Sorting algorithms: Quicksort

Numerous sorting algorithms are there. We have discussed so far about

- Insertion sort
- Merge sort
- Heap sort

We now take a look at **Quicksort** that on an average runs 2-3 faster that Merge sort or Heap sort.

Quicksort (Divide-and-conquer)

The major steps:

(Pivot) Given an array of numbers, choose a pivot p

(Partition) Reorder the elements, so that all elements < p appear before p, and all elements > p appear after p. The elements equal to p can appear anywhere in between the smaller (than p) and the larger (than p) elements.

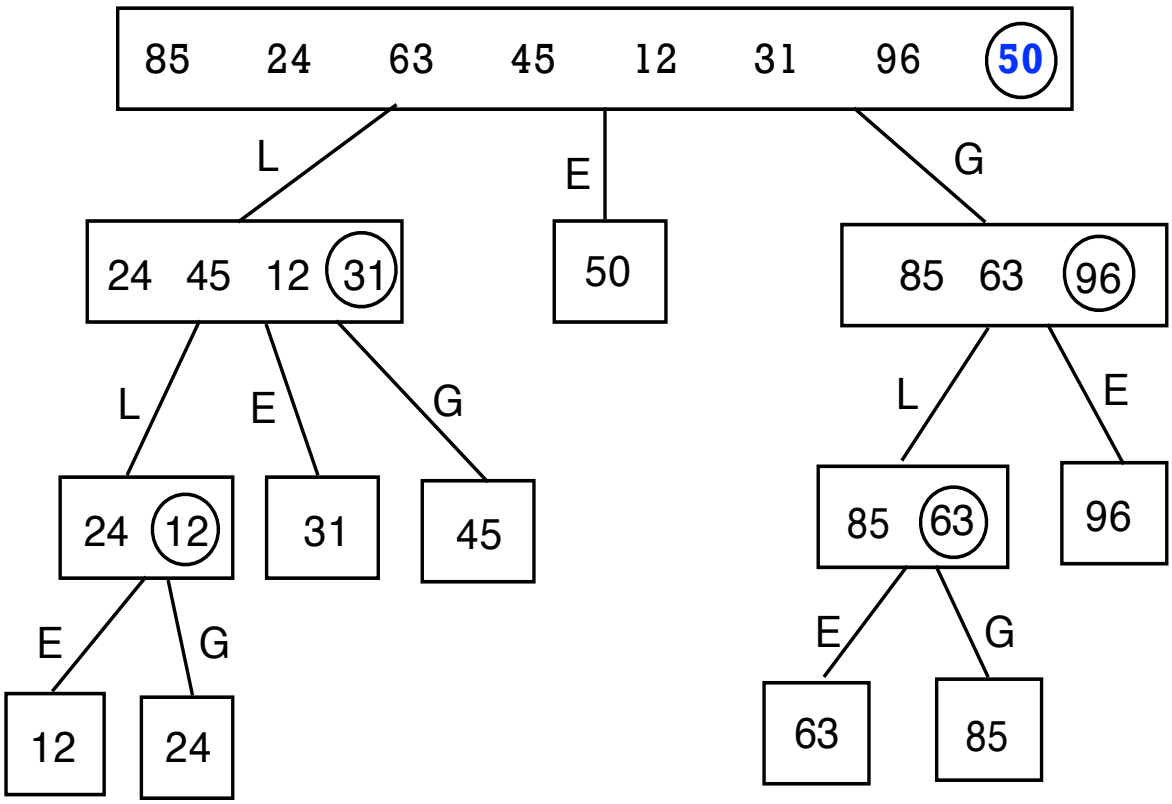
(Recursion) Apply this to the sub-arrays in the partition, until their sizes become one, the base case.

(Concatenation) Combine the sorted sub-arrays into a sorted queue

Any element can be chosen as the pivot, but usually the last one is picked since it gives the best performance. Assume that the initial values are in a queues S

```
//partition the queue
public static int void(quicksort Queue S){
int pivot = S.last();
// construct three queues L, G, E (not shown)
while !S.isEmpty() {
int element = S.dequeue();
if (element < pivot)
    L.enqueue (element);
else if (element = pivot)
    E.enqueue (element)
else G.enqueue (element);
}
// Recursive step
quicksort(L);
quicksort(G);
//Concatenate results
while (!L.isEmpty)
    S.enqueue(L.dequeue());
while (!E.isEmpty)
    S.enqueue(L.dequeue());
while (!G.isEmpty)
    S.enqueue(L.dequeue());
}
```

Example

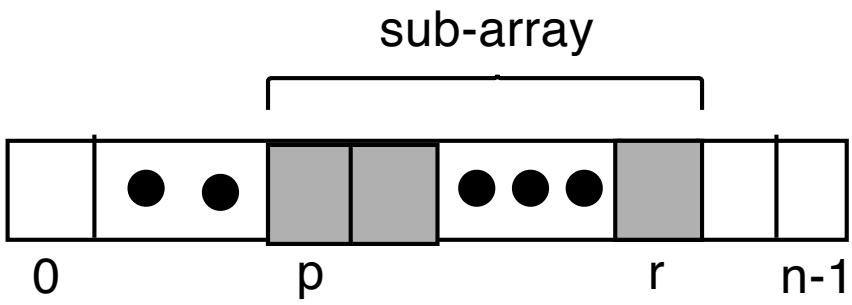


Complexity: Best case O(n log n) Why?

Complexity: Worst case O(n^2) Why?

Can we avoid using queues and shuffle the elements **in-place** within the array to sort the elements? Yes.

Array Version: Algorithm 1



Pseudocode for Quicksort

```
QuicksortInPlace (A, p, r)
If p < r then
    q = partition (A, p, r)
    QuicksortInPlace (A, p, q-1)
    QuicksortInPlace (A, q+1, r)
```

The value of q divides the array into two parts.

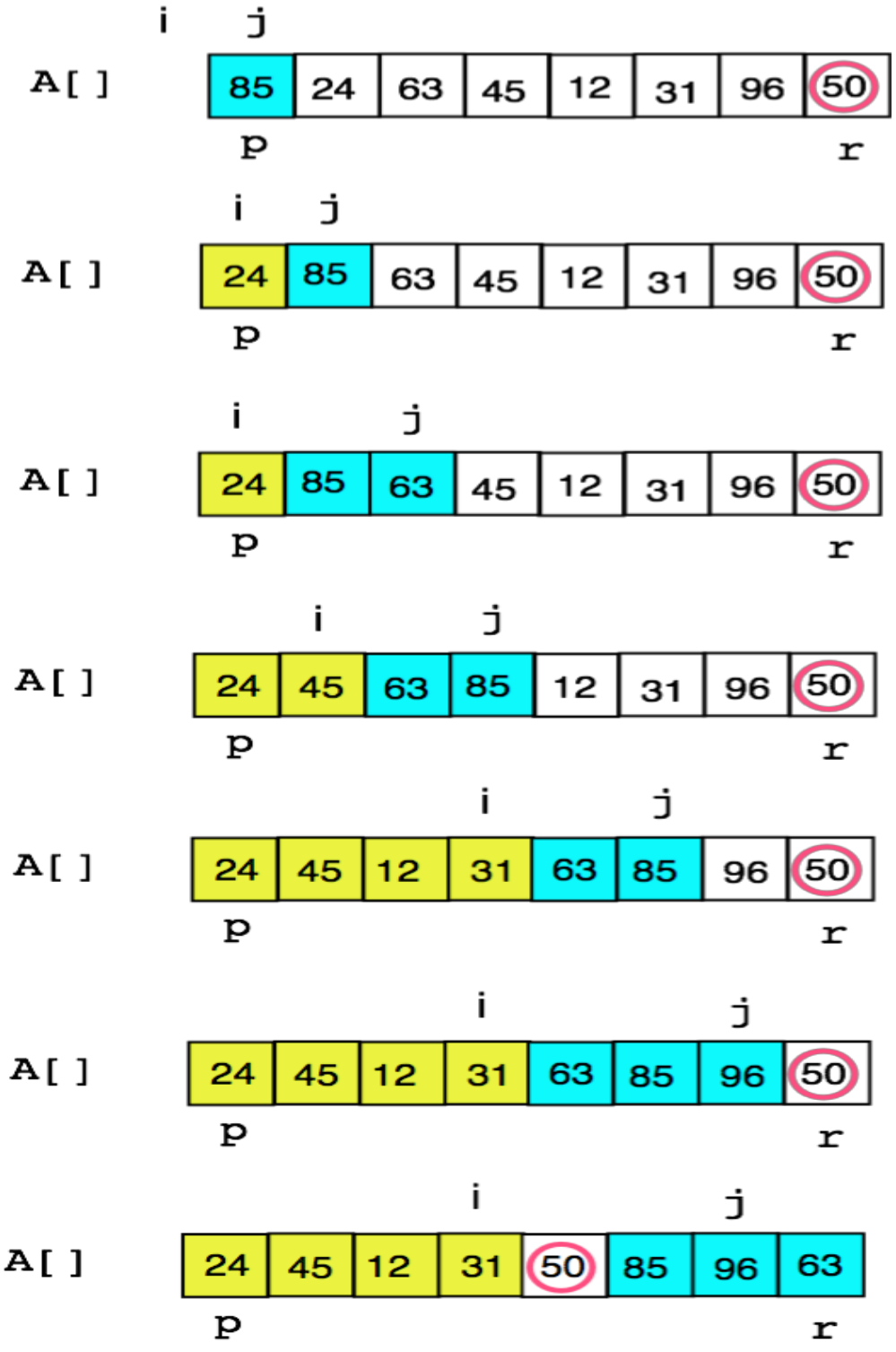
The initial call is

QuicksortInPlace (A, 0, length(A)-1)

Pseudo-code for partition in QuickSortInPlace

```
partition (A, p, r)
pivot = A[r]
i = p-1
for j = p to r-1 {
    if A(j) < pivot {
        i=i+1
        swap (A[i], A[j])
    }
swap (A[i+1], A[r])
// q = i+1 divides the array
```

An example of partition



Performance of Quicksort

Quick sort vs Merge sort

Both are comparison-based sorts.

Merge sort simply divides the list into two (almost) equal parts, but does some extra work before merging the parts.

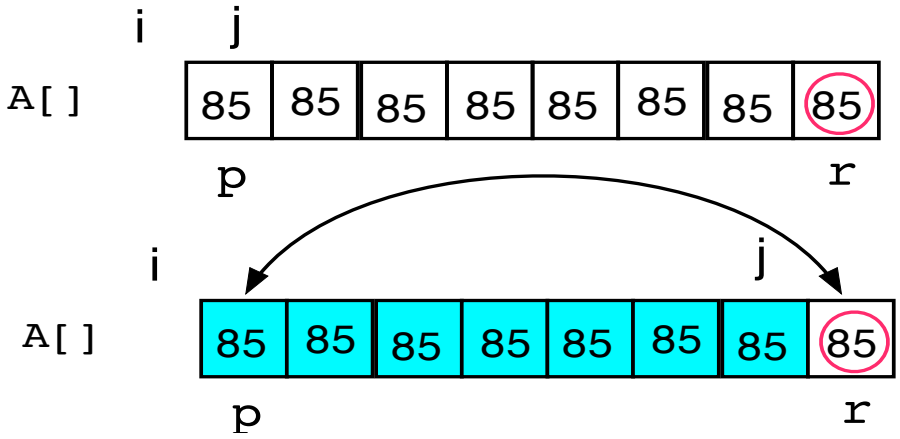
Quicksort does the extra work before dividing it into parts, but merging is simple concatenation.

Quicksort is the fastest known comparison-based sort

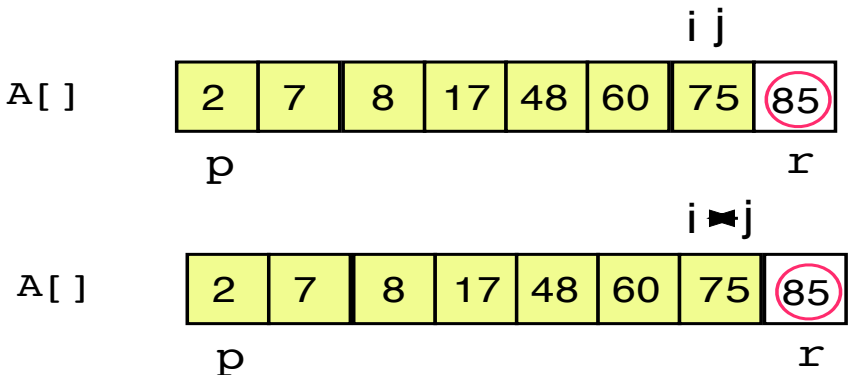
The link https://www.youtube.com/watch?v=YvTW7341kpA Contains an old (1980's) but nice video on three sorting techniques

What if all keys equal?

Such keys can go to either half. Algorithm 1 discussed earlier will lead to $O(n^2)$ time complexity



What if all the keys are already sorted?



Again, Algorithm 1 will lead to $O(n^2)$ time complexity.

Randomly choosing the pivot overcomes the second problem.

Randomly choose pivot and swap it with the last item

Random pivot helps $(1/4 - 3/4)$ split in most cases). Also, for large arrays, *Median of three* (random choices) works even better. Take three randomly chosen array indices and pick the middle one to pick the pivot.

Quicksort on Linked List

Split into three lists L (less than) G (greater than), E (equal to pivot). Of these, sort only L, G not E. It reduces the effort, but does not work with array in-place)

Sort (5, 7, 5, 0, 6, 5, 5)
0 | 5, 5, 5, 5 | 7, 6

Random pivot choice is annoying for Linked List.

Quicksort Algorithm 2

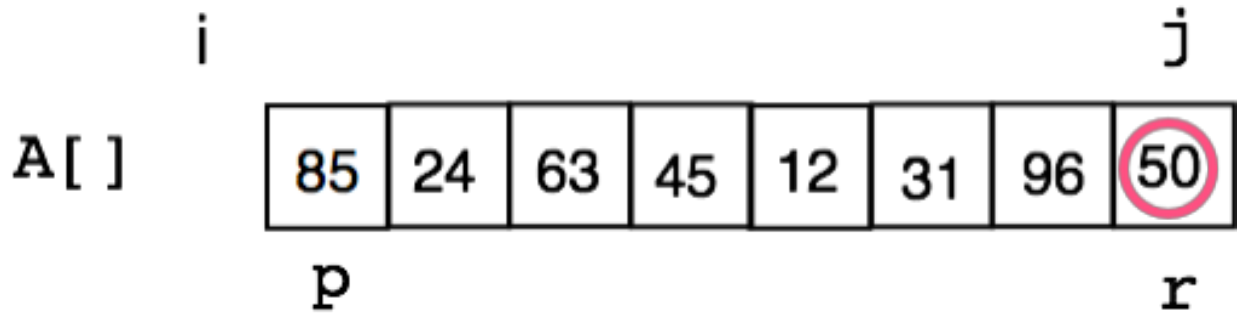
It is a better version of Quicksort.

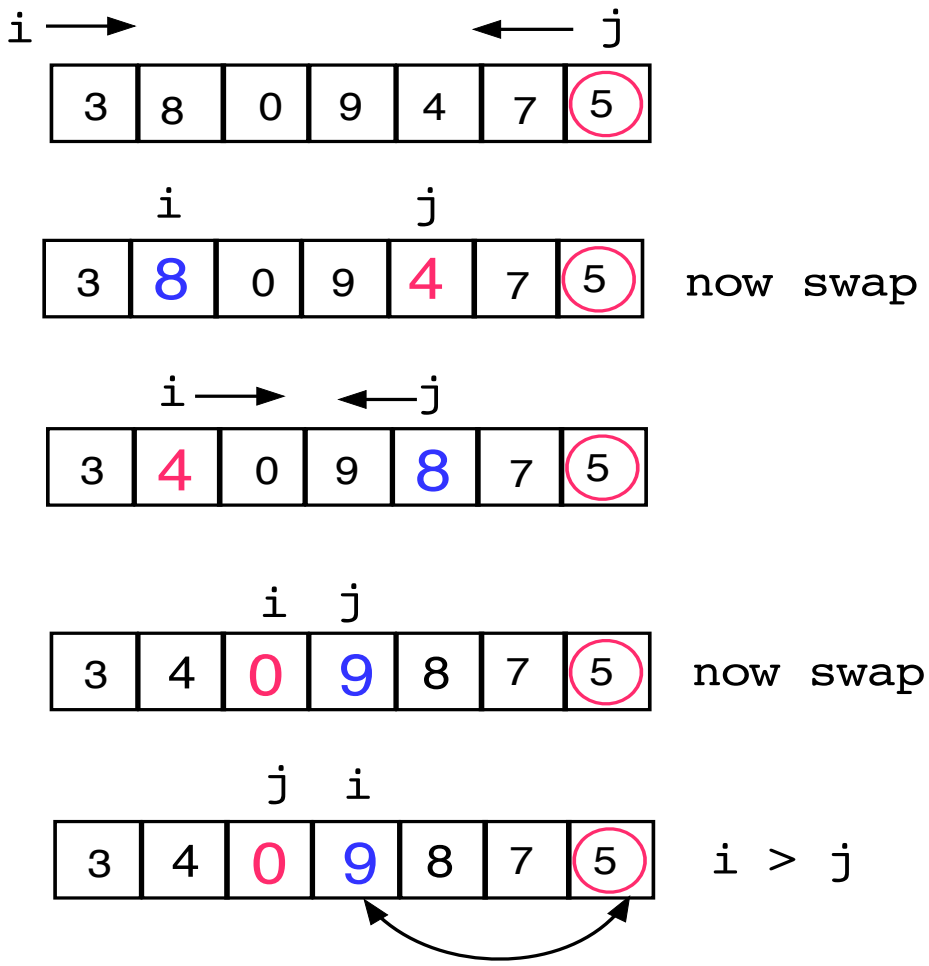
```
To sort A[p] – A[r], use two pointers i and j

Initialize i= p-1 and j = r
(Between i,j  sandwich the items to be sorted).

Move i from left to right as long as A[i]<pivot,
Move j from right to left as long as A[j]>pivot.

Swap (A[i], A[j])
```




i
j
3 8 0 9 4 7 5
i
j
3 8 0 9 4 7 5
now swap
i
j
3 4 0 9 8 7 5
i
j
3 4 0 9 8 7 5
now swap
j
i
3 4 0 9 8 7 5
i > j

```
public static void quicksort(int[ ] a, int low, int high){

   if (low < high) {
     int pivotIndex = random number from low to high;
     pivot = a[pivotindex];
     a[pivotIndex] = a[high]; // Swap pivot with last
     a[high] = pivot;

      int i = low - 1;
      int j = high;

     do {
       do { i++; } while (a[i] < pivot);
       do { j--; } while ((a[j] > pivot)&&(j > low));
       if (i < j) {
         swap a[i] and a[j];
       }
     } while (i < j);

     a[high] = a[i];
     a[i] = pivot;    // Put pivot where it belongs

     quicksort(a, low, i - 1); // Sort left sub-array
     quicksort(a, i + 1, high); // Sort right sub-array
   }
 }
```

Notes

Works great with both arrays containing all equal elements, and already sorted arrays,

Can the `"do {i++}" loop` walk off the end of the array and generate an out-of- bounds exception? No, because `a[high]` contains the pivot, so i will stop advancing when `i == high` (if not sooner).

There is no such assurance for `j,` so the `"do {j--}"` loop explicitly tests whether `"j > low"` before retreating.