Quicksort

Weiss, from section 8.6 to the end of chapter 8 (pages 364–391)
Quicksort

• Mergesort is an *out-of-place* sorting (it creates an auxiliary array).
• Divide an array into two sub-arrays such that every small element precedes every large element. Then, sort the sub-arrays.
  ▪ Sort sub-arrays before merging them.
  ▪ Sorting does not require the auxiliary array.

• Determine a value that ends up to be the median.
• It does not work out if you sort the array to find the median.
• A computer scientist suggested to choose a *random* element to be a simulated median. This element is called the *pivot*. 
Quicksort

• Consider $A$ as an array of integers that are randomly arranged.

$$A = A[0],..., A[N-1]$$

• Suppose that middle is an index in middle of an array. $A[middle]$ is the pivot.

• Get the pivot out of way by swapping it with the last element.
Partitioning

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>right</td>
<td>Last (pivot)</td>
</tr>
</tbody>
</table>

c. Swap A[left] and A[right].
   • A[last] ≤ A[right], A[right+1], ..., A[last-1]
d. Repeat steps a and b until all elements of the array are checked.
Example

5 3 8 4 7 1 2 9 6

5 3 8 4 6 1 2 9 7

5 3 8 4 7 1 2 9 6

5 3 8 4 6 1 2 9 7

middle

last

left

left

right
public static <AnyType extends Comparable<? super AnyType>>
void quicksort(AnyType[] a) {
    quicksort(a, 0, a.length - 1);
}

private static <AnyType extends Comparable<? super AnyType>>
void quicksort(AnyType[] a, int low, int high) {

    int middle = (low + high) / 2;
    swapReferences(a, middle, high);
    AnyType pivot = a[high];

    int i, j;
    for (i = low; j = high - 1; ; ) {
        while (a[i++].compareTo(pivot) < 0);
        while (pivot.compareTo(a[j--]) < 0);
        if (i ≥ j) break;
        swapReferences(a, i, j);
    }
    swapReferences(a, i, high);
    quicksort(a, low, i-1);
    quicksort(a, i+1, high);
}
Picking Pivot

• If the array is sorted or reverse sorted, choosing the first element gives quadratic time.
• It is because these pivots generate unevenly sized sub-arrays.

1 2 3 4 5 6 7 8 9

• It is suggested to choose the pivot to be the *median* of the first, middle, and the last elements (median-of-three partitioning).
Time Complexity

• In the best case, array is partitioned into equally sized sub-arrays.
• It takes linear time to check the elements of the array.
  
  \[ T(n) = n + 2T(n/2) \]
  
  \[ T(n) \in O(n \log n) \]

• In the worst case, array is partitioned into extremely unequal sub-arrays: sub-arrays of size 0 and n-1
  
  \[ T(n) = n + T(n - 1) + T(0) \]
  
  \[ T(n) = n + ((n - 1) + T(n - 2)) \]
  
  \[ T(n) = 2n - 1 + T(n - 2) \]
  
  \[ T(n) = 2n - 1 + ((n - 2) + T(n - 3)) \]
  
  \[ T(n) = 3n - 3 + T(n - 3) \]
  
  \[ T(n) = kn - k + T(n - k) \]
  
  Recurrence stops when \( k = n \) → \( T(n) = n^2 - n \)
  
  \[ T(n) \in O(n^2) \]
Small Arrays

- For small arrays, recursive calls of quicksort have overhead.  
  ➤ Use insertion sort when array size is smaller than a cutoff.
- The running time of combining quicksort with insert sort is $O(n \log n)$.  

**Insertion Sorting**

<table>
<thead>
<tr>
<th>Array Position</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial State</td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>After a[0..1] is sorted</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>After a[0..2] is sorted</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>After a[0..3] is sorted</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>After a[0..4] is sorted</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>After a[0..5] is sorted</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

```java
public static <AnyType extends Comparable<? super AnyType>>
void insertionSort( AnyType[] a )
{
    for( int p = 1; p < a.length; p++ )
    {
        AnyType tmp = a[ p ];
        int j = p;

        for( ; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
```

Worst case runtime is $O(N^2)$