Quick Sort

cse2011

section 11.2 of textbook

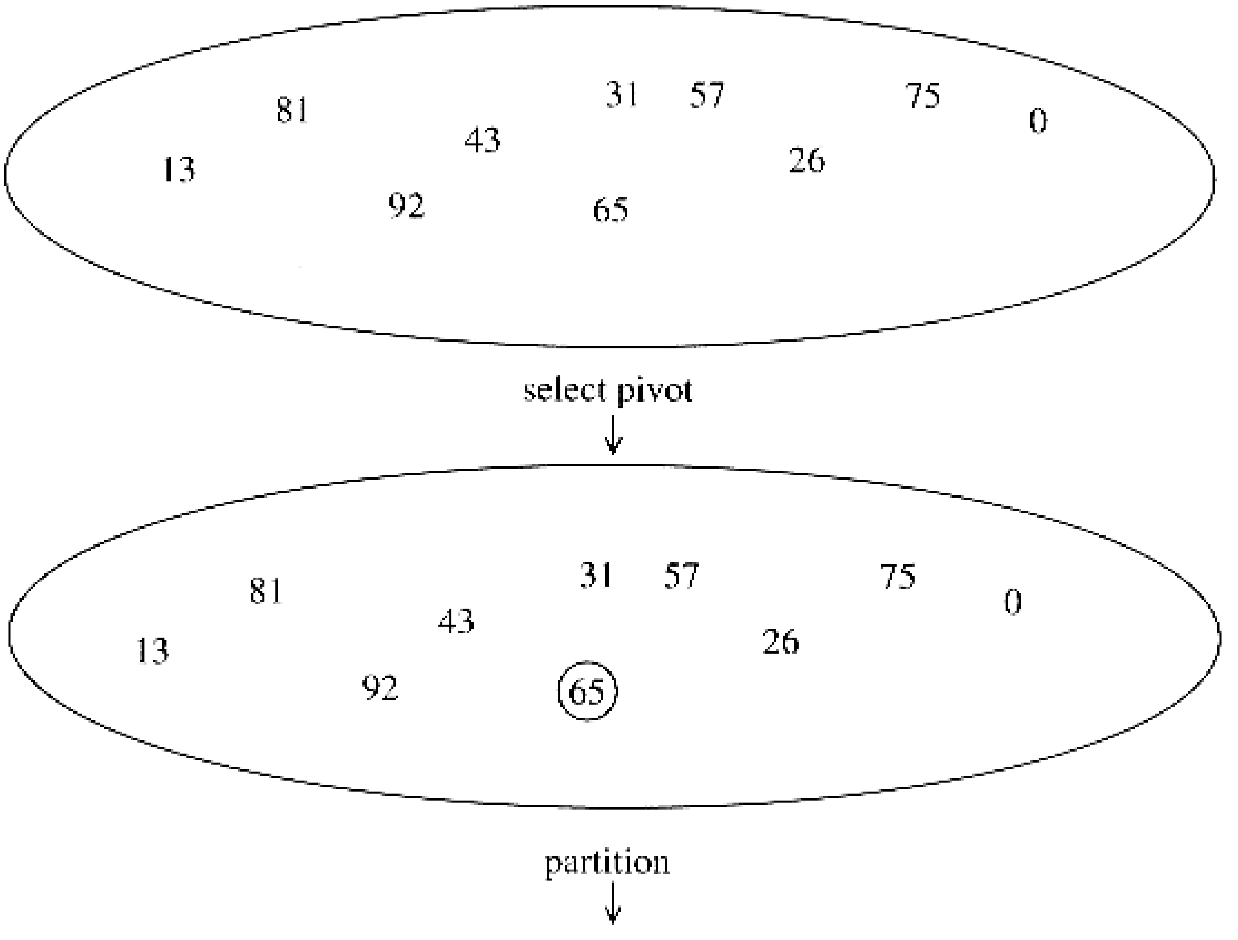
Quick Sort

- **Fastest** known sorting algorithm in practice
- Average case: O(N log N)
- Worst case: $O(N^2)$
  - But the worst case can be made exponentially unlikely.
- Another **divide-and-conquer** recursive algorithm, like merge sort.

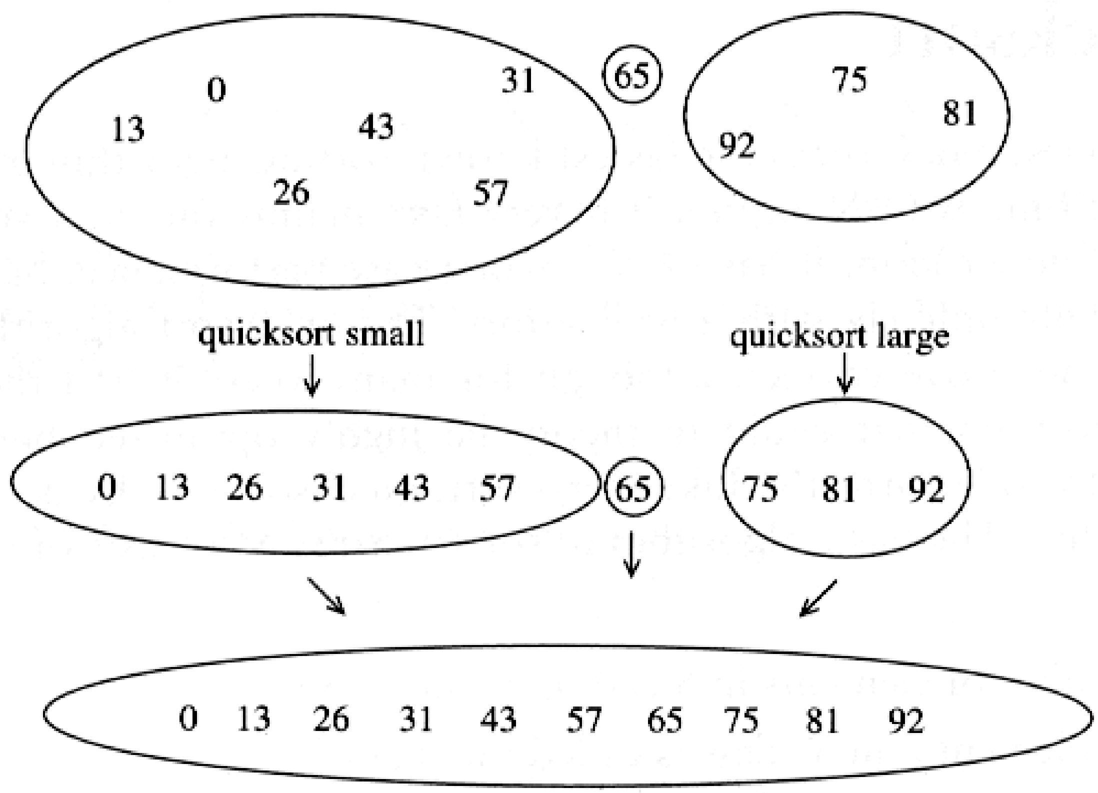
Quick Sort: Main Idea

1. If the number of elements in **S** is 0 or 1, then return (base case/base of the recurrence).
2. Pick any element **v** in **S** (called the pivot).
3. Partition the elements in **S** except **v** into two **disjoint** groups:
   1. $S_1 = \{x \in S - \{v\} \mid x \leq v\}$
   2. $S_2 = \{x \in S - \{v\} \mid x \geq v\}$
4. Return {QuickSort(S_1) + **v** + QuickSort(S_2)}

Quick Sort: Example



Example of Quick Sort...

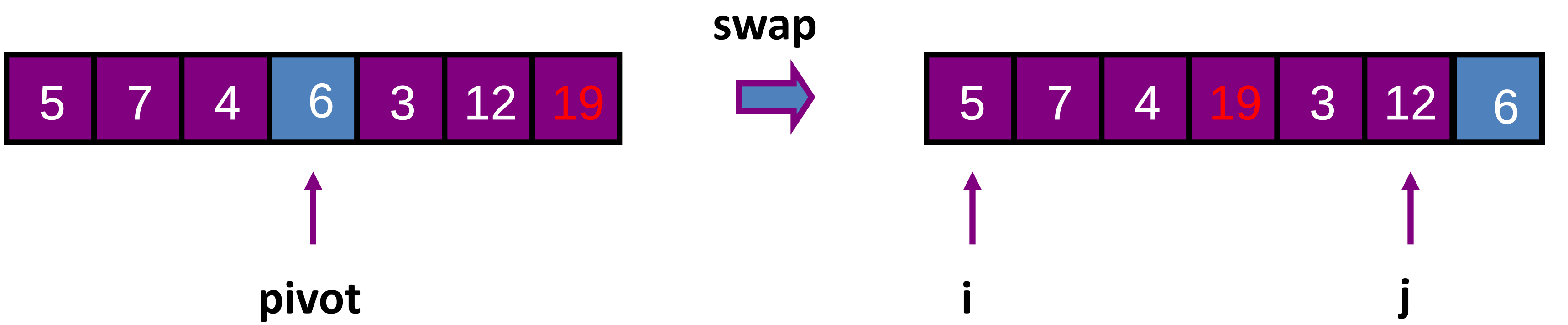


Issues To Consider

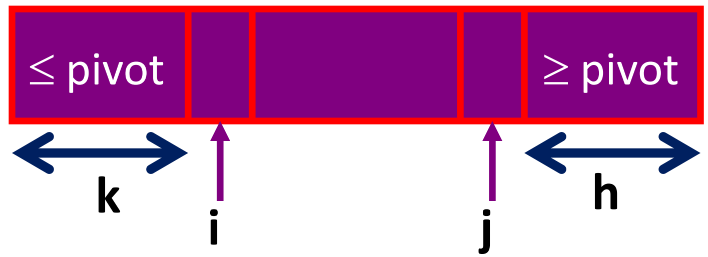
- **How to pick the pivot?**
  - Many methods (discussed later)
- **How to partition?**
  - Several methods exist.
  - The one we consider is known to give good results and to be easy and efficient.
  - We discuss the partition strategy first.

Partitioning Strategy

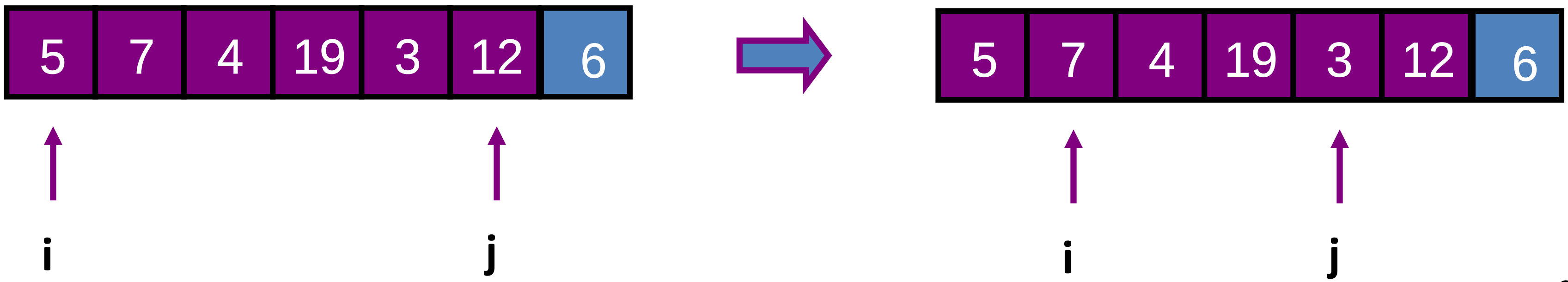
- We want to partition array A[left .. right].
- For now, assume that ***pivot*** *= A[(left+right)/2]*.
- **First**, get the ***pivot*** element out of the way by swapping it with the last element (swap pivot and A[right]).
- Let **i** start at the first element and **j** start at the next-to-last element *(i = left, j = right – 1)*



Partitioning Strategy

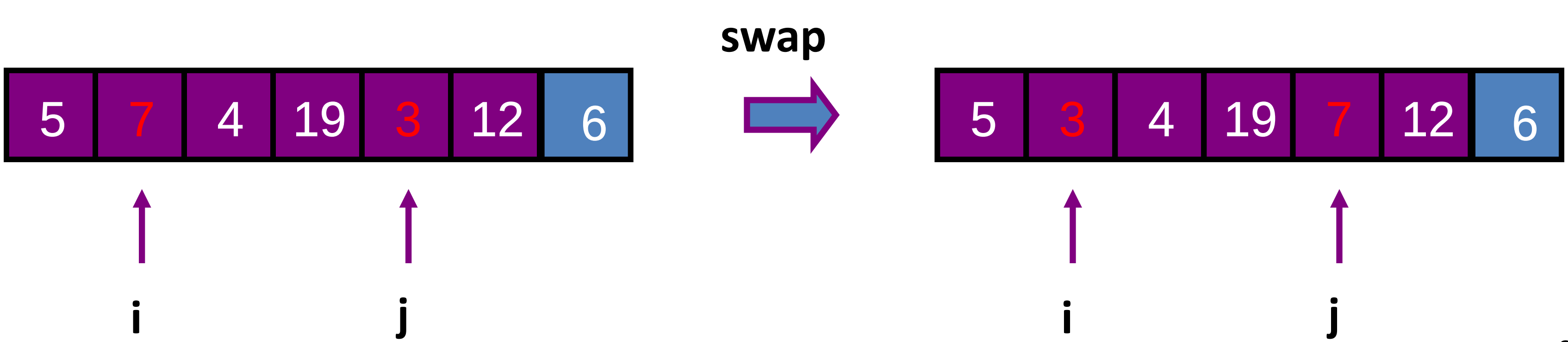


- Want to have
  - A[k] ≤ pivot, for k < i
  - A[h] ≥ pivot, for h > j
- When i < j
  - Move **i** right, **skipping** over elements **smaller** than the pivot
  - Move **j** left, **skipping** over elements **greater** than the pivot
  - When both **i** and **j** have stopped
    - A[i] ≥ pivot
    - A[j] ≤ pivot ⇒ A[i] and A[j] should now be swapped



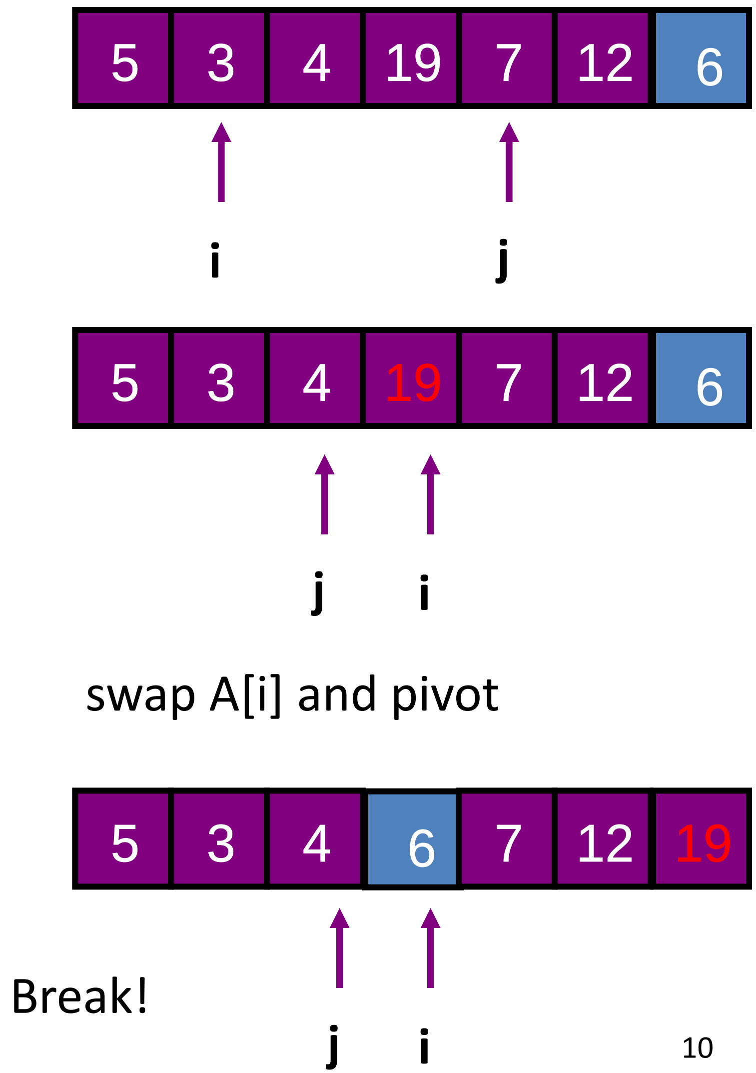
Partitioning Strategy (2)

- When **i** and **j** have stopped and **i** is to the left of **j** (thus legal)
  - Swap **A[i]** and **A[j]**
    - The large element is pushed to the right and the small element is pushed to the left
  - After swapping
    - $A[i] \leq$ pivot
    - $A[j] \geq$ pivot
  - Repeat the process until **i** and **j** <u>**cross**</u>



Partitioning Strategy (3)

- When **i** and **j** have crossed
  - **swap A[i] and pivot**
- Result:
  - A[k] ≤ pivot, for k < i
  - A[h] ≥ pivot, for h > j



Picking the Pivot

- There are several ways to pick a pivot.
- Objective: Choose a pivot so that we will get 2 partitions of (almost) equal size.

Picking the Pivot (2)

- Use the **first element** as pivot
  - if the input is random, ok.
  - if the input is presorted (or in reverse order)
    - all the elements go into S_2 (or S_1).
    - this happens consistently throughout the recursive calls.
    - results in $O(N^2)$ behavior (we analyze this case later).
- Choose the pivot **randomly**
  - generally safe,
  - but random number generation can be **expensive** and does not reduce the running time of the algorithm.

Picking the Pivot (3)

- Use the **median** of the array (ideal pivot)
  - The $\lceil N/2 \rceil$ *th* largest element
  - Partitioning **always** cuts the array into roughly **half**
  - An **optimal** quick sort (O(N log N))
  - However, hard to find the exact median

- **Median-of-three partitioning**
  - eliminates the bad case for sorted input.
  - reduces the number of comparisons by 14%.

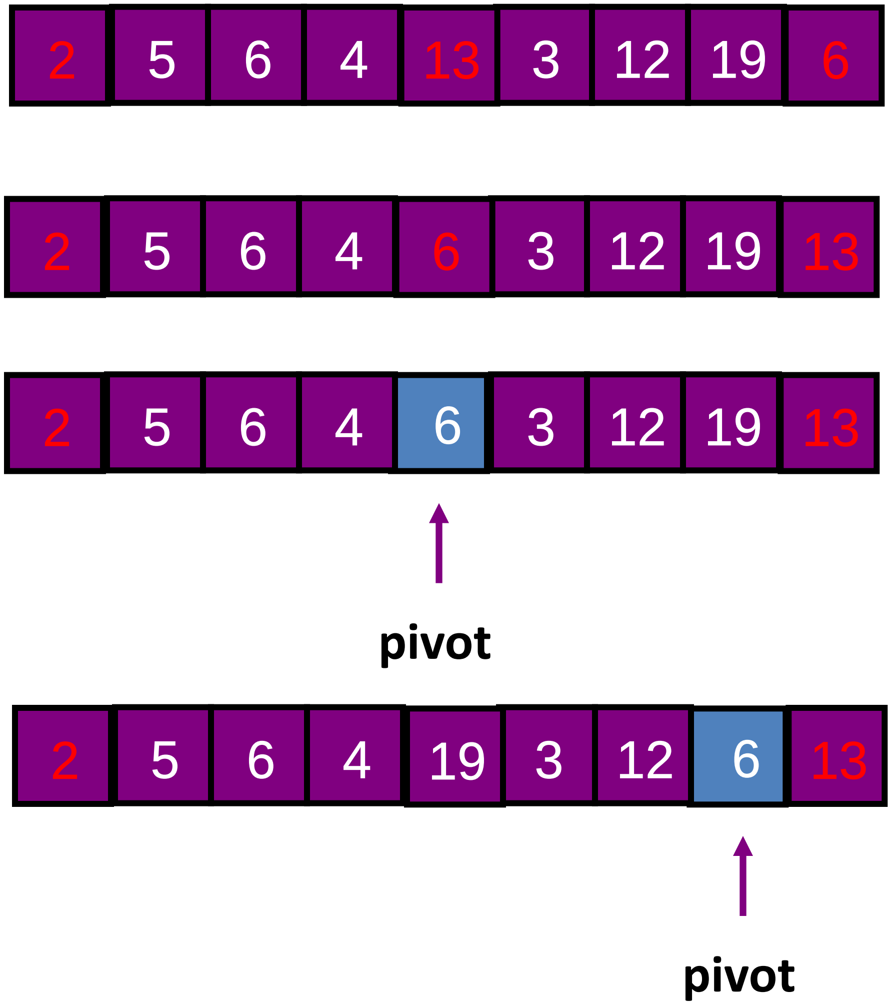
Median of Three Method

- Compare just three elements: the leftmost, rightmost 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 the pivot is at the second last position **(why?)**

```
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 ] );
```

Median of Three: Example



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]

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

Quick Sort Summary

- **Recursive case**: QuickSort(a, left, right)

  pivot = median3(a, left, right);

  Partition a[left … right] into a[left … i-1], i, a[i+1 … right];

  QuickSort(a, left, i-1);

  QuickSort(a, i+1, right);

- **Base case**: when do we stop the recursion?
  - In theory, when left >= right.
  - In practice, …

Small Arrays

- For very small arrays, quick sort **does not** perform as well as insertion sort
- Do not use quick sort recursively for small arrays
  - Use a sorting algorithm that is efficient for small arrays, such as insertion sort.
- When using quick sort recursively, switch to insertion sort when the sub-arrays have between 5 to 20 elements (10 is usually good).
  - saves about 15% in the running time.
  - avoids taking the median of three when the sub-array has only 1 or 2 elements.

Quick Sort: Pseudo-code

```
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

- The partitioning code we just saw works only if pivot is picked as median-of-three.
  - A[left] ≤ pivot and A[right] ≥ pivot
  - Need to partition only
    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;
}
```

Homework

- Assume the pivot is chosen as the middle element of an array: pivot = a[(left+right)/2].
- Rewrite the partitioning code and the whole quick sort algorithm.

Quick Sort Faster Than Merge Sort

- Both quick sort and merge sort take O(N log N) in the average case.
- But quick sort is faster in the average case:
  - The inner loop consists of an increment/decrement (by 1, which is fast), a test and a jump.
  - There is no extra juggling as in merge sort.

```
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;
}
```

inner loop

Analysis

Assumptions:

- A random pivot (no median-of-three partitioning)
- No cutoff for small arrays (to make it simple)

1. If the number of elements in S is 0 or 1, then return (base case).
2. Pick an element **v** in **S** (called the **pivot**).
3. Partition the elements in **S** except **v** into two disjoint groups:
   1. $S_1 = \{x \in S - \{v\} \mid x \leq v\}$
   2. $S_2 = \{x \in S - \{v\} \mid x \geq v\}$
4. Return {QuickSort(S_1) + v + QuickSort(S_2)}

Analysis (2)

- Running time
  - pivot selection: constant time, i.e. $O(1)$
  - partitioning: linear time, i.e. $O(N)$
  - running time of the two recursive calls

- $T(N) = T(i) + T(N - i - 1) + cN$
  - i: number of elements in $S1$
  - c is a constant

Worst-Case Scenario

- What will be the **worst case**?
  - The pivot is the smallest element, all the time
  - 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 Scenario

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

- $T(N) = T(N/2) + T(N/2) + cN = 2T(N/2) + cN$

- This recurrence is similar to the merge sort recurrence.
- The result is $O(N\log N)$.

Average-Case Analysis

- Assume that each of the sizes for S_1 is equally likely $\Rightarrow$ has probability 1/N.
- This assumption is valid for the pivoting and partitioning strategy just discussed (but may not be for some others),
- On average, the running time is O(N log N).
- Proof: pp 272–273, Data Structures and Algorithm Analysis by M. A. Weiss, 2nd edition

Next week …

- Arrays (review) and Linked Lists (3.2, 3.3)
- Stacks, queues (Chapter 5)