# CS 260
# Quicksort

## Quicksort

Quicksort is a recursive sort that functions by splitting the data into smaller pieces by portioning. It continues this portioning until the data is complete sorted. It is not a stable sort due to the moving of data that occurs during the portioning phase.

Quick sort is order N log N and does not require extra memory for data, although it does require extra stack space due to recursion. Its worst-case performance is unfortunately O(N^2), like insertion, selection, or bubble sort. It works well with data that is already partially sorted.

Quick sort is like merge sort in that the algorithm divides the array is into subarrays recursively and then sorts each of the subarrays. It does not require a merge step since items are swapped in place.

The concept behind quick sort is partitioning an array into two parts using a pivot value. All values less than the pivot are moved to one side and all values greater than or equal to the pivot are moved to the other side. Then the pivot is placed at the index between these two partitions. It is now in its final location, where it would be in a sorted array. This is then recursively repeated on each partition until the array is fully sorted.

## Example

Here is a simple example of using quicksort.

The array before starting is shown below.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 22 | 10 | 2 | 11 | 13 | 16 | 12 | 18 | 5 | 9 | 8 | 6 | 19 | 3 | 14 | Starting values |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | Index or position |

The leftmost value in the array, 7, is picked as pivot, the array is partitioned around it, and it is placed in its final location.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 3 | 6 | 2 | **7** | 13 | 16 | 12 | 18 | 11 | 9 | 8 | 10 | 19 | 22 | 14 | Pivot == 7 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | Index or position |

After partitioning, all elements less than 7 are to its left, and all elements larger than 7 are to its right. The exact placement of elements depends on the partitioning algorithm, but the pivot will be placed where it should be in a properly sorted array.

Then repeat this on the left side. Use 5 as the pivot since it is in the smallest index on that side.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | **5** | 6 | **7** | 13 | 16 | 12 | 18 | 11 | 9 | 8 | 10 | 19 | 22 | 14 | Pivot == 5 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | Index or position |

And on the right side. Use 13 as the pivot since it is in the smallest index on that side.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | **5** | 6 | **7** | 9 | 10 | 12 | 8 | 11 | **13** | 18 | 16 | 19 | 22 | 14 | Pivot == 13 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | Index or position |

Continue partitioning subarrays until all values are in sorted order (all sub arrays are of length one). Each time pick the pivot as the item with the smallest index in the subarray being partitioned.

## Pseudo code

Two functions are required; recQuickSort recursively partitions each of the subarrays using partition to do the work of partitioning. There are many algorithms for partitioning, this is a fairly simple one.

```
// recursive program that does quicksort
recQuickSort(theArray, first, last)

        // if the array is longer than length 1
        // partition it, then QuickSort each sub-array
    if (first < last)

            // partition, returns index of properly placed pivot
        pivot = partition(theArray, first, last)

            //now recursively sort each sub-array
        recQuickSort(theArray, first, pivot-1)
        recQuickSort(theArray, pivot+1, last)

// simple partition.
// Uses the first location for pivot
//   better would be to use median of first, last, middle
//
// moves elements less than pivot to beginning of array
//   change comparision operator to order largest to smallest
//
// more efficient would use pointers from each side
//
// returns index of placed pivot
partition(theArray, first, last)

    pivotIndex = first
    pivotElement = theArray[first]

    int index = first + 1

       // Move items less than pivot to lower indices
    while (index <= last)
        if (theArray[index] <= pivotElement)

            pivotIndex += 1
            swap(theArray[index], theArray[pivotIndex])

        index += 1

             // Place pivot in the proper place
    swap(theArray[pivotIndex], theArray[first])

    return pivotIndex
```

Again, this algorithm returns its sorted array via the parameters, so extra care needs to be taken when coding in Python.