## **QuickSort**

Go to <http://math.hws.edu/eck/js/sorting/xSortLab.html> to watch the sorts in action.

Quick Sort is another O(n log n) sort. Its general strategy is to select a *pivot* (or a *partition*)and move every smaller item to the left of the pivot and every larger item to its right. Then we call Quick Sort on the left side and the right side, recurring until we are done.

private static void sort(double[] array, int first, int last)  
{  
 int splitPt;  
 if (first < last) // General case  
 {   
 splitPt = rearrange(array, first, last);  
 sort(array, first, splitPt - 1); // sort left side  
 sort(array, splitPt + 1, last); // sort right side  
 }  
}

Normally the code generates a tree structure, with the rearrange method at each branch running in linear Or O(n) time. That general arrangement should remind you of the MergeSort.

1. Let's run one pass of the rearrange method on the toy array below. For this teaching example, we will let the pivot be 0 and already in the middle, as shown. We want to rearrange the array so that all negative numbers appear to the left of 0 and the positive numbers appear to the right of 0. We keep track of two indexes, one starting at the far left and the other at the far right. Compare the left value to 0. If smaller, move over. Compare the right value to 0. If larger, move over. Keep comparing and moving until you find two values that are each on the “wrong side.” Then swap that pair. Repeat. Show the contents of the array after one pass:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| -1 | 2 | 1 | -4 | 0 | -6 | 3 | -2 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

Look at the result of one pass of the rearrangemethod. All negatives are to the left and all positives are to the right. The 0 is in its correct place, and never has to be compared or moved again. Fill in the blanks in this code, which is part of the rearrange method.

while (first <= last)  
 {   
 if (array[first] <= pivot) //if it's on the correct side,   
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ // move right   
 else if (array[last] >= pivot) //if it's on the correct side,   
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ // move left   
 else //if both on the wrong side,   
 {   
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ // then swap them,  
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ // update both right and left  
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 }

}

# The pivot above was the value 0. We will use a version of the Quick Sort in which the pivot-value is whatever happens to be the value at the first index. We look for pairs of values that each are “on the wrong side” and swap them. In the example below, the pivot is 9. You have to imagine that the 9 is going to be in the middle somewhere, and all values that are smaller than 9 will be on the left and all values that are greater than 9 will be on the right. After all the “on the wrong side” pairs have been swapped, and the indices meet each other, one last swap puts the pivot 9 at that index-meeting place.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 20 | 3 | 5 | 60 | 6 | 14 | 11 |
| *first = pivot* |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

1. Then recur on the left side of the array and recur on the right side of the array, running the rearrange method on each side:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 6 | 3 | 9 | 60 | 20 | 14 | 11 |
|  |  |  | 9 |  |  |  |  |

1. Assuming that the rearrange method uses the *first item* as the pivot, display the contents of array Arr after the call rearrange(Arr, first, last);

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 18 | 8 | 4 | 11 | 7 | 6 | 3 | 10 | 1 | 5 | 20 |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |

Arr

1. The pivot does not have to be the first. The pivot could be (first+last)/2. If that is the case, show one pass of the rearrange method on this data:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 5 | 12 | 3 | 7 | 8 | 5 | 2 |
|  |  |  |  |  |  |  |

1. We can reason intuitively about the Big-O for Quick Sort. If we have large arrays of random data, each pivot (the first item) will probably cut the array near the middle. (You have seen that before in the Binary Search.) Therefore, for *n* items of data (in random order), how many levels of recursion do you need? \_\_\_\_\_\_\_\_ What is the Big-O of each rearrange method (see above)? \_\_\_\_\_\_\_ Therefore, what is the Big-O of the Quick Sort for random data? \_\_\_\_\_\_\_\_\_\_

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

# The Quick Sort has a worst case in which the Big-O degenerates to O(n2). Here is an array already in sorted order (ascending order). How many levels (recursive calls) does the Quick Sort make to sort this array in ascending order?  \_\_\_\_\_ How many comparisons does it do in each level and in total?\_\_\_\_\_\_ What is the Big-O in this case? \_\_\_\_\_\_

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

# For similar reasons, the Big-O for an array in reverse order (descending order) is also O(n2). How many levels (recursive calls) does the Quick Sort make to sort this array in ascending order?  \_\_\_\_\_ How many comparisons does it do in each level and in total?\_\_\_\_\_\_ What is the Big-O in this case? \_\_\_\_\_\_ Indeed, the QuickSort in this case behaves just like a Selection Sort. Try it and see.

# Know thy Big-O: the Quick Sort is \_\_\_\_\_\_\_\_\_\_\_ in the best case, \_\_\_\_\_\_\_\_\_ in the average case, and \_\_\_\_\_\_\_\_\_\_ in the worst case, i.e., the case of the "bad pivot."

# **Lab Assignment**

QuickSort\_Driver is written for you. Fill in the blanks in the QuickSort class for doubles. Then complete the code to sort Comparables.