Quick Sort

Kuan-Yu Chen (陳冠宇)

2019/03/27 @ TR-310-1, NTUST

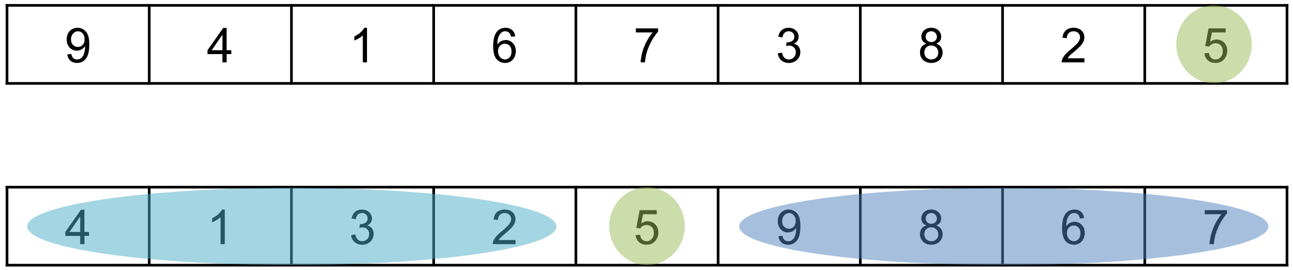
Review

- A **binary heap** is a complete binary tree in which every node satisfies the heap property
  - Min Heap
  - Max Heap
    - MAX-HEAPIFY takes time $O(\log_2 n)$
    - BUILD-MAX-HEAP takes time $O(n)$
    - HEAPSORT takes time $O(n \log_2 n)$

Quick Sort.

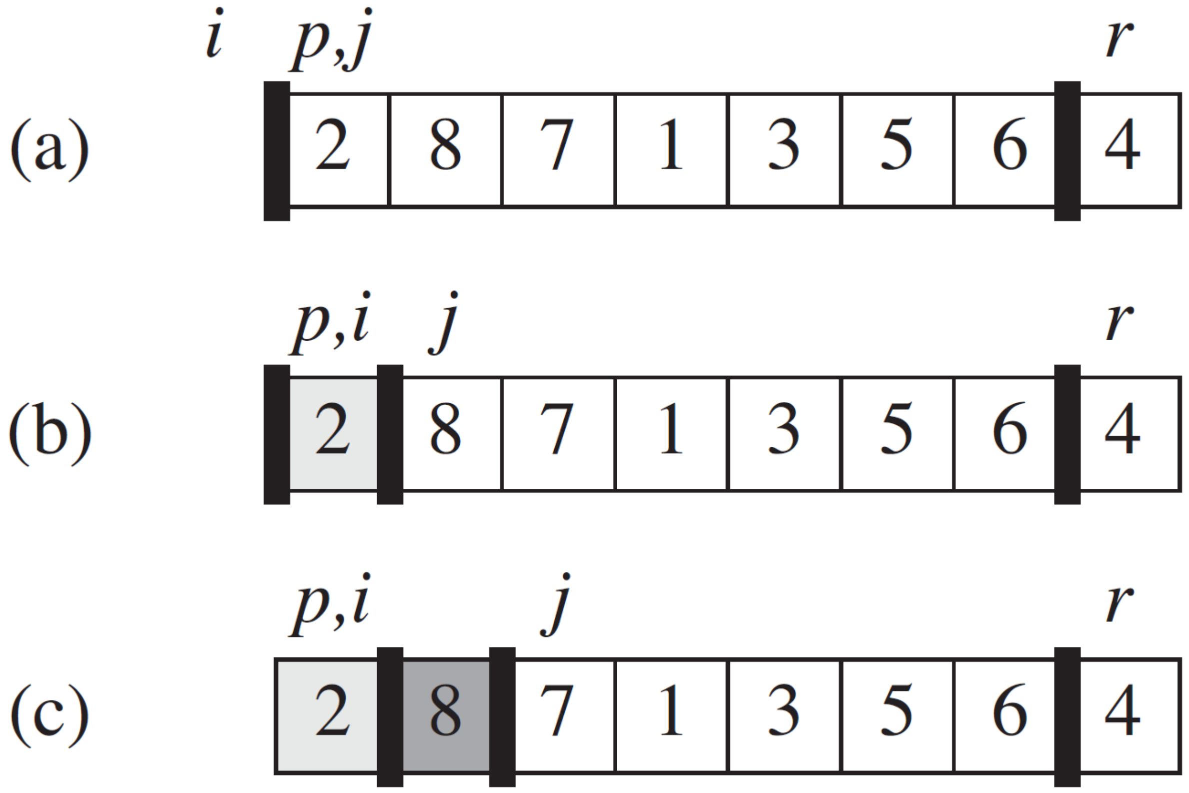
- Quick sort is a widely used sorting algorithm developed by C. A. R. Hoare
  - Quick sort is also known as partition exchange sort

- The quick sort algorithm works as follows:
  1. Select an element **pivot** from the array elements
  2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it
  3. Recursively sort the two sub-arrays thus obtained

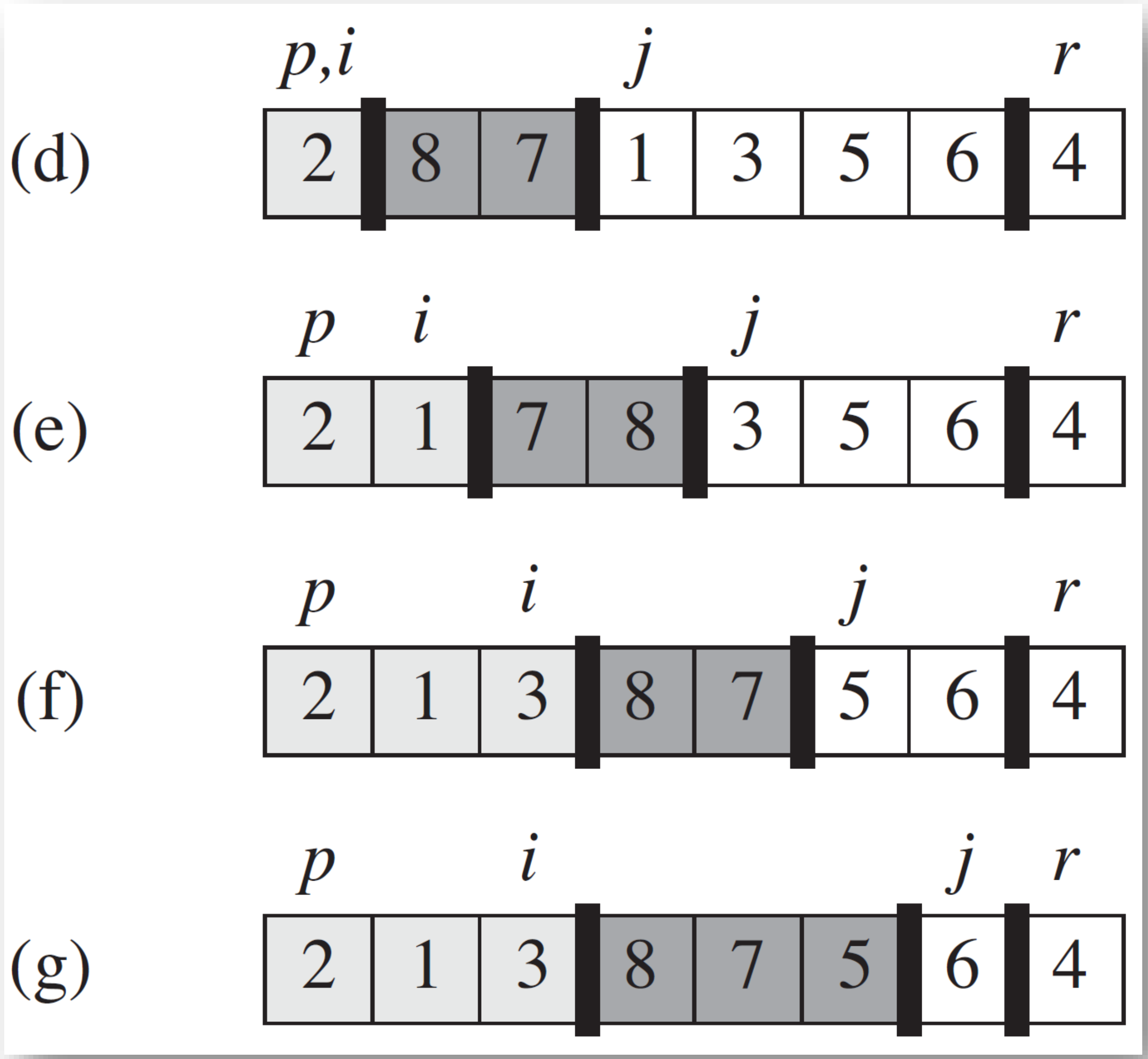


Example.

- Sort the given array using quick sort algorithm

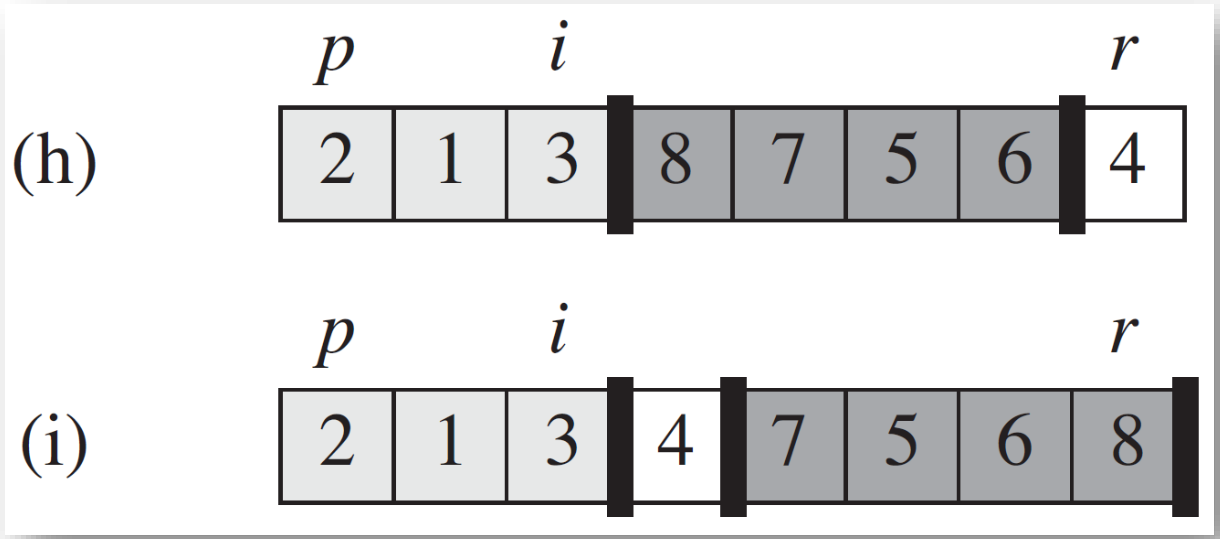


Example..

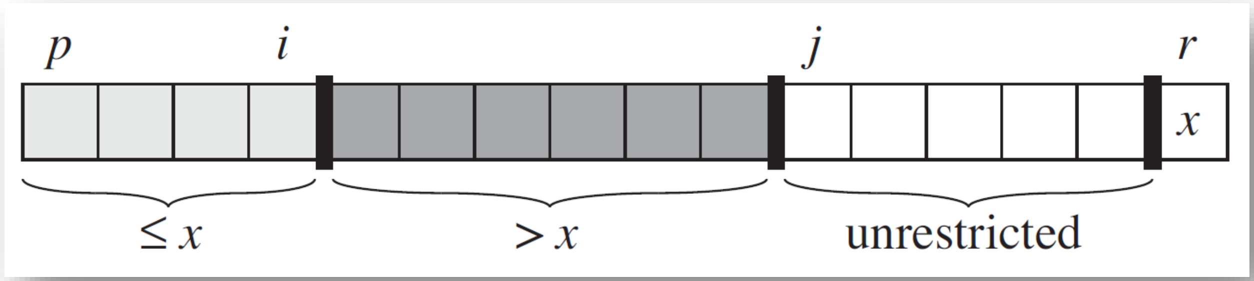


(d)
p,i
j
r
2 8 7 1 3 5 6 4
(e)
p
i
j
r
2 1 7 8 3 5 6 4
(f)
p
i
j
r
2 1 3 8 7 5 6 4
(g)
p
i
j
r
2 1 3 8 7 5 6 4

Example…

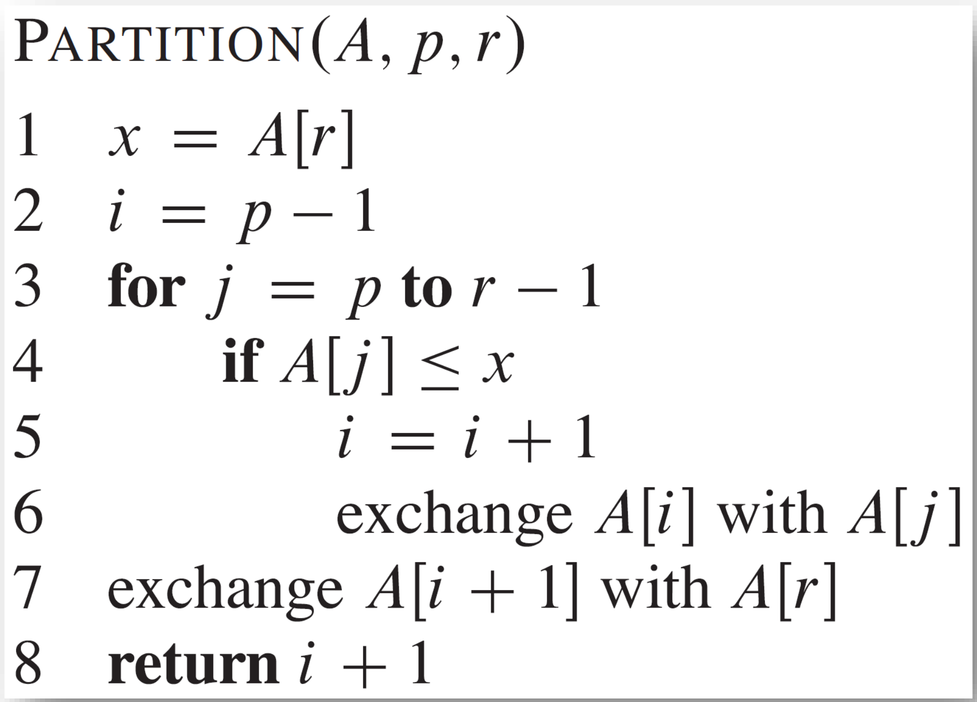


- During the procedure, four regions maintained by the PARTITION function on a subarray

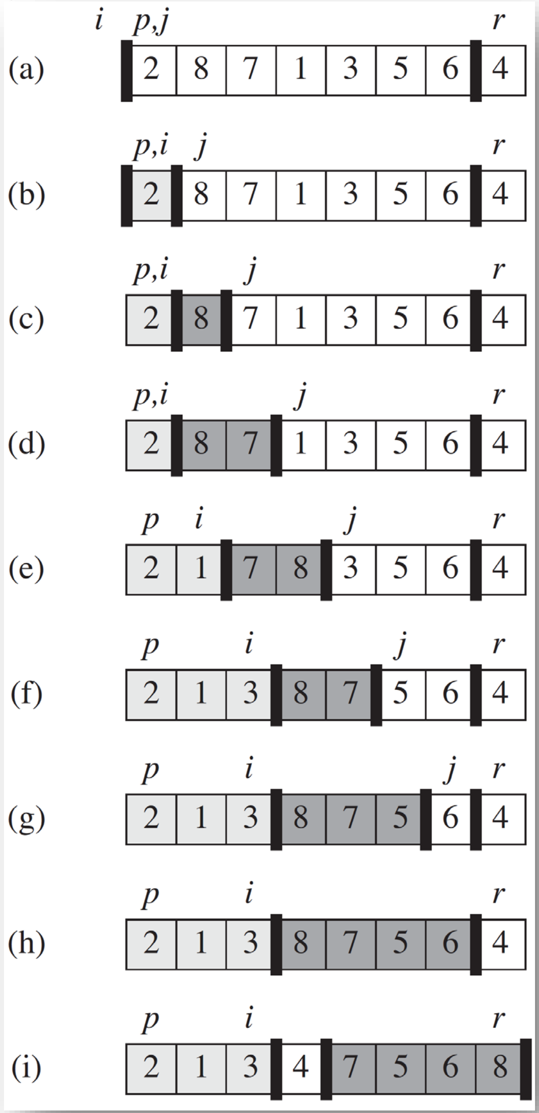


Quick Sort…

- The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p \,..\, r]$ in place

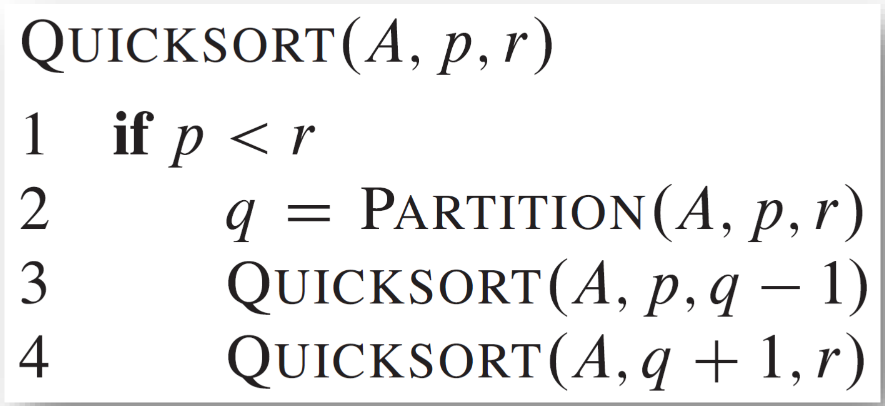


```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



(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

Quick Sort….



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

- Quicksort, like merge sort, applies the divide-and-conquer paradigm
  - **Divide:** Partition (rearrange) the array into two (possibly empty) subarrays
  - **Conquer:** Sort the two subarrays by recursive calls to quicksort
  - **Combine:** All the subarrays are already sorted, no work is needed to combine them

Analyses.

- The running time of quicksort depends on whether the partitioning is balanced or unbalanced
  - Worst-case partitioning
    - The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with $n-1$ elements and one with 0 elements
    - The partitioning costs $\Theta(n)$ time
    - Thus, $T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$
    - By using substitution method, it is easy to get that $T(n) = \Theta(n^2)$
  - Proof the worst-case (for Big-O only!)
    - By the substitution method, we guess $T(n) \le cn^2$

$$\begin{aligned} T(n) &= \max_{0 \le q \le n-1} \left(T(q) + T(n-q-1)\right) + \Theta(n) \\ &\le \max_{0 \le q \le n-1} \left(cq^2 + c(n-q-1)^2\right) + \Theta(n) \\ &= c \max_{0 \le q \le n-1} \left(q^2 + (n-q-1)^2\right) + \Theta(n) \\ &\le c(n-1)^2 + \Theta(n) = O(n^2) \end{aligned}$$

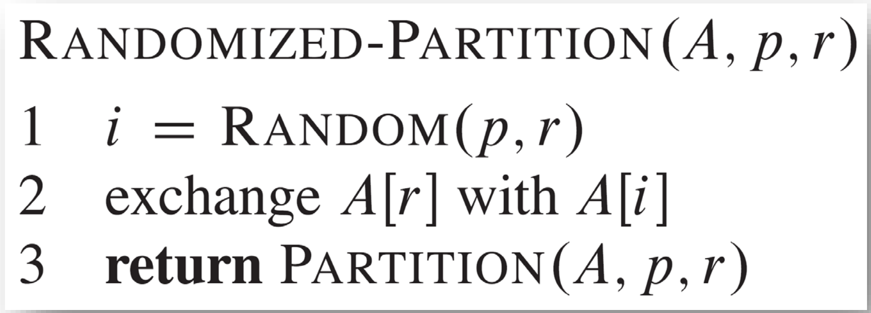
$$\begin{aligned} &a + b = c \\ &\Longrightarrow (a+b)^2 = c^2 \\ &\Longrightarrow a^2 + 2ab + b^2 = c^2 \\ &\Longrightarrow a^2 + b^2 \le c^2 \end{aligned}$$

Analyses..

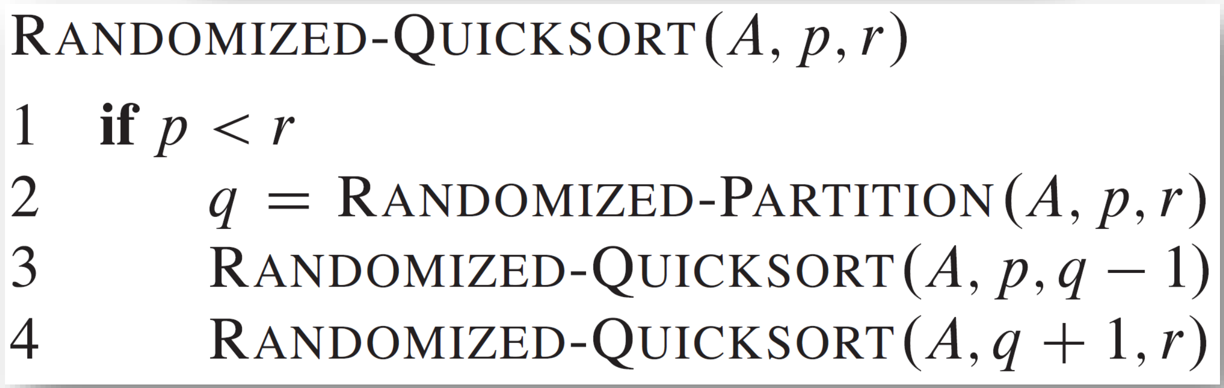
- Best-case partitioning
  - The best case will occur when the partition function produces two subproblems, each of size no more than $\frac{n}{2}$
  - In other words, one is of size $\left\lfloor \frac{n}{2} \right\rfloor$ and one of size $\left\lceil \frac{n}{2} \right\rceil - 1$
  - The partitioning costs $\Theta(n)$ time
  - Thus, $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$
  - By case 2 of the master theorem, we can obtain $T(n) = \Theta(n \log_2 n)$

Randomized Quick Sort.

- Instead of always using $A[r]$ as the pivot, we will select a randomly chosen element from the subarray $A[p..r]$
  - Because we randomly choose the pivot element, we expect the split of the input array to be reasonably well balanced on average



```
RANDOMIZED-PARTITION(A, p, r)
    i = RANDOM(p, r)
    exchange A[r] with 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)
```

Randomized Quick Sort..

- Instead of always using $A[r]$ as the pivot, we will select a randomly chosen element from the subarray $A[p..r]$
  - Because we randomly choose the pivot element, we expect the split of the input array to be reasonably well balanced on average

```
PARTITION($A, p, r$)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

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

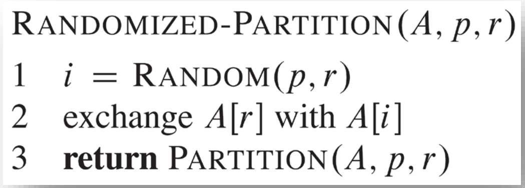
```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q - 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

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

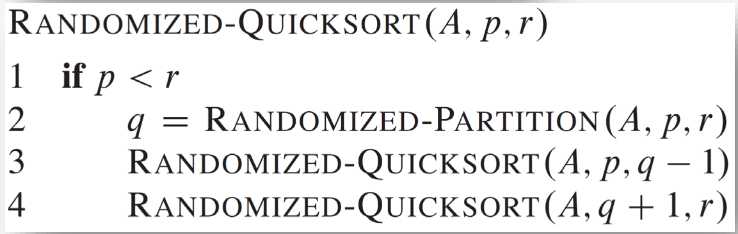
Analyses.

- The QUICKSORT and RANDOMIZED-QUICKSORT procedures differ only in how they select pivot elements
  - They are the same in all other respects

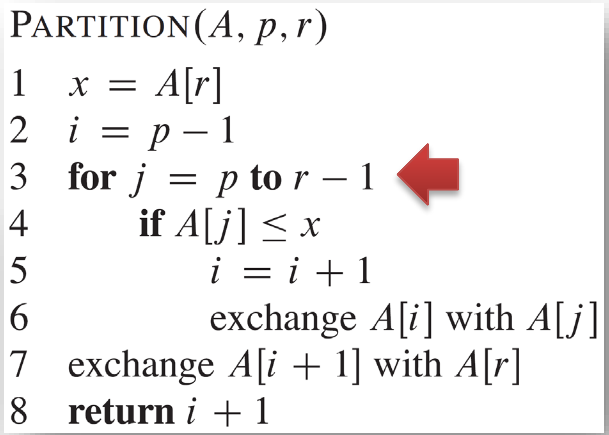
- In the PARTITION procedure, each iteration of the **for** loop performs a comparison, comparing the pivot element to another element of the array *A*
  - If we can count the total number of times that the loop is executed, we can bound the total time



```
RANDOMIZED-PARTITION$(A, p, r)$
$i = $ RANDOM$(p, r)$
exchange $A[r]$ with $A[i]$
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$(A, q + 1, r)$
```



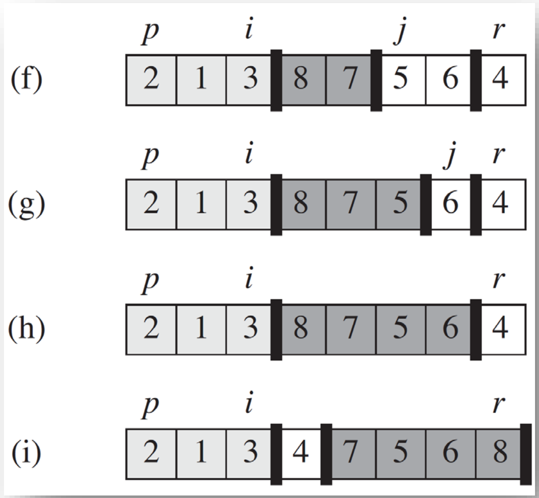
```
PARTITION$(A, p, r)$
$x = A[r]$
$i = p - 1$
for $j = p$ to $r - 1$
    if $A[j] \leq x$
        $i = i + 1$
        exchange $A[i]$ with $A[j]$
exchange $A[i + 1]$ with $A[r]$
return $i + 1$
```

Analyses..

- For ease of analysis, we rename the array A as $z_1, z_2, \dots, z_n$
  - z_i being the i^{th} element
  - $Z_{ij} = \{z_i, z_{i+1}, \dots, z_j\}$ to be the set of elements between z_i and z_j

- It is easy to understand that each pair of elements is compared **at most** once
  - We define $X_{ij} = \mathrm{I}\{z_i \textit{ is compared to } z_j\}$
  - Since each pair is compared at most once, we can easily characterize the total number of comparisons performed by the algorithm

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$



Analyses...

- Taking expectations of both sides

$$\begin{aligned}\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\ &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} P(X_{ij}\colon\ z_i \textit{ is compared to } z_j)\end{aligned}$$

- Since the RANDOMIZED-PARTITION procedure chooses each pivot **randomly** and **independently**

$$\begin{aligned}P\left(X_{ij}\colon\ z_i \textit{ is compared to } z_j\right) &= P\left(z_i \textit{ or } z_j \textit{ is chosen to be pivot from } Z_{ij}\right) \\ &= P\left(z_i \textit{ is chosen to be pivot from } Z_{ij}\right) + P\left(z_j \textit{ is chosen to be pivot from } Z_{ij}\right) \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\ &= \frac{2}{j-i+1}\end{aligned}$$

Analyses….

- Consequently, we can get
  - Let $k = j - i$

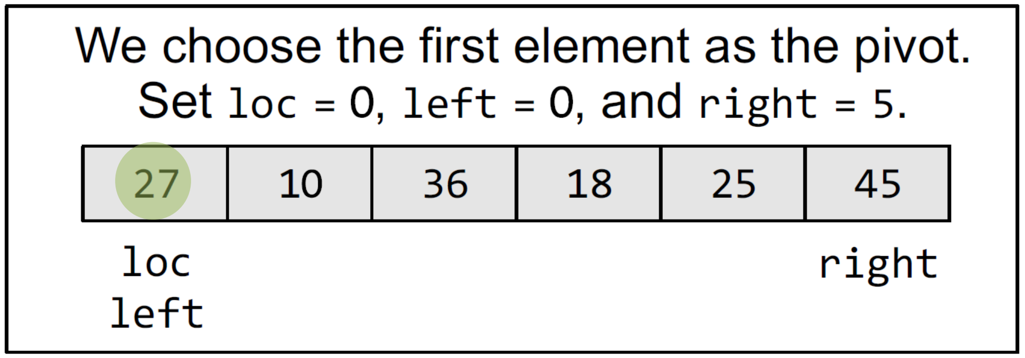
$$
\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(X_{ij}\colon z_i \textit{ is compared to } z_j) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} \mathrm{O}(\log_2 n) \\
&= \mathrm{O}(n \log_2 n)
\end{aligned}
$$

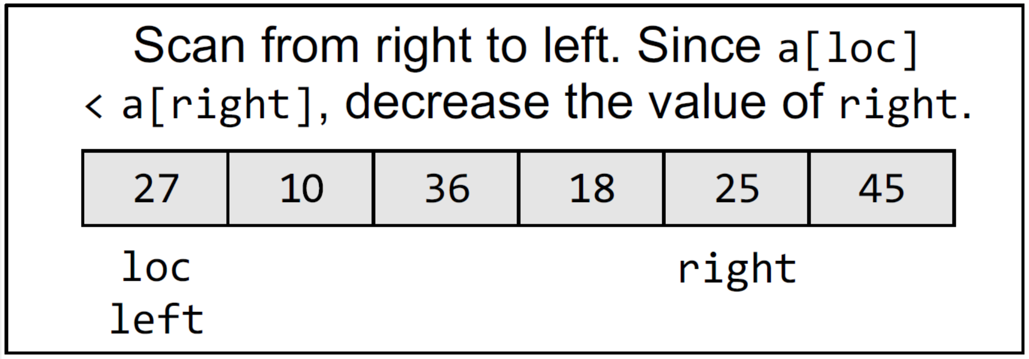
- We conclude that, using RANDOMIZED-PARTITION, the expected running time of quicksort is $\mathrm{O}(n \log_2 n)$

Variant.

- Sort the given array using quick sort algorithm

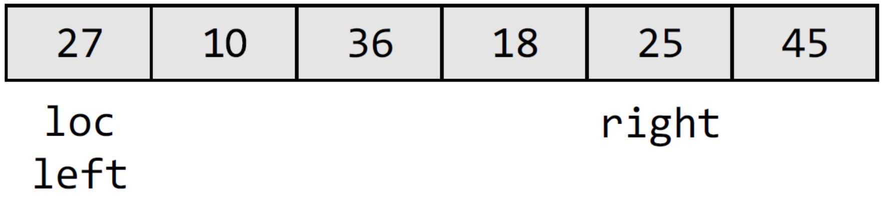
27	10	36	18	25	45


We choose the first element as the pivot. Set `loc` = 0, `left` = 0, and `right` = 5.

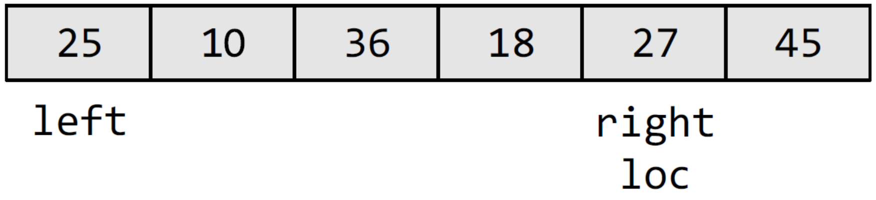

Scan from right to left. Since `a[loc]` < `a[right]`, decrease the value of `right`.

Variant..

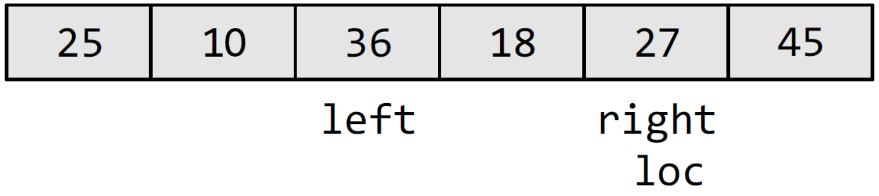
Scan from right to left. Since `a[loc]` < `a[right]`, decrease the value of `right`.



Since `a[loc]` > `a[right]`, interchange the two values and set `loc = right`.

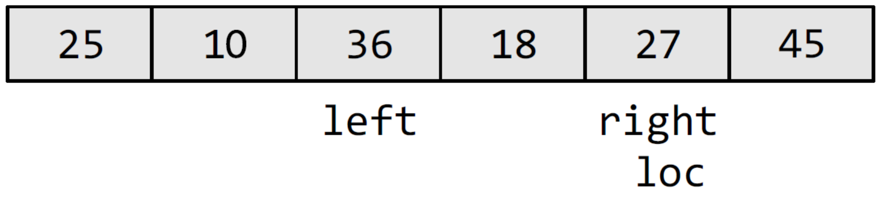


Start scanning from left to right. Since `a[loc]` > `a[left]`, increment the value of `left`.

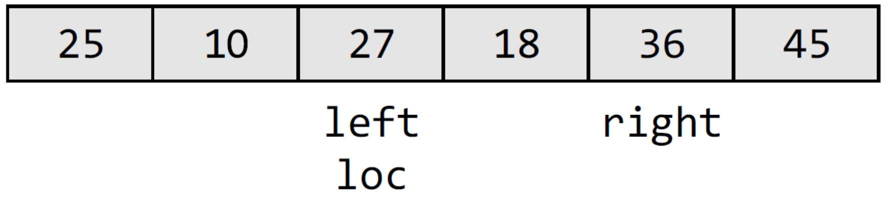


Variant...

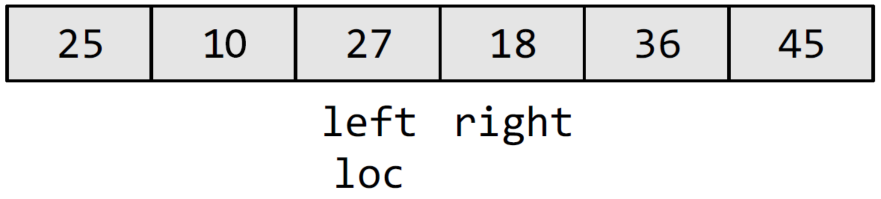
Start scanning from left to right. Since `a[loc]` > `a[left]`, increment the value of `left`.



Since `a[loc]` < `a[left]`, interchange the values and set `loc = left`.

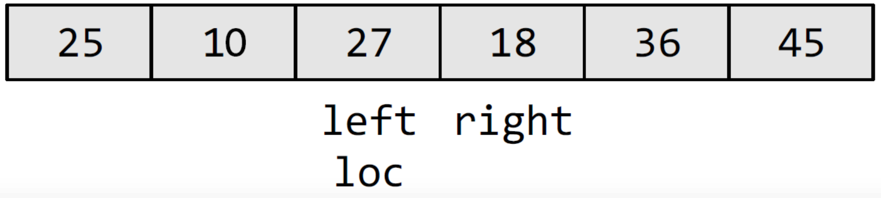


Scan from right to left. Since `a[loc]` < `a[right]`, decrement the value of `right`.

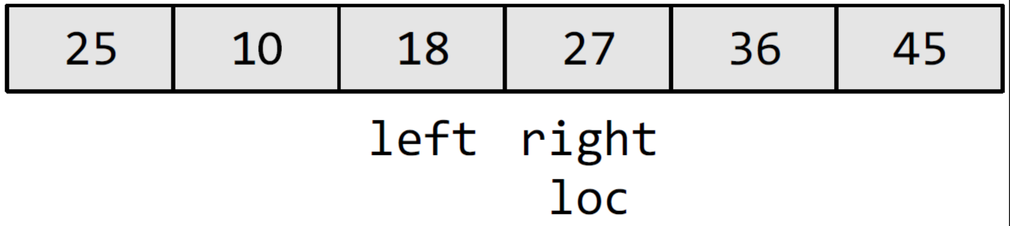


Variant….

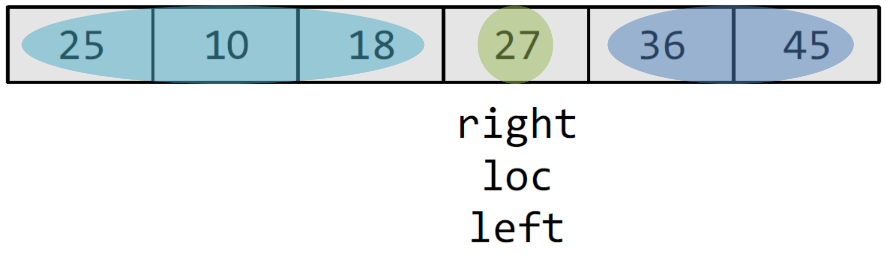
Scan from right to left. Since `a[loc] < a[right]`, decrement the value of `right`.



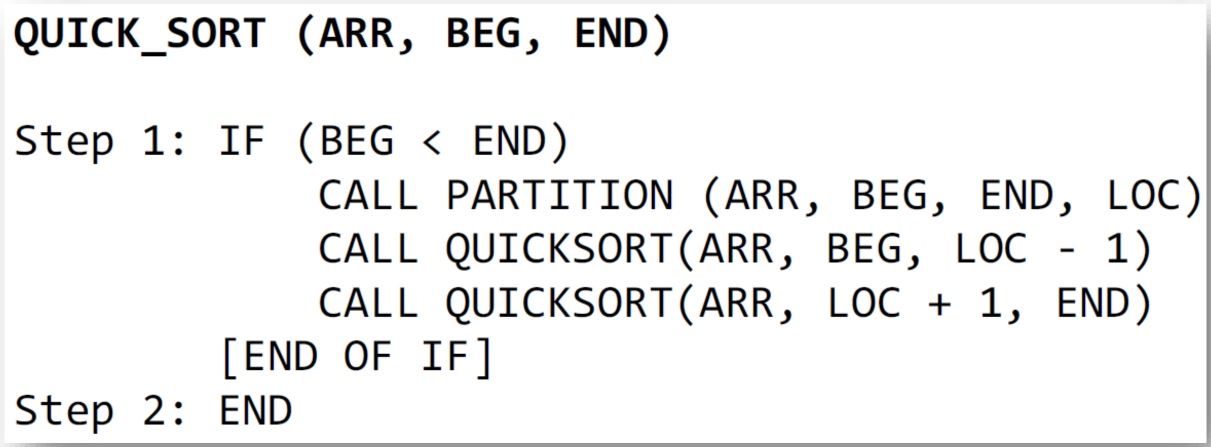
Since `a[loc] > a[right]`, interchange the two values and set `loc = right`.



Start scanning from left to right. Since `a[loc] > a[left]`, increment the value of `left`.



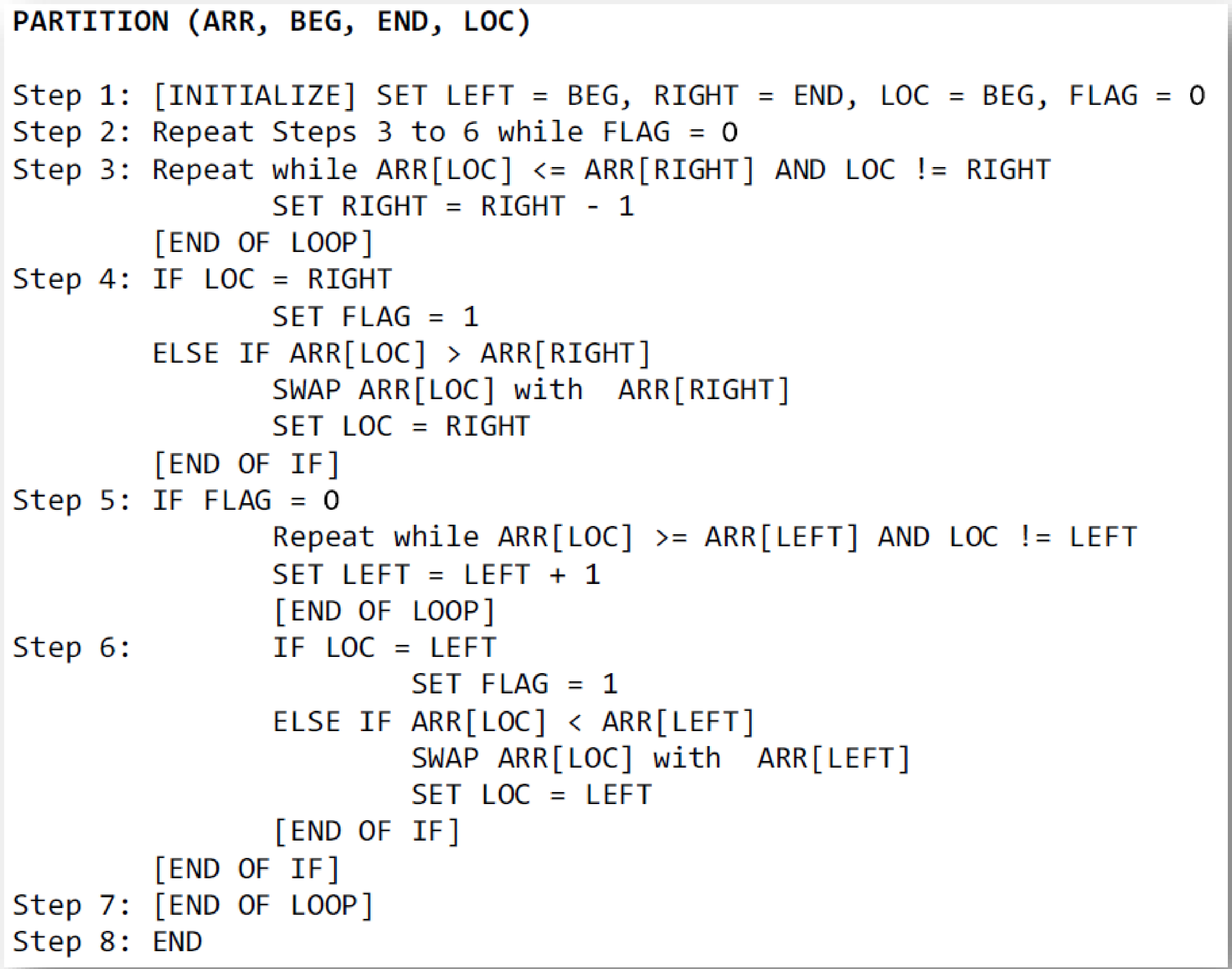
Variant.....



```
QUICK_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)
            CALL PARTITION (ARR, BEG, END, LOC)
            CALL QUICKSORT(ARR, BEG, LOC - 1)
            CALL QUICKSORT(ARR, LOC + 1, END)
        [END OF IF]
Step 2: END
```

Variant……



```
PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0
Step 2: Repeat Steps 3 to 6 while FLAG = 0
Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT
               SET RIGHT = RIGHT - 1
        [END OF LOOP]
Step 4: IF LOC = RIGHT
               SET FLAG = 1
        ELSE IF ARR[LOC] > ARR[RIGHT]
               SWAP ARR[LOC] with  ARR[RIGHT]
               SET LOC = RIGHT
        [END OF IF]
Step 5: IF FLAG = 0
               Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
               SET LEFT = LEFT + 1
               [END OF LOOP]
Step 6:        IF LOC = LEFT
                       SET FLAG = 1
               ELSE IF ARR[LOC] < ARR[LEFT]
                       SWAP ARR[LOC] with  ARR[LEFT]
                       SET LOC = LEFT
               [END OF IF]
        [END OF IF]
Step 7: [END OF LOOP]
Step 8: END
```

Questions?

kychen@mail.ntust.edu.tw