Data Structures and Algorithms

Lecture 6: Quicksort (快速排序)

Introduction

- Quicksort was proposed by C. A. R. Hoare (Turing award in 1980) in 1961, it is one of the Top 10 Algorithms in the 20th century.
- Fastest known sorting algorithm in practice
- Average case: O(N log N)
- Worst case: $O(N^2)$
  - But, the worst case seldom happens.
- Another divide-and-conquer recursive algorithm, like mergesort

THE TOP 10 LIST

1946: The Metropolis Algorithm
1947: Simplex Method
1950: Krylov Subspace Method
1951: The Decompositional Approach to Matrix Computations
1957: The Fortran Optimizing Compiler
1959: QR Algorithm
1962: Quicksort
1965: Fast Fourier Transform
1977: Integer Relation Detection
1987: Fast Multipole Method

Dantzig von Neumann Hestenes Householder Backus Hoare Greengard

Quicksort for set S

- Divide step:
  - Pick any element (***pivot,*** *主元*) v in set S
  - Partition S – {v} into two disjoint groups

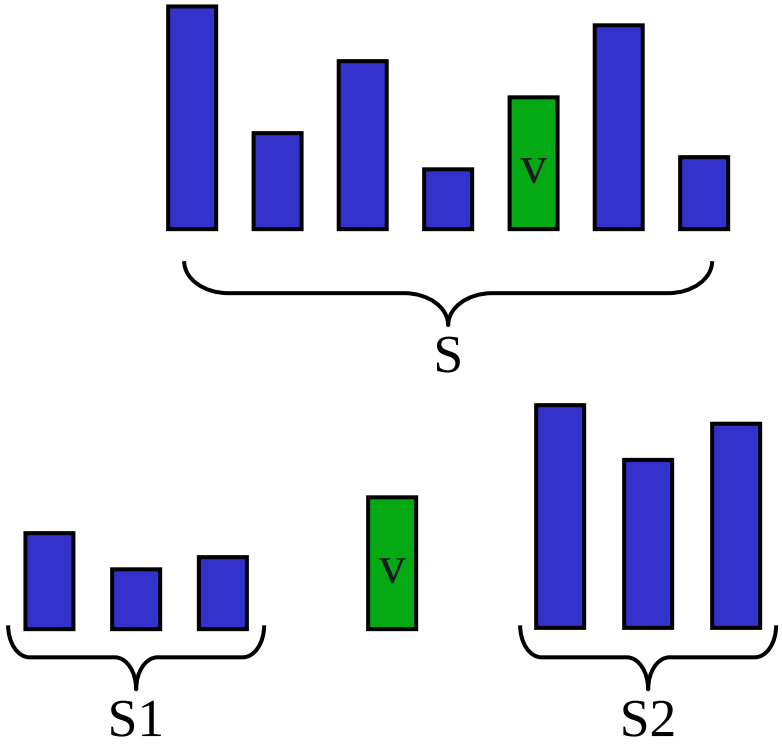
    S1 = {x ∈ S – {v} | x <= v}

    S2 = {x ∈ S – {v} | x **>=** v}

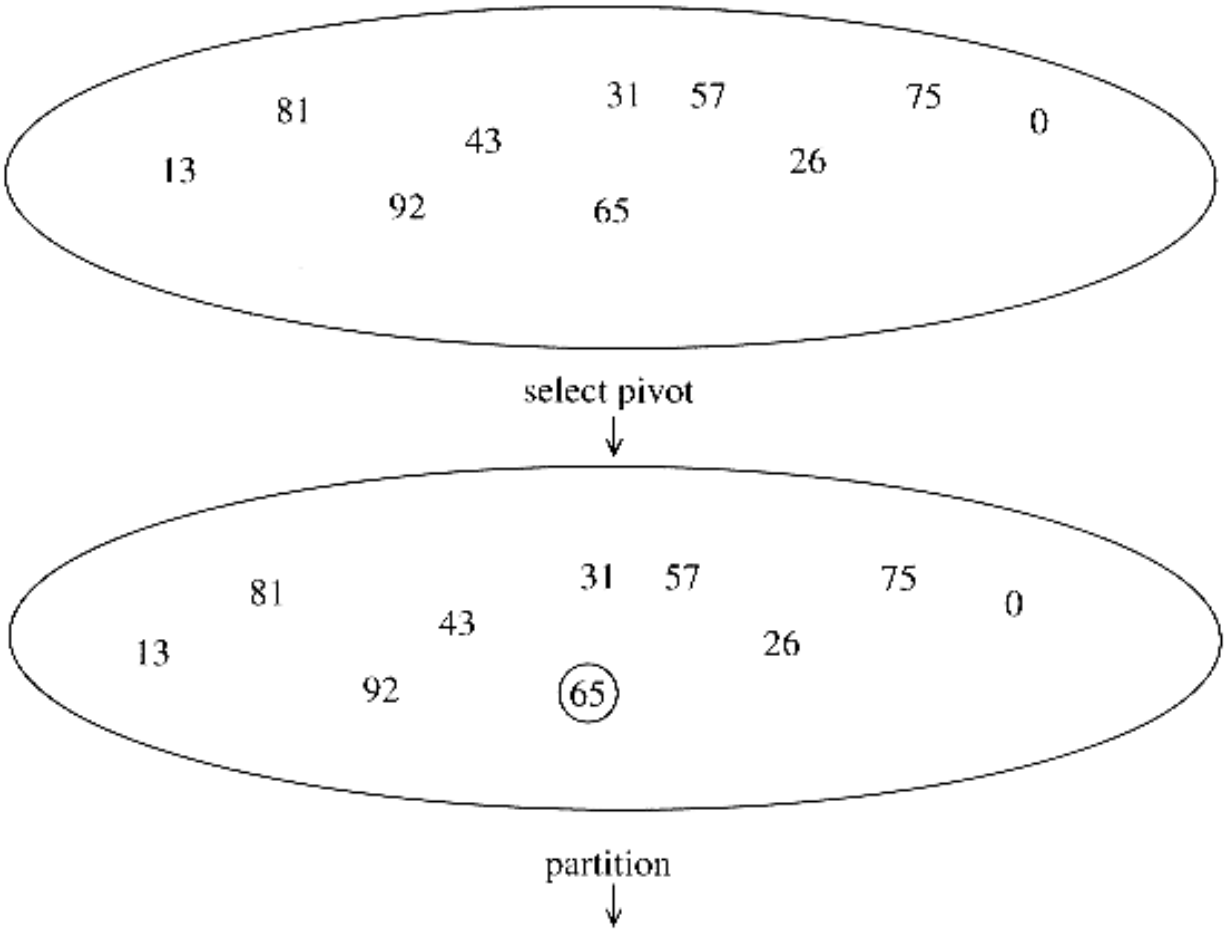
  Note: S1 and S2 may overlap. Why?

- Conquer step: recursively sort S1 and S2

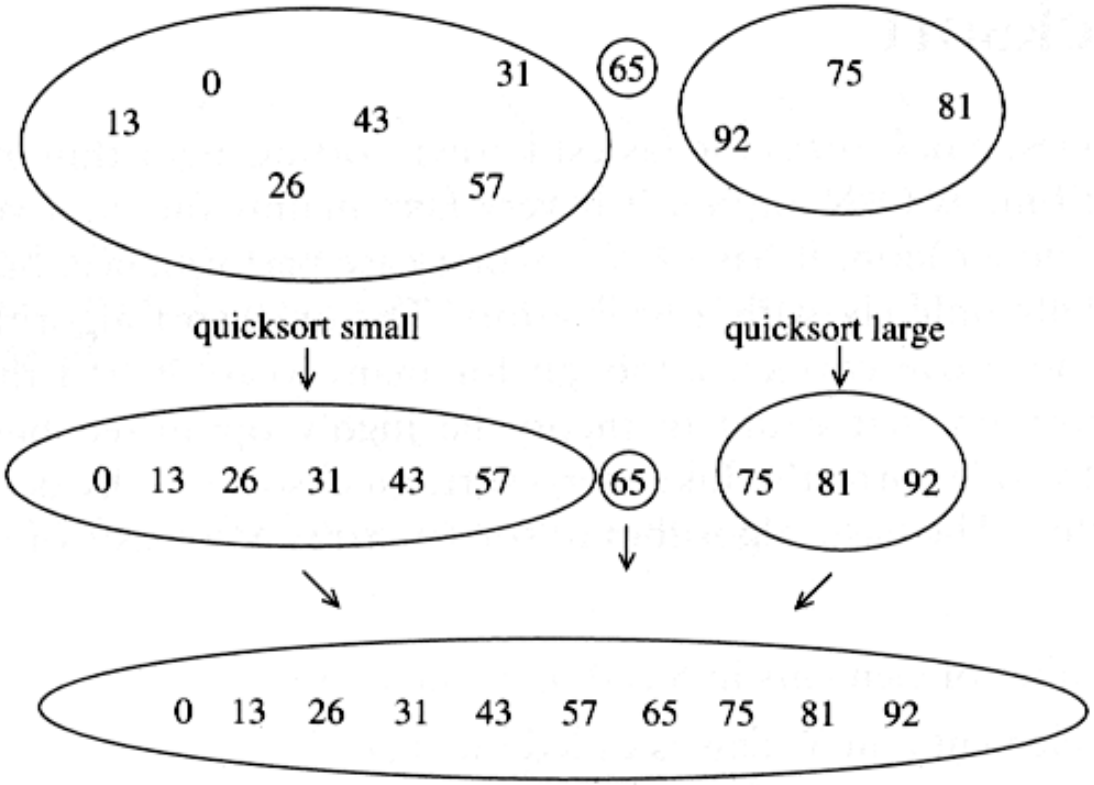
- Combine step: the sorted S1 (by the time returned from recursion), followed by v, followed by the sorted S2 (i.e., nothing extra needs to be done)



Example: Quicksort



Example: Quicksort...



Pseudocode

Input: an array A[left ... right]

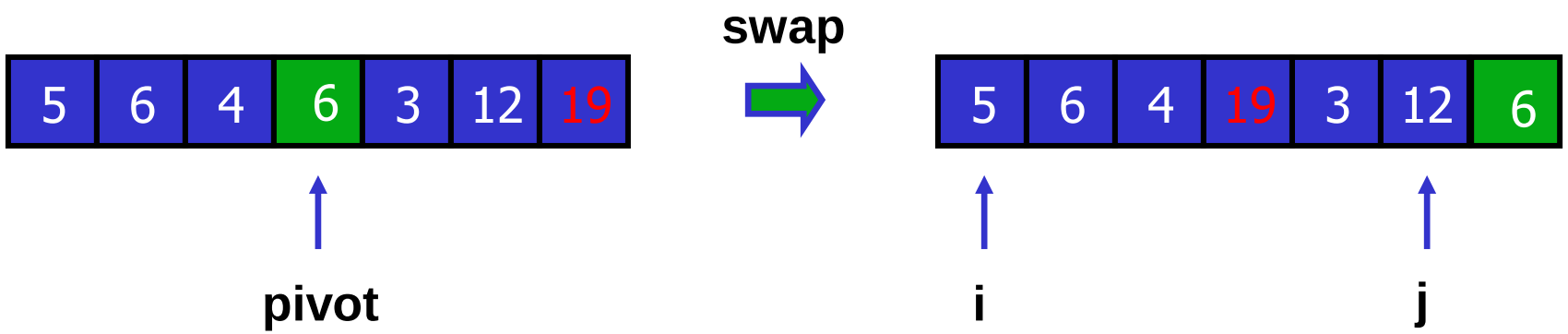
```
Quicksort (A, left, right) {
    if (left < right) {// The array has at least two elements
        q = Partition (A, left, right)   //q is the position of the pivot element
        Quicksort (A, left, q-1)
        Quicksort (A, q+1, right)
    }
}
```

Partitioning

- Partitioning
  - This is a **key step** of the quicksort algorithm
  - **Goal**: given the picked pivot, partition the remaining elements into two smaller sets
  - Many ways to implement how to partition:
    - Even the slightest deviations may cause surprisingly bad results.
- We will learn an easy and efficient partitioning strategy here.
- How to pick a pivot will be discussed later

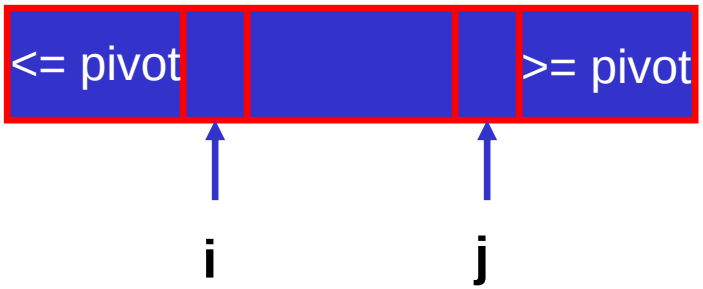
Partitioning Strategy

- Want to partition an array A[left ... right]
- First, get the pivot element out of the array by swapping it with the last element. (Swap pivot and A[right])
- Let i start at the first element and j start at the next-to-last element (i = left, j = right – 1)

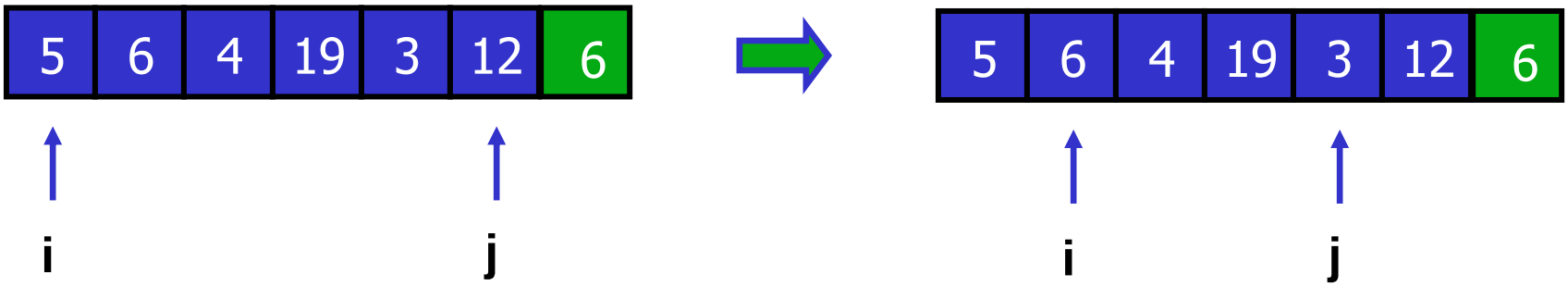


Partitioning Strategy

- Want to have
  - A[p] <= pivot, for p < i
  - A[p] >= pivot, for p > j

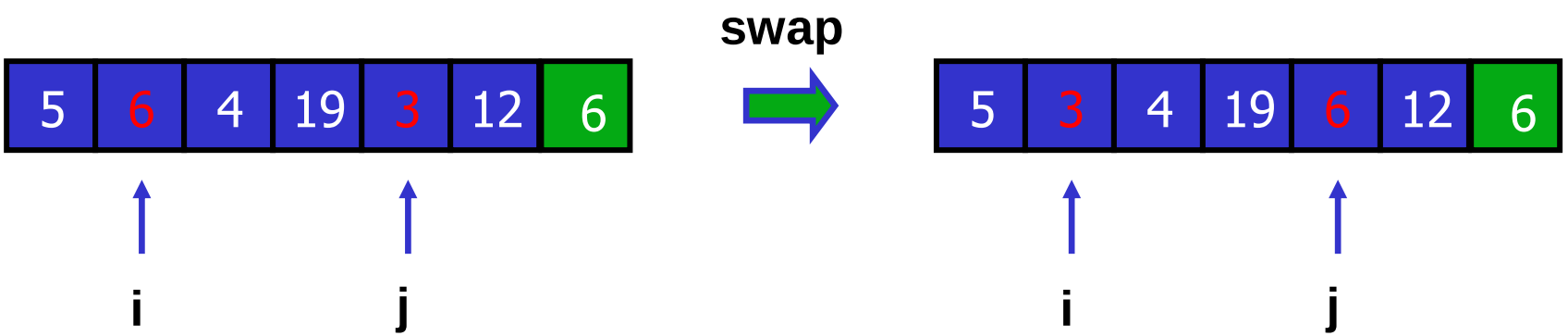


- When i < j
  - Move i right, skipping over elements smaller than the pivot
  - Move j left, skipping over elements greater than the pivot
  - When both i and j have stopped
    - A[i] >= pivot
    - A[j] <= pivot { A[i] and A[j] should now be **swapped**}



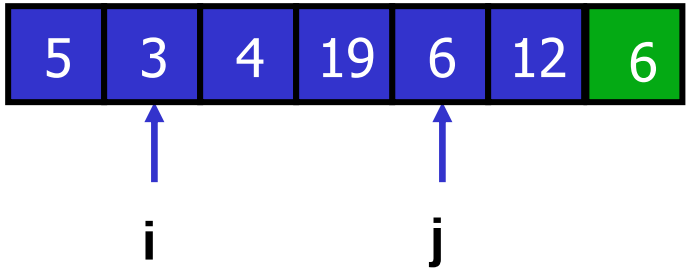
Partitioning Strategy

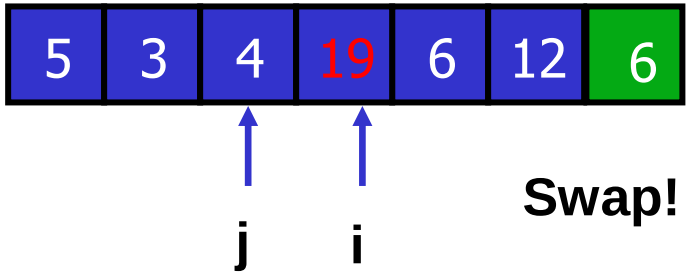
- When i and j have stopped and i is to the left of j (thus legal)
  - Swap A[i] and A[j]
    - The larger element is pushed to the right and the smaller element is pushed to the left
  - After swapping
    - A[i] <= pivot
    - A[j] >= pivot
  - Repeat the process until i and j **cross**

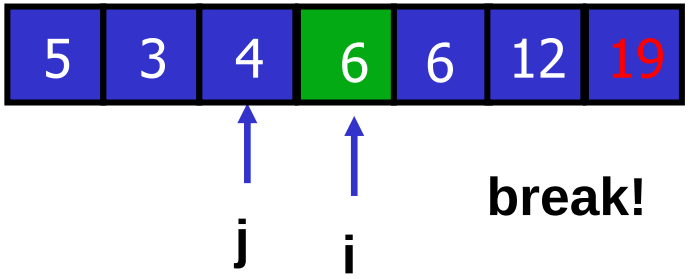


Partitioning Strategy

- When i and j have crossed
  - Swap A[i] and pivot
- Result:
  - A[p] <= pivot, for p < i
  - A[p] >= pivot, for p > i







Pseudo Code for Partition

```
PARTITION(A, left, right)
1.      p = PIVOT(A, left, right)
2.     //p is the position of the pivot
3.     swap A[p] and A[right]
4.     i = left, j = right-1, pivot = A[right]
5.     WHILE true
6.         WHILE i<right AND A[i]<pivot
7.             i++
8.         WHILE j>=left AND A[j]>pivot
9.             j--
10.        IF i<j
11.            swap A[i++] and A[j--]
12.        ELSE
13.            BREAK
14.    swap A[i] and A[right]
15.    return i
```

Small arrays

- For very small arrays, quicksort does not perform as well as insertion sort
  - how small depends on many factors, such as the time spent making a recursive call, the compiler, etc
- Do not use quicksort recursively for small arrays
  - Instead, use a sorting algorithm that is efficient for small arrays, such as insertion sort

Picking the Pivot

- Use the first element as pivot
  - if the input is random, this strategy is ok
  - if the input is presorted (or in reverse order)
    - all the elements go into S2 (or S1)
    - this happens consistently throughout the recursive calls
    - Results in $O(n^2)$ behavior (Analyze this case later)
- Choose the pivot randomly
  - generally safe
  - random number generation can be expensive

Picking the Pivot

- Use the median of the array
  - Partitioning always cuts the array into roughly half
  - An optimal quicksort (O(N log N))
  - However, hard to find the exact median
    - e.g., sort an array to pick the value in the middle

Pivot: median of three

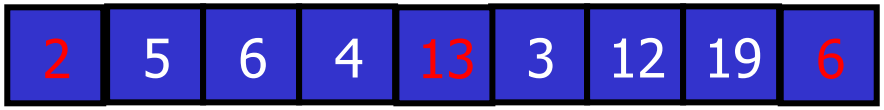
- We will use median of three
  - Compare just three elements: the left most, right most and center
  - Swap these elements if necessary so that
    - A[left] = Smallest
    - A[right] = Largest
    - A[center] = Median of three
  - Pick A[center] as the pivot
  - Swap A[center] and A[right – 1] so that pivot is at second last position (why?)

median3

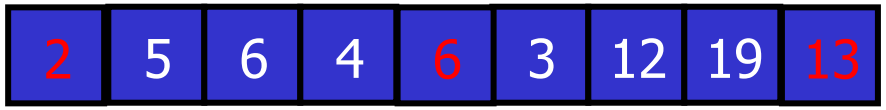
```
int center = ( left + right ) / 2;
if( a[ center ] < a[ left ] )
    swap( a[ left ], a[ center ] );
if( a[ right ] < a[ left ] )
    swap( a[ left ], a[ right ] );
if( a[ right ] < a[ center ] )
    swap( a[ center ], a[ right ] );

    // Place pivot at position right - 1
swap( a[ center ], a[ right - 1 ] );
```

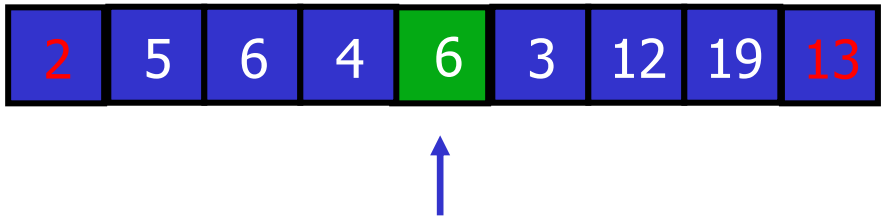
Pivot: median of three



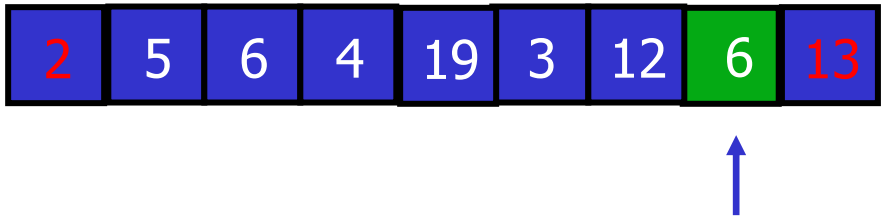
A[left] = 2, A[center] = 13,
A[right] = 6



Swap A[center] and A[right]



Choose A[center] as pivot



Swap pivot and A[right – 1]

Note we only need to partition A[left + 1, …, right – 2]. Why?

Main Quicksort Routine

```
if( left + 10 <= right )
{
    Comparable pivot = median3( a, left, right );

        // Begin partitioning
    int i = left, j = right - 1;
    for( ; ; )
    {
        while( a[ ++i ] < pivot ) { }
        while( pivot < a[ --j ] ) { }
        if( i < j )
            swap( a[ i ], a[ j ] );
        else
            break;
    }

    swap( a[ i ], a[ right - 1 ] );  // Restore pivot

    quicksort( a, left, i - 1 );     // Sort small elements
    quicksort( a, i + 1, right );    // Sort large elements
}
else  // Do an insertion sort on the subarray
    insertionSort( a, left, right );
```

Choose pivot

Partitioning

Recursion

For small arrays

Partitioning Part

- Works only if pivot is picked as median-of-three.
  - A[left] <= pivot and A[right] >= pivot
  - Thus, only need to partition A[left + 1, …, right – 2]
- j will not run past the beginning
  - because a[left] <= pivot
- i will not run past the end
  - because a[right-1] = pivot

```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

Quicksort V.S. Mergesort

- **Auxiliary Space :** Mergesort uses extra space, quicksort requires little space and exhibits good cache locality. Quick sort is an in-place sorting algorithm. In-place sorting means no additional storage space is needed to perform sorting. Merge sort requires a temporary array to merge the sorted arrays and hence it is not in-place giving Quick sort the advantage of space.

- **Worst Cases :** The worst case of quicksort **$O(n^2)$** can be avoided by using randomized quicksort. It can be easily avoided with high probability by choosing the right pivot. Obtaining an average case behavior by choosing right pivot element makes it improvise the performance and becoming as efficient as Merge sort.

- **Merge sort is better for large data structures:** Mergesort is a stable sort, unlike quicksort and heapsort, and can be easily adapted to operate on linked lists and very large lists stored on slow-to-access media such as disk storage or network attached storage. Refer this for details

- Why is quicksort faster than mergesort?
  - The inner loop consists of an increment/decrement (by 1, which is fast), a test and a jump.
  - There is no extra juggling (重新编排) as in mergesort.

Worst Case

- The answer depends on strategy for choosing pivot. In early versions of Quick Sort where leftmost (or rightmost) element is chosen as pivot, the worst occurs in following cases.

- 1) Array is already sorted in same order.
  2) Array is already sorted in reverse order.
  3) All elements are same (special case of case 1 and 2)

Analysis

- Assumptions:
  - A random pivot (no median-of-three partitioning)
  - No cutoff for small arrays
- Running time
  - pivot selection: constant time, i.e. O(1)
  - partitioning: linear time, i.e. O(N)
  - running time of the two recursive calls： T(i)+T(N-i-1)
- T(N)=T(i)+T(N-i-1)+cN where c is a constant
  - i: number of elements in S1

Worst-Case Analysis

- What will be the worst case?
  - The pivot is the smallest element, all the time → **i = 0**
  - Partition is always unbalanced

$$
\begin{aligned}
T(N) &= T(N-1) + cN \\
T(N-1) &= T(N-2) + c(N-1) \\
T(N-2) &= T(N-3) + c(N-2) \\
&\vdots \\
T(2) &= T(1) + c(2) \\
T(N) &= T(1) + c\sum_{i=2}^{N} i = O(N^2)
\end{aligned}
$$

Best-case Analysis

- What will be the best case?
  - Partition is perfectly balanced.
  - Pivot is always in the middle (median of the array)

$$T(N) = 2T\left(\frac{N}{2}\right) + O(N)$$

$$= 2T\left(\frac{N}{2}\right) + c \cdot N$$

$$= 2\left[2T\left(\frac{N}{2^2}\right) + c \cdot \frac{N}{2}\right] + c \cdot N = 2^2 T\left(\frac{N}{2^2}\right) + 2c \cdot N$$

$$\vdots$$

$$= 2^i T\left(\frac{N}{2^i}\right) + ic \cdot N$$

Let $2^i = \mathrm{N}$, *then* $i = log_2 N = logN/log2$

$$= N \cdot T\left(\frac{N}{N}\right) + ic \cdot N = \mathrm{N} + c \cdot N\log \mathrm{N} = O(N\log \mathrm{N})$$

Average-Case Analysis

- Assume
  - Each of the sizes for S1 is equally likely
- This assumption is valid for our pivoting (median-of-three) strategy
- On average, the running time is O(N log N)

  (Covered in COMP2230)

Consider special cases

- When all elements are the same?
- Other cases?