• Merge Sort is an easy-split, hard-join method.
- split() is trivial.
  public int split(int[] A, int lo, int hi) {
    return (lo + hi + 1)/2;
  }

- join() merges two smaller, sorted arrays.
  * Specifically, it merges A[lo:s-1] and A[s:hi] into _tempA[lo:hi], then copies _tempA back to A.
  public void join(int[] A, int lo, int s, int hi) {
    merge(A, lo, s, hi);
    for (int i = lo; i <= hi; i++) {
      A[i] = _tempA[i];
    }
  }
private void merge(int[] A, int lx, int mx, int rx)
{
    int i = lx;
    int j = mx;

    for (int k = lx; k <= rx; k++) {
        if ((i < mx) && (j <= rx)) {
            if (A[i] < A[j])
                _tempA[k] = A[i++];
            else
                _tempA[k] = A[j++];
        }
        else if (i < mx) {
            _tempA[k] = A[i++];
        }
        else if (j <= rx) {
            _tempA[k] = A[j++];
        }
    }
}
• Merge Sort takes $O(n \log n)$ steps.

  - Because each `split()` divides the array into two (almost) equal-sized parts, each element is `join()`ed $\log n$ times.
Quick Sort

- Quick Sort is a **hard-split, easy-join** method.

- The following diagram illustrate one step.
public int split(int[] A, int lo, int hi) {
    int key = A[lo];
    int lx = lo; // left index.
    int rx = hi; // right index.

    // Invariant 1: key <= A[rx+1:hi].
    // Invariant 3: there exists ix in [lo:rx]
    //                      such that A[ix] <= key.
    // Invariant 4: there exists jx in [lx:hi]
    //                      such that key <= A[jx].

    while (lx <= rx) {
        while (key < A[rx]) { // will terminate due to invariant 3.
            rx--;
            // Invariant 1 is maintained.
        }
while (A[lx] < key) {  // will terminate due to invariant 4.
    lx++;              // Invariant 2 is maintained.

if (lx <= rx) {         // swap A[lx] with A[rx]:
    int temp = A[lx];
    A[rx] = temp;      // invariant 4 is maintained.
    rx--;             // invariant 1 is maintained.
    lx++;             // invariant 2 is maintained.
}


return lx;
}
public void join(int[] A, int lo, int s, int hi) {
    // nothing to do!
}
Quick Sort (cont.)

- If the pivot chosen by `split()` divides the array into two (almost) equal-sized parts, each element is `split()` \( \log n \) times.

- Thus, in this case, Quick Sort takes \( O(n \log n) \) steps.
Quick Sort (cont.)

- On the other hand, an unfortunate choice of the pivot could divide the array into two parts, one that contains no elements and another that contains \(n - 1\) elements.

- In this case, Quick Sort takes \(O(n^2)\) steps.
Quick Sort (cont.)

- Various strategies are used to choose the pivot. (None is perfect.)
  - Pick the first element (worst-case scenario is a nearly-sorted or nearly-inverse-sorted array).
  - Take the median of the first, last, and middle elements. This is often used in practice, since it behaves well on the nearly-sorted case, which can be quite common in some applications.
## Summary

<table>
<thead>
<tr>
<th>Sort</th>
<th>Best-Case Cost</th>
<th>Worst-Case Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Merge</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quick</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

where $n$ is the size of the container