**//steven guo**

**//quick sorting program**

**//10/20/19**

**import** java.util.Random;

//quicksort class

**public** **class** quickSort {

/\* This function takes last element as pivot,

places the pivot element at its correct

position in sorted array, and places all

smaller (smaller than pivot) to left of

pivot and all greater elements to right

of pivot \*/

**int** partition(**int** arr[], **int** low, **int** high)

{

**int** pivot = arr[high];

**int** i = (low-1); // index of smaller element

**for** (**int** j=low; j<high; j++)

{

// If current element is smaller than or

// equal to pivot

**if** (arr[j] <= pivot)

{

i++;

// swap arr[i] and arr[j]

**int** temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// swap arr[i+1] and arr[high] (or pivot)

**int** temp = arr[i+1];

arr[i+1] = arr[high];

arr[high] = temp;

**return** i+1;

}

/\* The main function that implements QuickSort()

arr[] --> Array to be sorted,

low --> Starting index,

high --> Ending index \*/

**void** sort(**int** arr[], **int** low, **int** high)

{

**if** (low < high)

{

/\* pi is partitioning index, arr[pi] is

now at right place \*/

**int** pi = partition(arr, low, high);

// Recursively sort elements before

// partition and after partition

sort(arr, low, pi-1);

sort(arr, pi+1, high);

}

}

//function that fills array with random number from 1-10000

**public** **static** **int**[] randArray(**int** size, **int**[] arr)

{

Random rand = **new** Random();

**int** number;

**for**(**int** i = 0; i < size-1; i++)

{

number = rand.nextInt(10000)+1;

arr[i] = number;

}

**return** arr;

}

//function that fills array with sorted numbers

**public** **static** **int**[] sortedArray(**int** size, **int**[] arr)

{

**for**(**int** i = 0; i < size-1; i++)

{

arr[i] = i;

}

**return** arr;

}

//function that fills array with sorted numbers except every 10th element is random

**public** **static** **int**[] almostSortedArray(**int** size, **int**[] arr)

{

Random rand = **new** Random();

**for**(**int** i = 0; i < size-1; i++)

{

**if**(i%10==0)

{

arr[i] = rand.nextInt(10000)+1;

}

**else**

{

arr[i] = i;

}

}

**return** arr;

}

**public** **static** **void** main(String arr[])

{

**final** **int** SIZE = 1000;

**final** **int** SIZE1 = 10000;

**final** **int** SIZE2 = 100000;

**long** time = System.*nanoTime*();

**int**[] arr1 = **new** **int**[SIZE];

**int**[] arr2 = **new** **int**[SIZE1];

**int**[] arr3 = **new** **int**[SIZE2];

quickSort ob = **new** quickSort();

ob.sort(*randArray*(SIZE, arr1),0,arr1.length-1);

System.***out***.println("It took " + ((System.*nanoTime*() - time)/1e-9) + " seconds for random array with 1000 elements");

time = System.*nanoTime*();

ob.sort(*randArray*(SIZE1, arr2),0,arr2.length-1);

System.***out***.println("It took " + ((System.*nanoTime*() - time)/1e-9) + " seconds for random array with 10000 elements");

time = System.*nanoTime*();

ob.sort(*randArray*(SIZE2, arr3),0,arr3.length-1);

System.***out***.println("It took " + ((System.*nanoTime*() - time)/1e-9) + " seconds for random array with 100000 elements");

time = System.*nanoTime*();

ob.sort(*sortedArray*(SIZE, arr1),0,arr1.length-1);

System.***out***.println("It took " + ((System.*nanoTime*() - time)/1e-9) + " seconds for sorted array with 1000 elements");

time = System.*nanoTime*();

ob.sort(*sortedArray*(SIZE1, arr2),0,arr2.length-1);

System.***out***.println("It took " + ((System.*nanoTime*() - time)/1e-9) + " seconds for sorted array with 10000 elements");

time = System.*nanoTime*();

ob.sort(*almostSortedArray*(SIZE, arr1),0,arr1.length-1);

System.***out***.println("It took " + ((System.*nanoTime*() - time)/1e-9) + " seconds for almost sorted array with 1000 elements");

time = System.*nanoTime*();

ob.sort(*almostSortedArray*(SIZE1, arr2),0,arr2.length-1);

System.***out***.println("It took " + ((System.*nanoTime*() - time)/1e-9) + " seconds for almost sorted array with 10000 elements");

time = System.*nanoTime*();

ob.sort(*almostSortedArray*(SIZE2, arr3),0,arr3.length-1);

System.***out***.println("It took " + ((System.*nanoTime*() - time)/1e-9) + " seconds for almost sorted array with 100000 elements");

time = System.*nanoTime*();

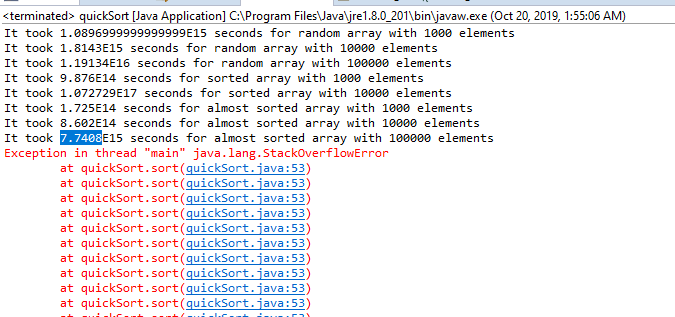
ob.sort(*sortedArray*(SIZE2, arr3),0,arr3.length-1);

System.***out***.println("It took " + ((System.*nanoTime*() - time)/1e-9) + " seconds for sorted array with 100000 elements");

time = System.*nanoTime*();

}

}



The average case of a quick sort is O(n log n). The worst case is O(n2). The best case and average case is O(n log n). Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays. First it picks an element called a pivot from the array, then it reorders the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition* operation. Then it recursively applies the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values. My theory on quick sort is that the sorted array should be the slowest. The most unbalanced partition occurs when one of the sublists returned by the partitioning routine is of size *n* – 1. This may occur if the pivot happens to be the smallest or largest element in the list, or in some implementations when all the elements are equal. If this happens repeatedly in every partition, then each recursive call processes a list of size one less than the previous list. Consequently, we can make *n* − 1 nested calls before we reach a list of size 1. This means that the call tree is a linear chain of *n* − 1 nested calls. The *i*th call does *O*(*n* − *i*) work to do the partition, and ∑ i = 0 n ( n − i ) = O ( n 2 ) {\displaystyle \textstyle \sum \_{i=0}^{n}(n-i)=O(n^{2})} , so in that case Quicksort takes *O*(*n*²) time. To sort an array of n distinct elements, quicksort takes O(n log n) time in expectation, averaged over all n! permutations of n elements with equal probability. My theory on quick sort is that when the array is sorted, it should have the fastest time. Based on the results, the random array had the best results. I got an overflow error when I ran the sorting algorithm with a random array with 100000 elements. I think be because of the large values of the elements.