Quick Sort

Recall Merge Sort

Short comings of Merge sort?

Merge Sort - Shortcomings

- Merging L and R arrays create a new array
- No obvious way to efficiently merge *in place*
- Extra storage is required to merge and can be costly
- Merging happens because elements in left half might have to move right and vice versa

Motivation – Another Sorting algorithm / Quick Sort

- Can we divide so that everything to the left is smaller than everything to the right

- No need to Merge

Quick Sort Algorithm

Quick Sort - Introduction

- Tony Hoare - Early 1960's
- Well Known Computer Scientist
- Turing Award Winner

Quick sort - Idea

- Choose a pivot element
- Typically the first/last value in the array is pivot
- Divide/Partition the array A into lower and upper parts w.r.t pivot
- Move pivot between lower and upper partition
- Recursively sort the two partitions

Quick Sort : Divide and Conquer

- **Divide :**
  - Partition the array A[p..r] to two sub arrays A[p..q-1] and A[q+1..r], where q is computed as part of the *PARTITION* function
  - Each element of A[p..q-1] is less than or equal to A[q]
  - Each element of A[q+1..r] is greater than A[q]
  - The sub arrays can be empty or non-empty

Quick Sort : Divide and Conquer

- **Conquer :**
  - Sort the two sub arrays A[p..q-1] and A[q+1..r]
  - By recursive calls to quick sort

Quick Sort - Divide and Conquer

- **Combine :**
  - The two sub arrays are already sorted
  - The entire array A[p ..r] is already sorted
  - Nothing particular to do in *combine* step

Pseudocode of Quick Sort

QUICKSORT(A, p, r)

```
if p < r
    then q <— PARTITION(A, p, r)
         QUICKSORT(A, p, q-1)
         QUICKSORT(A, q+1, r)
```

To sort an entire array the initial call is

QUICKSORT(A, 1, A.length)

How do we partition an array ?

- Suppose array A = [9, 7, 5, 11, 12, 2, 14, 3, 10, 6]

- Array entry A[r] is selected as the pivot element, where r is the index of the last element of the array

- A[r] is compared with the array elements until a smaller element *s* (less than or equal to pivot) is obtained

- If *s* is obtained, it is exchanged with the first element of the array initially

- A[r] is again compared with all other elements of A and exchange of smaller element is done further

Working of *PARTITION*

- A = [9, 7, 5, 11, 12, 2, 14, 3, 10, 6]
- Pivot = A[r] = 6
- 6 is compared with the elements of A until a smaller element (than 6) is obtained
- In this case, 6 is compared with 9,7 and 5
- Exchange 9 with 5
- A = [5, 7, 9, 11, 12, 2, 14, 3, 10, 6]

Working of *PARTITION*

- A = [5, 7, 9, 11, 12, 2, 14, 3, 10, 6]
- Again 6 is compared with 11,12 and 2
- 7 and 2 are exchanged
- A= [5, 2, 9, 11, 12, 7, 14, 3, 10, 6]
- Again 6 is compared with 14 and 3
- 9 and 3 are exchanged
- A= [5, 2, 3, 11, 12, 7, 14, 9, 10, 6]

Working of *PARTITION*

- A= [5, 2, 3, 11, 12, 7, 14, 9, 10, 6]

- Again 6 is compared with 10

- We can see that 5, 2 and 3 are lesser or equal to than the pivot and we have compared pivot with all other elements of A

- Exchange pivot with 11 so that all the elements before pivot is less than or equal to pivot and all the elements after pivot is greater than pivot

- A= [5, 2, 3, 6, 12, 7, 14, 9, 10, 11]

- [5, 2, 3] and [12, 7, 14, 9, 10, 11] are two partitions separated by the pivot 6

Detailed working of *PARTITION*

p r
9, 7, 5, 11, 12, 2, 14, 3, 10, 6

p r
9, 7, 5, 11, 12, 2, 14, 3, 10, 6
i j

p r
9, 7, 5, 11, 12, 2, 14, 3, 10, 6
i j

p r
9, 7, 5, 11, 12, 2, 14, 3, 10, 6
i j

p r

9, 7, 5, 11, 12, 2, 14, 3, 10, 6

i j

r

5, 7, 9, 11, 12, 2, 14, 3, 10, 6

i j

r

5, 7, 9, 11, 12, 2, 14, 3, 10, 6

i j

r

5, 7, 9, 11, 12, 2, 14, 3, 10, 6

i j

r

5, 7, 9, 11, 12, 2, 14, 3, 10, 6

i j

r

5, 7, 9, 11, 12, 2, 14, 3, 10, 6

i j

r

5, 2, 9, 11, 12, 7, 14, 3, 10, 6

i j

r

5, 2, 9, 11, 12, 7, 14, 3, 10, 6

i j

r

5, 2, 9, 11, 12, 7, 14, 3, 10, 6

i j

```
                               r
5 ,2 , 9, 11, 12, 7, 14, 3, 10, 6
       i                 j

                               r
5 ,2 , 3, 11, 12, 7, 14, 9, 10, 6
       i                 j

                               r
5 ,2 , 3, 11, 12, 7, 14, 9, 10, 6
       i                     j

                              r
5 ,2 , 3,  6 , 12, 7, 14, 9, 10,11
       i                     j

5 ,2 , 3,  6 , 12,7 , 14, 9, 10,11
```

5 ,2 ,3 , 6 , 12,7 , 14, 9, 10,11

- [5, 2, 3] and [12, 7, 14, 9, 10, 11] are two partitions separated by the pivot 6
- All the elements before pivot is less than or equal to pivot and all the elements after pivot is greater than pivot

Design of *PARTITION*

- How many counters/pointers do we need?
  - One counter which goes from p to r-1 : to compare all other elements of A with pivot
  - Another counter which keeps track of the position of the smaller element (less than or equal to pivot)

Pseudo code : *PARTITION*

```
PARTITION (A, p, r)
x = A[r]
i= p - 1
for j = p to r - 1
      do if A[j] <= x
            then  i=i+1
            Exchange A[i] with A[j]
Exchange A[i +1] with A[r]
return i +1
```

Pseudocode of Quick Sort

```
QUICKSORT(A, p, r)

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

Exercise

- Trace the working of *PARTITION* with the sub array [5,2,3]

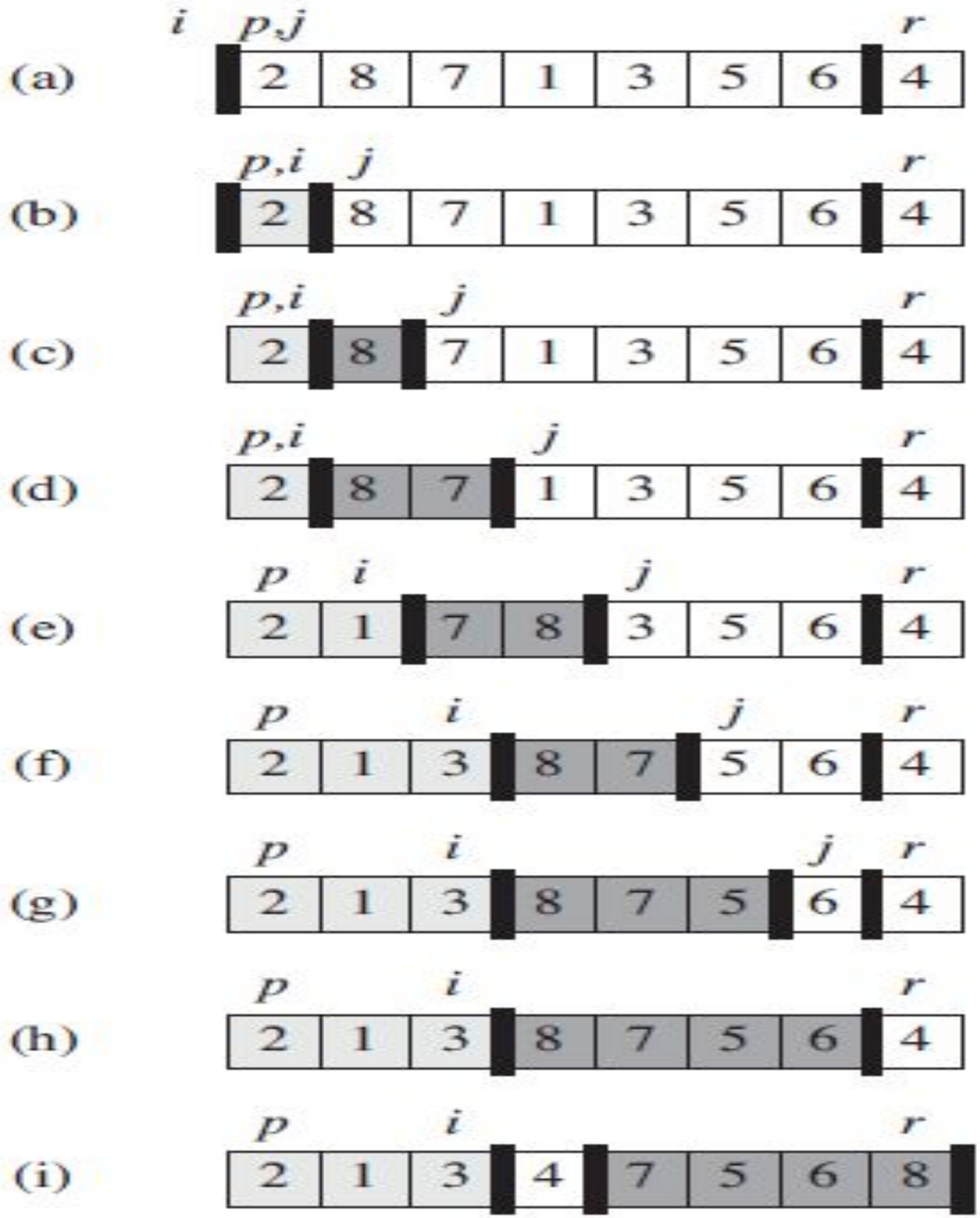
- Trace the working of *PARTITION* with the sub array [12, 7, 14, 9, 10, 11]

- Trace the working of Quick sort with the array [2 4 6 7 9 23]

- Trace the working of Quick sort with the array [26 24 15 7 3 2]

Correctness of *PARTITION*

- PARTITION selects an element x = A[r] as a ***pivot*** element around which to partition the subarray A[p .. r]
- As the procedure runs, it partitions the array into **four** (possibly empty) regions
- Suppose A = [2,8,7,1, 3,5,6,4]



(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

Four regions

- 1^{st} region : Values no greater than pivot (lightly shaded array elements in previous fig.)
- 2^{nd} region : Values greater than pivot (heavily shaded array elements)
- 3^{rd} region: Values not processed for both 1^{st} 2^{nd} region (un shaded elements)
- 4^{th} region: pivot element itself

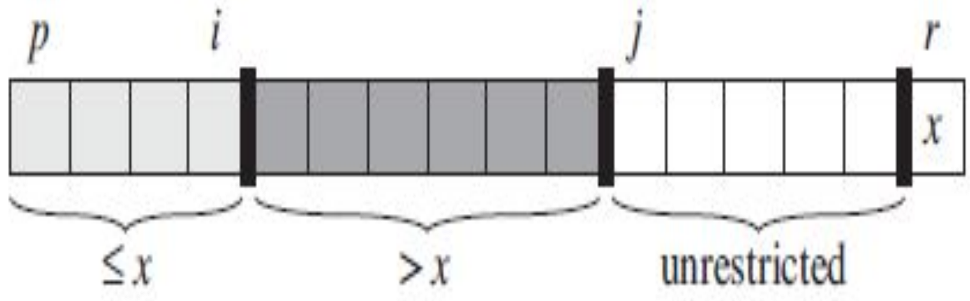


Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p \mathinner{..} r]$. The values in $A[p \mathinner{..} i]$ are all less than or equal to x, the values in $A[i+1 \mathinner{..} j-1]$ are all greater than x, and $A[r] = x$. The subarray $A[j \mathinner{..} r-1]$ can take on any values.

Loop invariant

- At the start of each iteration of *for* loop in lines 3- 6 , the four regions satisfy certain properties:
- We state those as **loop invariant** :

- At the beginning of each iteration of the loop (lines 3- 6), for any array index *k* :
  - **If *p<= k <= i , then A[k] <= x***
  - **If *i +1 <= k <= j-1, then A[k] >x***
  - **If *k=r, then A[k] = x***

```
PARTITION (A, p, r)
1 x= A[r]
2 i = p - 1
3 for j = p to r - 1
4       if A[j] <= x
5             i=i+1
6       exchange A[i] with A[j]
7 exchange A[i +1] with A[r]
8 return i +1
```

Observation

- The indices between *j* and *r*-1 are not covered by any of the above three cases
- The values in these entries have no particular relationship to the pivot *x*

Correctness using loop invariant

- Loop invariant:
- At the beginning of each iteration of the loop (lines 3- 6), for any array index k :
  - **If** ***$p <= k <= i$, then $A[k] <= x$***
  - **If** ***$i + 1 <= k <= j-1$, then $A[k] > x$***
  - **If** ***$k=r$, then $A[k] = x$***
- We need to show that:
  - The loop invariant is true prior to the first iteration
  - Each iteration of the loop maintains the invariant
  - Invariant provides a useful property to show the correctness when the loop terminates

Initialization

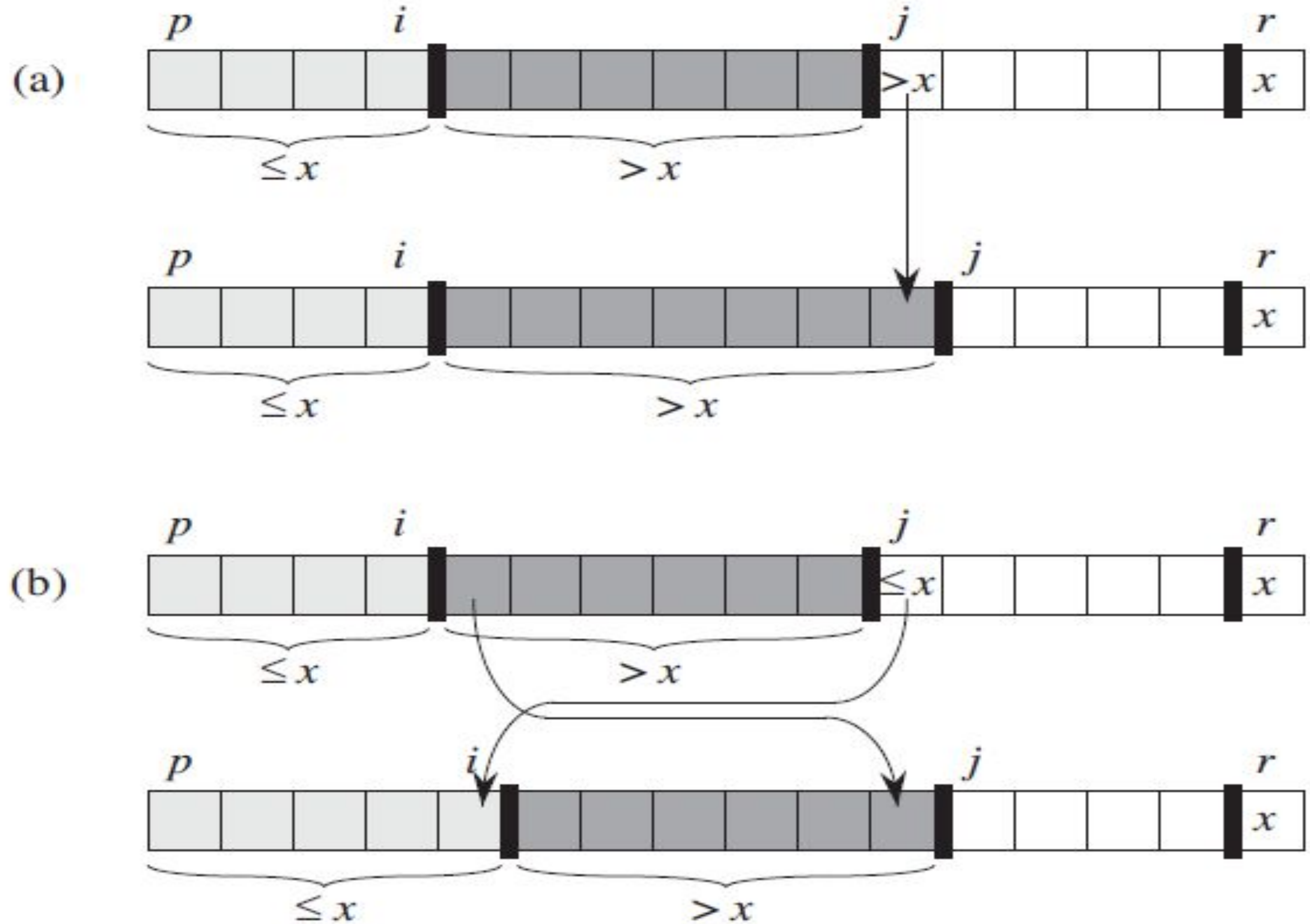
- Prior to the first iteration of the loop:
  - $i=p-1$ and $j=p$
  - No values in between p and i
  - No values in between $i+1$ and $j-1$
  - The first two conditions of loop invariant :

    If *p<= k <= i , then A[k] <= x* *and* **If *i +1 <= k <= j-1, then A[k] >x*** are satisfied trivially
  - The third condition of loop invariant: **If *k=r, then A[k] = x*** is satisfied with the assignment in line 1

    (x= A[r]) of the pseudo code

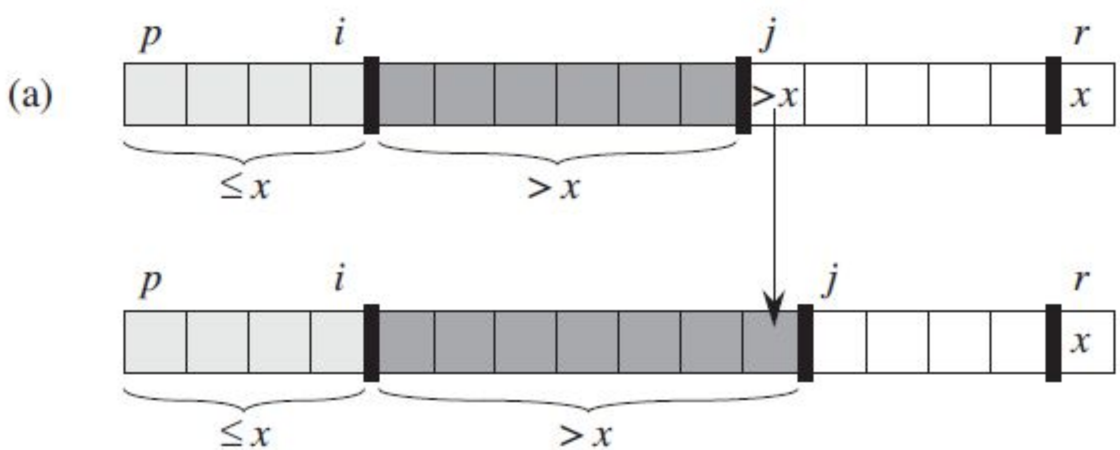
Maintenance

- We consider two cases: (a) *A[j] >x* (b) *A[j] <= x*



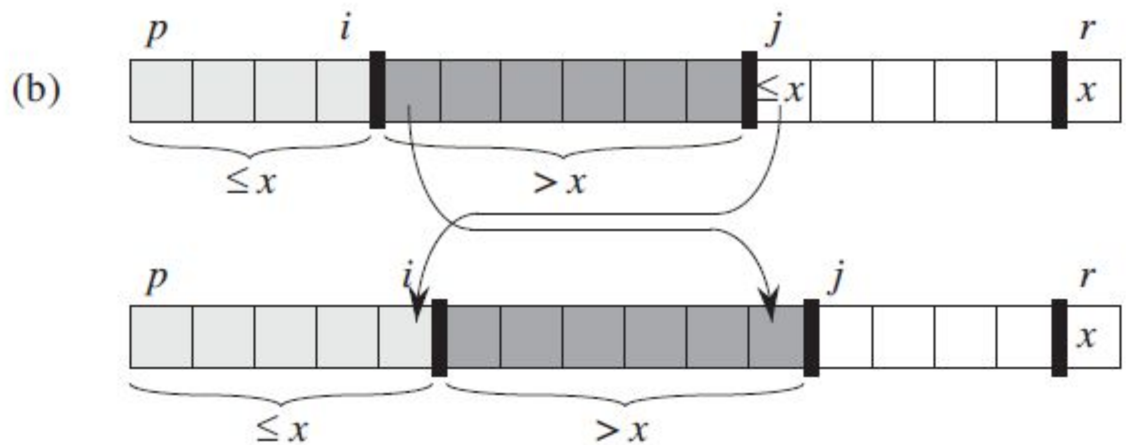
Maintenance : case (a) *A[j] >x*

- Only action in the loop is to increment *j*
- After *j* is incremented, the condition 2 of the loop invariant, **If *i +1 <= k <= j-1, then A[k] >x*** holds for *A[j-1]* and all other entries remain unchanged.
- Hence, the loop invariant holds



Maintenance : (b) *A[j] <= x*

- The loop increments *i* , Swaps *A[i]* and *A[j]* & Increments *j*
- Because of swap, *A[i]<=x and* condition 1 of loop invariant **If *p<= k <= i , then A[k] <= x*** is satisfied
- *A[j-1] >x,* the condition 2 of the loop invariant, **If *i +1 <= k <= j-1, then A[k] >x*** holds for A[j-1]



Termination

- At termination $j=r$.
- Every element is in one of the 3 sets described by the loop invariant
- We have partitioned the array into these three sets: those less than or equal to x, those greater than x and those equal to x
- Last two lines of *PARTITION* moves the pivot to its correct place in A and returns the index

Thank You