**Algorithm Study Template**

**Algorithm**: Quicksort

**aka**: 2-Way Partitioned In-Place Quicksort

**Techniques**: Divide and conquer

**Categories**: Sorting

**Problem**: Historically, quicksort was written for a machine-aided language translation project in the Soviet Union, where the sort was used on words to be translated. This made the words easier to match to an already-sorted Russian-to-English dictionary which was stored on magnetic tape and prevented the user from needing to jump back and forth between sections of the dictionary while translating.

**Applications**: Quicksort is a very popular sorting algorithm, and is often used to sort large datasets of varying types, often as a precondition for searches. Specifically, when searching for a unique integer key in a file of unsorted records, one could use quicksort to put the keys in sorted order, as well as a binary search to find the specific key requested.

**References**:

* <http://www.sorting-algorithms.com/quick-sort>
* <http://algs4.cs.princeton.edu/lectures/23Quicksort.pdf>
* <http://algs4.cs.princeton.edu/23quicksort/>
* <http://xlinux.nist.gov/dads/HTML/quicksort.html>
* <http://www.algolist.net/Algorithms/Sorting/Quicksort>
* Algorithms, 4th Edition, by Sedgewick and Wayne, pages 288-295
* Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Pages 113–122 of section 5.2.2: Sorting by Exchanging.

**Implementation details**:

* **Big Idea**: Create a pivot point at the start of the list, partition the list into two halves, move pivot to the center of the list, and sort the two halves of the list recursively.
* **Description**: Set the value at the beginning of the list as the pivot point and then set two indices, left and right, as the second and last values of the list, respectively. Then, move the left index forward through the list until it references a value greater than pivot and move the right index backward through the list until it references a value less than pivot. At this point, if the value referenced at the left index is smaller than the value referenced at the right index, swap the two values. If not, swap the value at the beginning of the list (which is pivot) with the value at the right index and then keep sorting recursively with two sub-lists. The pivot, formerly at the beginning of the list, is now considered the center of the list for the purpose of creating sub-lists. The first sub-list goes from the beginning of the primary list to the value preceding the pivot (which is now at the center). The second sub-list goes from the first value following the pivot (center) to the end of the primary list. It is also worth noting that the sort is done completely in place, meaning that the sorting is done inside the array of unsorted values, rather than creating new arrays for the sorted values and sub-lists.
* **Pseudo-code**: (adapted from <http://en.wikipedia.org/wiki/Quicksort>)

\*(note that the left and right values being passed in correspond to the beginning and ending indices of the list)

**function** quicksort(array, left, right)

*// If the list has 2 or more items*

**if** left < right{

set pivotIndex to beginning of array

*// Get lists of bigger and smaller items and final position of pivot*

left index := left + 1

right index := right

while(left index <= right index){

//while value at left index < pivot

while((left index <= right) && (list[left index] <= pivot)){

++left index

}

//while value at right index > pivot

while((right index >= left) && (list[right index] > pivot)){

--right index

}

//left and right indices may need to be swapped

if(left index < right index){

swap(array, left index, right index)

}

}

}

//move pivot to final position

swap(array, left, right index)

*// Recursively sort elements smaller than the pivot*

quicksort(array, left, right index - 1)

*// Recursively sort elements at least as big as the pivot*

quicksort(array, right index + 1, right)

* **Specific implementation**: (see QuickSorts.java)

**Correctness**:

**Theoretical**: (adapted from <http://www.algolist.net/Algorithms/Sorting/Quicksort> ) Quicksort divides the array into two parts and every element **a** from the left part is less or equal to every element **b** from the right part. Also, **a** and **b** satisfy the **a ≤ pivot ≤ b** inequality. After completion of the recursive calls, both parts of the original array become sorted and, due to effective partitioning based on the original pivot value, the whole array is now sorted.

**Empirical**: Because I tested with reasonably large arrays of random integers, outputting the results for visual verification of sorting was not a reasonable option. Instead, I wrote a boolean testSort method to ensure that each sorted array is in ascending order. The method compares each value in the array to the value after it and sets a flag to false if the value at the lower index is greater than the value at the higher index. The method then returns true or false to indicate whether the sort succeeded or failed. If the sort test were to fail, the user would be notified and the program would terminate.

The user inputs an integer to specify the size of the array, which is to be filled with random integer values and then sorted in ascending order. The program uses a try-catch statement and a few if statements to ensure that input values are integers and are greater than zero. Thus, negative list sizes are not allowed, and empty lists are not allowed either. However, lists with only one value are permitted. If the program receives unacceptable input, the user is notified and the program terminates.

I tested with one thousand, ten thousand, one hundred thousand, one million, ten million, one hundred million, and one hundred fifty million values. I had intended to test with even more values, but unfortunately my computer ran out of memory and prevented me from doing so. The sort test never failed and the following information was output to the user each time to indicate the success of the sort (specific number of integers removed for generality):

The sorted order of the list has been verified.

All <number> integers have been placed in sorted order.

**Performance**:

**Theoretical**: (via <http://en.wikipedia.org/wiki/Quicksort>) The average performance is O(n log n) and worst case is O(n2), but the worst case is considered to be extremely rare for most “typical” datasets.

**Empirical**: Execution times are given in milliseconds and seconds, and swaps are counted for comparison to other sorts. Only the algorithm itself is timed, no printed output or other “noise” is intentionally included.

1,000 values:

* Quicksort with 2-Way Partitioning completed in 1.217 milliseconds, which is also 0.001217 seconds.
* 150 swaps were performed.

10,000 values:

* Quicksort with 2-Way Partitioning completed in 25.134 milliseconds, which is also 0.025134 seconds.
* 1257 swaps were performed.

100,000 values:

* Quicksort with 2-Way Partitioning completed in 41.759 milliseconds, which is also 0.041759 seconds.
* 19282 swaps were performed.

1,000,000 values:

* Quicksort with 2-Way Partitioning completed in 119.946 milliseconds, which is also 0.119946 seconds.
* 201465 swaps were performed.

10,000,000 values:

* Quicksort with 2-Way Partitioning completed in 1308.783 milliseconds, which is also 1.308783 seconds.
* 1728904 swaps were performed.

100,000,000 values:

* Quicksort with 2-Way Partitioning completed in 14547.325 milliseconds, which is also 14.547325 seconds.
* 16901419 swaps were performed.

150,000,000 values:

* Quicksort with 2-Way Partitioning completed in 22447.384 milliseconds, which is also 22.447384 seconds.
* 15785471 swaps were performed.

**Anecdotes**: Most programmers consider quicksort to be the best sorting algorithm in terms of performance.

**History**: (via <http://en.wikipedia.org/wiki/Quicksort>) Quicksort was developed in 1960 by Tony Hoare while in the Soviet Union as a visiting student at Moscow State University. At the time, Hoare was working on a machine-aided language translation project for the National Physical Laboratory. The sort was used on words to be translated. This made the words easier to match to an already-sorted Russian-to-English dictionary which was stored on magnetic tape and prevented Hoare from needing to jump back and forth between sections of the dictionary while translating.

**Variations**: There are many variations of quicksort. Most notably, the two-way partitioned implementation which I utilized here could also be done without in-place sorting. By that, I mean that I could have used separate arrays for the sub-lists, rather than sorting everything inside of the original array. Another variation is the three-way partitioned implementation which is fully detailed later in this report. Others include, but are not limited to: balanced quicksort, external quicksort, and quick radix sort.

**Alternatives**: Other popular sorting algorithms include bubble sort, selection sort, insertion sort, and shell sort. The other sorting algorithm that is most often favorably compared to quicksort in terms of performance is the merge sort.

**Credits:**

* <http://en.wikipedia.org/wiki/Quicksort>
* <http://www.sorting-algorithms.com/static/QuicksortIsOptimal.pdf>
* <http://rosettacode.org/wiki/Sorting_algorithms/Quicksort>

**Algorithm Study Template**

**Algorithm**: Multi-key Quicksort

**aka**: Three-Way Partitioned In-Place Quicksort or Three-Way Radix Quicksort

**Techniques**: Divide and conquer

**Categories**: Sorting

**Problem**: A two-way partitioned quicksort has acceptable performance when large numbers of duplicate keys appear, but there is a substantially better way to handle this case. When large numbers of duplicate keys appear, the recursive nature of the two-way partitioned quicksort works against the user by continuing to partition the array into smaller sub-arrays even when all the values inside those sub-arrays are equal. The multi-key quicksort does a better job by handling items with the same value as the pivot point as a separate partition.

**Applications**: This version of quicksort is useful when large numbers of duplicate keys are expected during sorting. For example, when sorting a string where certain characters appear often (perhaps ‘T’ or ‘S’), the multi-key quicksort outperforms a two-way partitioned quicksort by excluding keys which are identical to the pivot point from recursively sorted sub-lists.

**References**:

* Algorithms, 4th Edition, by Sedgewick and Wayne, pages 297-301
* <http://www.sorting-algorithms.com/quick-sort-3-way>
* <http://stackoverflow.com/questions/941447/quicksort-with-3-way-partition>
* <http://algs4.cs.princeton.edu/23quicksort/>
* <http://xlinux.nist.gov/dads/HTML/multikeyQuicksort.html>

**Implementation details**:

* **Big Idea**: Set a pivot point at the beginning of the unsorted array and partition the array into three parts, one for indices with values smaller than the pivot point, one for indices with values larger than the pivot point, and one for indices with values equal to the pivot point. Then, recursively sort the partitions with values either larger or smaller than the pivot point.
* **Description**: Begin by setting a pivot point at the beginning of the unsorted array. Then, set two indices, left and right, at the beginning and end of the array, respectively. Next, establish another index, i, at the second position in the array. Now, iterate through the array and compare the value at array[i] to the value at the pivot point. If the value at array[i] is smaller, swap the value at array[i] with the value referenced by the left index and increment both i and the left index. If the value at array[i] is larger, swap the value at array[i] with the value referenced by the right index and decrement the right index. If the value at array[i] is equal to the value of the pivot point, simply increment i and keep moving through the array. Once all three partitions (greater than, less than, and equal to the pivot point) have been put in place, the values less than or greater than the pivot point are recursively sorted as sub-lists.
* **Pseudo-code**:

\*(note that the left and right values being passed in correspond to the beginning and ending indices of the list)

**function** quicksort(array, left, right)

*// If the list has 2 or more items*

**if** left < right{

set pivotIndex to beginning of array

*// Find bigger, smaller, and equal items and final position of pivot*

left index = left

i = left + 1

right index = right

//while swaps are possible

while(i <= right index){

//if value at array[i] is smaller than left index, swap with left index

if(array[i] < value at pivotIndex){

swap(list,left index++,i++)

//if value at list[i] is larger than left index, swap with right index

}else if(array[i] > value at pivotIndex){

swap(list,i,right index--)

//if array[i] is equal to left index, increment i

}else{

i++;

}

}

*// Recursively sort elements smaller than the pivot*

quicksort(array, left, left index - 1)

*// Recursively sort elements at least as big as the pivot*

quicksort(array, right index + 1, right)

* **Specific implementation**: (see QuickSorts.java)

**Correctness**:

**Theoretical**: This version of quicksort uses a pivot point to find values equal to itself and place them in the center of the array, with values less than pivot ending up before this center space and values greater than pivot ending up after the center space. Those two spaces before and after the center are then recursively sorted in the same manner, eventually resulting in a fully sorted array.

**Empirical**: Because I tested with reasonably large arrays of random integers, outputting the results for visual verification of sorting was not a reasonable option. Instead, I wrote a boolean testSort method to ensure that each sorted array is in ascending order. The method compares each value in the array to the value after it and sets a flag to false if the value at the lower index is greater than the value at the higher index. The method then returns true or false to indicate whether the sort succeeded or failed. If the sort test were to fail, the user would be notified and the program would terminate.

The user inputs an integer to specify the size of the array, which is to be filled with random integer values and then sorted in ascending order. The program uses a try-catch statement and a few if statements to ensure that input values are integers and are greater than zero. Thus, negative list sizes are not allowed, and empty lists are not allowed either. However, lists with only one value are permitted. If the program receives unacceptable input, the user is notified and the program terminates.

I tested with one thousand, ten thousand, one hundred thousand, one million, ten million, one hundred million, and one hundred fifty million values. I had intended to test with even more values, but unfortunately my computer ran out of memory and prevented me from doing so. The sort test never failed and the following information was output to the user to indicate the success of the sort (specific number of integers removed for generality):

The sorted order of the list has been verified.

All <number> integers have been placed in sorted order.

**Performance**:

**Theoretical**: (via <http://en.wikipedia.org/wiki/Quicksort>) The best case is O(KN) and the worst case is O(2KN) or at least O(N2).

**Empirical**: Execution times are given in milliseconds and seconds, and swaps are counted for comparison to other sorts. Only the algorithm itself is timed, no printed output or other “noise” is intentionally included.

1,000 values:

* Quicksort with 3-Way Partitioning completed in 1.928 milliseconds, which is also 0.001928 seconds.
* 998 swaps were performed.

10,000 values:

* Quicksort with 3-Way Partitioning completed in 6.046 milliseconds, which is also 0.006046 seconds.
* 9999 swaps were performed.

100,000 values:

* Quicksort with 3-Way Partitioning completed in 17.12 milliseconds, which is also 0.01712 seconds.
* 99998 swaps were performed.

1,000,000 values:

* Quicksort with 3-Way Partitioning completed in 136.903 milliseconds, which is also 0.136903 seconds.
* 999998 swaps were performed.

10,000,000 values:

* Quicksort with 3-Way Partitioning completed in 1569.433 milliseconds, which is also 1.569433 seconds.
* 9999997 swaps were performed.

100,000,000 values:

* Quicksort with 3-Way Partitioning completed in 17801.024 milliseconds, which is also 17.801024 seconds.
* 99999997 swaps were performed.

150,000,000 values:

* Quicksort with 3-Way Partitioning completed in 27217.355 milliseconds, which is also 27.217355 seconds.
* 149999997 swaps were performed.

**Anecdotes**: <none>

**History**: This version of multi-key quicksort was created by E.W. Dijkstra. He popularized it as a classical programming exercise called the Dutch National Flag problem because it is similar to sorting an array with three possible key values, which might correspond to the three colors on the flag. The multi-key quicksort he developed was his solution to the classical problem.

**Variations**: In the 1990’s, Bentley and McIlroy developed a solution which performs better than Dijkstra’s. It is known as Bently-McIlroy Three-Way Partitioning, and is better at handling cases where the number of duplicate keys could be high or low.

**Alternatives**: Other versions of quicksort (two-way partitioned, balanced quicksort, external quicksort, etc.) could sort the same datasets as the multi-key quicksort with identical results, but poorer performance when the number of duplicate keys is high. Additionally, other sorts such as insertion sort, shell sort, or merge sort could also be utilized.

**Credits:**

* <http://en.wikipedia.org/wiki/Quicksort#Variants>
* Algorithms, 4th Edition, by Sedgewick and Wayne, page 298

**Two-Way and Three-Way Partitioned Quicksort Compared**

One major difference between these two quicksorts is the number of swaps performed by each. The three-way partitioned quicksort performs many more swaps than the two-way partitioned quicksort in cases where the number of duplicate keys is low. This is because the three-way partitioned quicksort is designed for use with arrays containing a high number of duplicate keys and is not meant to recursively sort as many sub-lists as the two-way partitioned quicksort. The two-way partitioned quicksort is meant to put one value in the center and then recursively sort everything on both sides of the center as sub-lists, but the three-way partitioned quicksort relies on creating a fairly sizable center partition of duplicate keys and then doing a lesser amount of recursive sorting on values to the left and right of the center partition. So, when the number of duplicate keys is low, the three-way partitioned quicksort ends up doing many more swaps because of the increased number of sub-lists that are being recursively sorted.

The two-way partitioned quicksort can be used in all the same applications as the three-way partitioned quicksort, but will not perform as well in terms of time or number of swaps when the number of duplicate keys is very high. In the tests I conducted, which used arrays of random integers, there was no way of knowing ahead of time whether the number of duplicate keys would be high or low. However, the performance results shown by the tests indicate that the number of duplicate keys must have been low in almost every case because the two-way partitioned quicksort outperformed the three-way partitioned quicksort almost every time. If the program is run many consecutive times for the same number of values, one can eventually see the three-way partitioned quicksort perform better (when the random input can be assumed to have contained a higher number of duplicates), but these instances are rare. For that reason, I would recommend using the three-way partitioned quicksort only in cases where the number of duplicate keys is known to be very high ahead of time. When random values are being used, or when the input values are otherwise unknown prior to sorting, the two-way partitioned quicksort typically performs better.