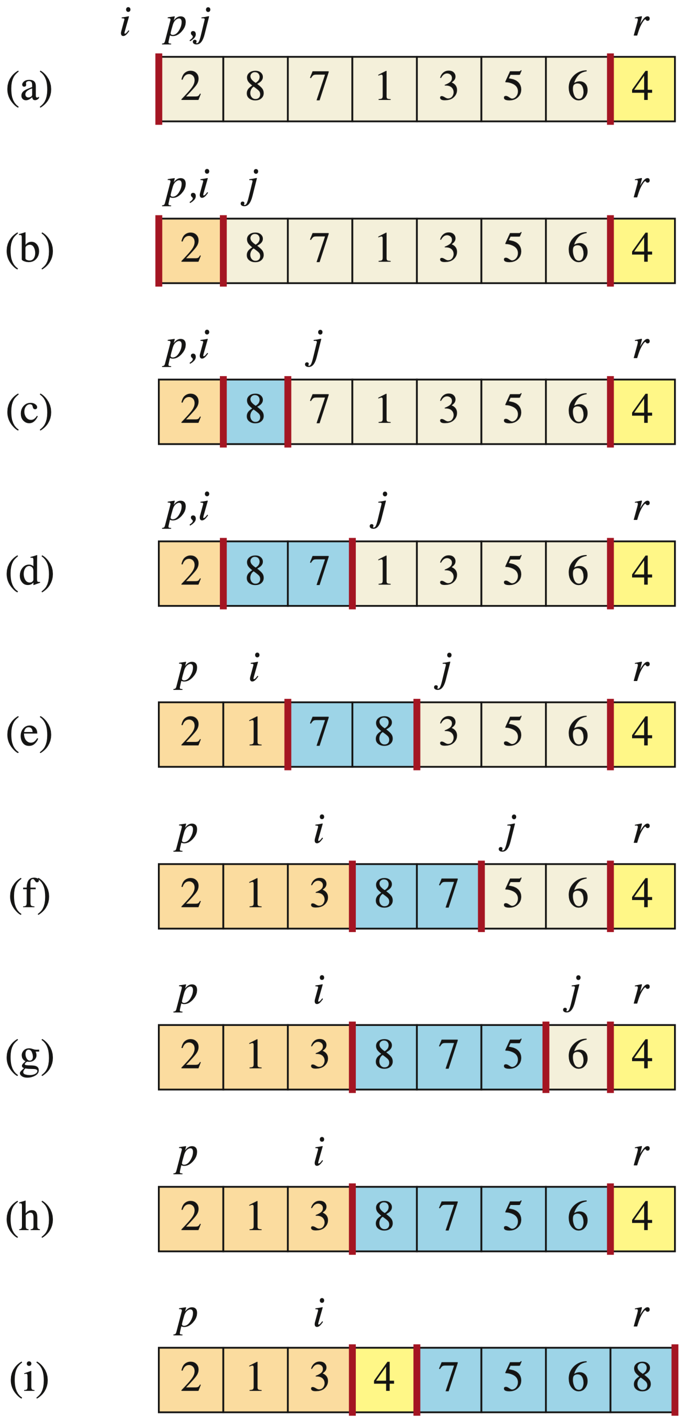
Divide:	Using a pivot x, partition the array A[p..r] into two subarrays A[p..q-1] and A[q+1..r] (one of them could be empty) such that each element of $A[p..q-1] \leq x$ and each element of $A[q+1..r] \geq x$ and $A[q] = x$ Index q will be computed and returned by this partition procedure (After partitioning, the element x is in its correct position)
Conquer:	Sort A[p..q-1] and A[q+1..r] recursively.
Combine:	No action for combine.

Note that here the effort is in dividing the problem (unlike mergesort where dividing was simple but combining was more difficult).

```
QUICKSORT(A, p, r)
1. if p < r
2.    // partition the array around the pivot which ends up at A[q]
2.    q = PARTITION(A, p, r)
3.    QUICKSORT(A, p, q - 1) // recursively sort the low side
4.    QUICKSORT(A, q + 1, r) // recursively sort the high side

PARTITION(A, p, r)
1. x = A[r] // the pivot
2. i = p - 1 //highest index on the low side
3. for j = p to r - 1  //process each element other than the pivot
4.     if A[j] <= x    // does this element belong to the low side
5.          i = i + 1
6.          exchange A[i] and A[j]
7. exchange A[i+1] and A[r] //just to right of the low side
8. return i + 1 // the new index of the pivot
```

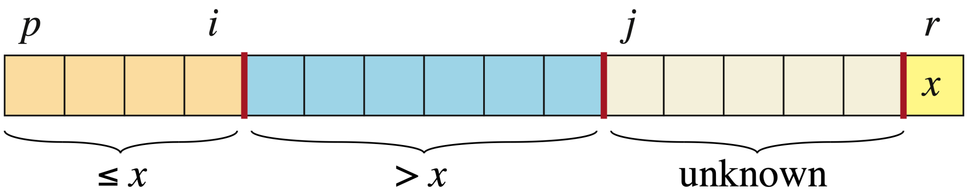
An example:


(a)
i p,j r
2 8 7 1 3 5 6 4
(b)
p,i j r
2 8 7 1 3 5 6 4
(c)
p,i j r
2 8 7 1 3 5 6 4
(d)
p,i j r
2 8 7 1 3 5 6 4
(e)
p i j r
2 1 7 8 3 5 6 4
(f)
p i j r
2 1 3 8 7 5 6 4
(g)
p i j r
2 1 3 8 7 5 6 4
(h)
p i r
2 1 3 8 7 5 6 4
(i)
p i r
2 1 3 4 7 5 6 8

Loop invariant: During the execution of the PARTITION procedure,

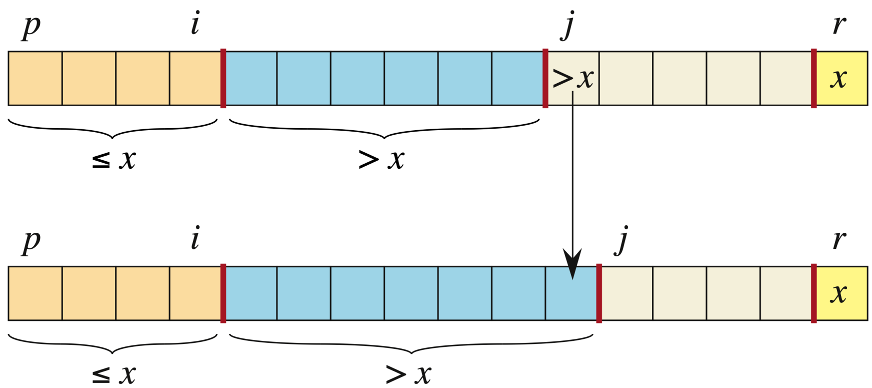
1. Elements in the array before index i are less than or equal to the pivot x. That is,
   $A[k] \leq x \quad \text{if } p \leq k \leq i$
2. Elements in the array between the index $i+1$ and $j-1$ are larger than the pivot x. That is,
   $A[k] > x \quad \text{if } i+1 \leq k \leq j-1$
3. Elements in the array after index j are not yet compared.

The loop invariant can be illustrated by the following picture:

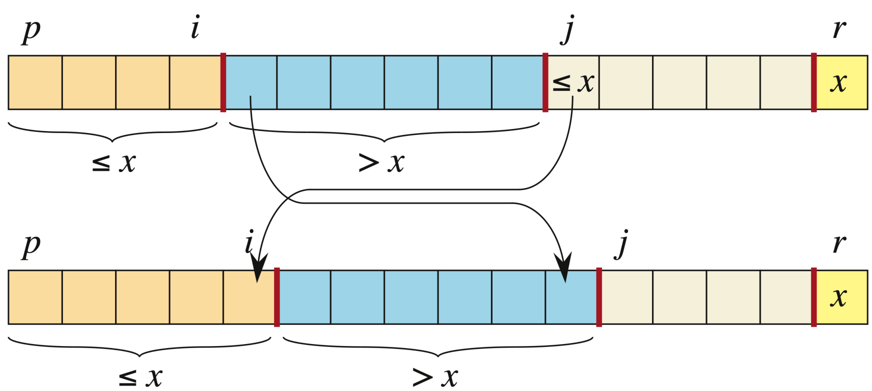


The following diagram shows the two cases to consider in the loop:

(a)



(b)



Recommended Exercises: Ex. 7.1-1, 7.1-2, 7.1-3, 7.1-4.

- EX 7.1-1: Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$.

- EX 7.1-2: What value of q does PARTITION return when all elements of the array $A[p..r]$ have the same value?
- EX 7.1-3: Give a brief argument that the running time of PARTITION on a subarray of length n is $\Theta(n)$ in the worst case.
- EX 7.1-4: Modify QUICKSORT to sort into mononically decreasing order.

Performance of Quicksort

Depending on whether the partition is "balanced" or not
If not balanced: asymptotically as slow as selection sort: $\Theta(n^2)$.

- Worst case partition:
  This case occurs when partition produces one subarray with $n-1$ elements.

  If this worst case partitioning occurs at each recursive step of the algorithm, then Quicksort takes $T(n) = \Theta(n^2)$ since there are n partitions altogether with n, $n-1$, $\ldots, 1$ elements.

  Thus, the **worst case** running time of Quicksort is $T(n) = n + (n-1) + \ldots + 1 = \Theta(n^2)$.

  For example, if the input array is already sorted in either order.
- Best case partition:
  This case occurs when partition produces two subarrays with $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil - 1$ elements. If this best case partitioning occurs at each recursive step of the algorithm, then Quicksort requires the following run time (we are simplifying $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil - 1$ to n/2 each):

  $$T(n) = 2T(n/2) + \Theta(n)$$

  This is the same as merge sort, so we have $T(n) = \Theta(n \lg n)$ (which we analyzed earlier using a recurrence tree).

  Let us also calculate the run-time directly using substitution.

  There is 1 partition with n elements, 2 partitions with $\frac{n}{2}$ elements, 4 partitions with $\frac{n}{4}$ elements, $\ldots$, n partitions with $\frac{n}{n}$ elements. Thus, the **best case** running time of Quicksort is

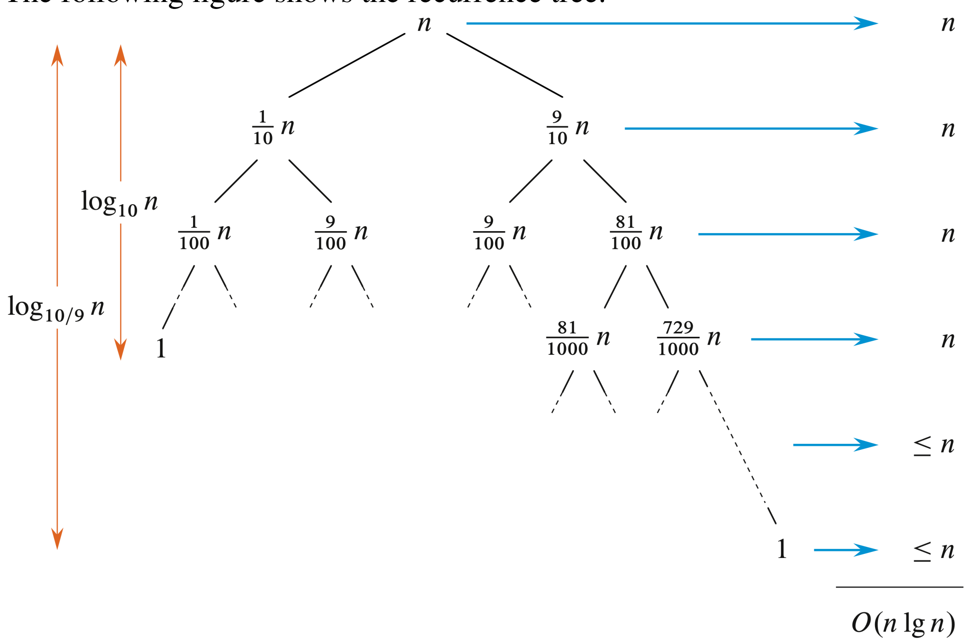
  $$\begin{aligned} T(n) &= 1 \cdot \tfrac{n}{1} + 2 \cdot \tfrac{n}{2} + \ldots + n \cdot \tfrac{n}{n} \\ &= 2^0 \cdot \tfrac{n}{2^0} + 2^1 \cdot \tfrac{n}{2^1} + \ldots + 2^{\log n} \cdot \tfrac{n}{2^{\log n}} \\ &= n + n + \ldots + n \quad (n \text{ adds itself } \log n \text{ times}) \\ &= n \log n \end{aligned}$$
- **Optional**: Average case running time of Quicksort:

The **average case** running time of Quicksort is $\Theta(n \log n)$. The average case running time analysis of quicksort will be discussed in more detail in CS 421.

We will provide some intuition with an example of an uneven split. Suppose partition always splits the given subarray into a 9-1 proportional split. For n elements, one side is $9n/10$ and the other side is $n/10$. That gives us a run time:

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

The following figure shows the recurrence tree.



Any split of constant proportionality results in a running time of $O(n \lg n)$.

Recommended Exercises: Ex 7.2-1, 7.2-2, 7.2-3, 7.2-4.

* EX 7.2-1: Use the substitution method to show that the recurrence $T(n) = T(n-1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$.
* EX 7.2-2: Show that the worst-case running time of Quicksort is $\Theta(n^2)$ when all elements of the array are the same value.
* EX 7.2-3: Show that the running time of Quicksort is $\Theta(n^2)$ when the array A contains n distinct values that are sorted in decreasig order.
* EX. 7.2-4: Banks often record transactions on an account in the order of the times of the transactions, but many people like to receive their bak statements with checks listed in order of the check number. People usually write checks in order of increasing check number, and mercchans usually cash them with reasonabe dispatch. The problem of converting time-of-transaction order to check-number order is a problem of soting almost sorted input. Explain persuasively why the procedure INSERTION-SORT is likely to beat QUICKSORT on this problem. (Hint: How long does it take to sort an array that is already sorted except for k elements? What is the value of k in this problem?)

Randomized Versions of Quicksort

For average-case analysis, we assume that all permutations of the input numbers are equally likely. However, this assumption is not realistic because:

- Two special inputs impose the worst-case split: sorted array in either order.
- These two inputs are very common for lots of applications.

We need to randomize the input array to reduce the probability of the worst-case. Two approaches to randomize the input array.

1. Before feeding the input to Quicksort algorithm, the input is randomly permuted.
2. Randomly choose the pivot element at each step in Quicksort.

The pseudocode for the 2nd approach.

```
RANDOMIZED-PARTITION(A, p, r)
i = Random(p, r)
exchange A[r] and A[i]
return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
   q = RANDOMIZED-PARTITION(A, p, r)
   RANDOMIZED-QUICKSORT(A, p, q-1)
   RANDOMIZED-QUICKSORT(A, q+1, r)
```

Recommended Exercises

- Interesting **bolts and nuts problem:**
  There are *n* pairs of bolts and nuts mixed together. Each bolt has one, and only one, matching nut. The constraint is that a nut can only be compared to a bolt (too big, too small, or match), not to another nut or bolt. Bolts canot be compared with bolts or nutes either. Please suggest an efficient algorithm to match all bolts and nuts.