Public class sorting algorithms engineering essay

Engineering

package assignment1. sorting;* @author Joris Schelfaut* @note Some of the code in this document was taken from the textbook* " Algorithms Fourth Edition" by Robert Sedgewick and Kevin Wayne.* These code fragments were then adapted to fit the assignment.

*/

public class SortingAlgorithms {private > boolean less(Comparable v, Comparable w) {return ((T) v). compareTo(((T) w)) < 0;

}

private > void exch(Comparable [] a, int i, int j) {Comparable t = a[i]; a[i] = a[j]; a[j] = t;

}

private > void insert(T[] a, int i, int j) {assert i > j; T temp = a[i]; for (int k = i; k > j; k--) {a[k] = a[k - 1];

}

a[j] = temp;

}

/**

* Sorts the given array using selection sort.

*

* @return The number of comparisons (i. e. calls to compareTo) performed by the algorithm.

```
*/
public > int selectionSort(T[] array) {int c = 0; int n = array. length; for (int i = 0; i < n; i++) {int min = i; for (int j = i + 1; j < n; j++) {if (less(array[j], array[min])) {min = j;

}
c++; // count the number of times " less" is performed (regardless of the result of less)

}
exch(array, i, min);

}
return c;

}
/**
* Sorts the given array using insertion sort.

*
* @return The number of comparisons (i. e. calls to compareTo) performed by the algorithm.* @note It took some time before we (Prince and I) managed to rewrite* Sledgewick method so it would count the number of* comparisons.
```

*/

public > int insertionSort(T[] array) {int c = 0; for (int i = 0; i < array. length; i++) {int j = i; while (j > 0) {c++; if (array[j - 1]. compareTo(array[i]) <= 0) {break;

}

--j;

}

insert(array, i, j);

}

return c;

}

/**

* Sorts the given array using (2-way) merge sort.

*

* HINT: Java does not supporting creating generic arrays (because the* compiler uses type erasure for generic types). For example, the statement* " T[] aux = new T[100];" is rejected by the compiler. Use the statement* " T[] aux = (T[]) new Comparable[100];" instead. Add an* "@SuppressWarnings(" unchecked")" annotation to prevent the compiler from* reporting a warning. Consult the following url for more information on* generics in Java:* http://download. oracle. com/javase/tutorial/java/generics/index. html

*

* @return The number of comparisons (i. e. calls to compareTo) performed by* the algorithm.

*/

public > int mergeSort(T[] array) {int c = 0; c = mergeSortRecursiveStep(array, 0, array. length - 1); return c;

}

private > int mergeSortRecursiveStep(Comparable [] array, int low, int high) {int c = 0; if (high <= low) {return 0;

}

int mid = low + (high - low) / 2; c += mergeSortRecursiveStep(array, low, mid); c += mergeSortRecursiveStep(array, mid + 1, high); c += merge(array, low, mid, high); return c;

}

private > int merge(Comparable [] array, int low, int mid, int high) {T[] aux = (T[]) new Comparable[array. length]; int counter = 0; int i = low, j = mid + 1;// AUX vullen. for (int k = low; k <= high; k++) {aux[k] = (T) array[k];

}

for (int k = low; k <= high; k++) {if (i > mid) {array[k] = aux[j++];} else if (j > high) {array[k] = aux[i++];} else {if (less(aux[j], aux[i])) {array[k] = aux[j++];} else {array[k] = aux[i++];

}

counter++;

}

}

return counter;

}

/**

* Sorts the given array using quick sort. Do NOT perform a random shuffle.

*

* @return The number of comparisons (i. e. calls to compareTo) performed by* the algorithm.

*/

public > int quickSort(T[] array) {int c = 0;// StdRandom. shuffle(array); c = quickSortRecursiveStep(array, 0, array. length - 1); return c;

}

private > int quickSortRecursiveStep(T[] array, int low, int high) {int c = 0; int[] temp = new int[2]; if (high <= low) {return 0;

}

temp = partition(array, low, high); int j = temp[1]; c += temp[0]; c += quickSortRecursiveStep(array, low, j - 1); c += quickSortRecursiveStep(array, j + 1, high); return c;

}

private > int[] partition(T[] array, int low, int high) {int c[] = new int[2]; int i = low, j = high + 1; Comparable v = array[low]; while (true) {while (less(array[++i], v)) {c[0]++; if (i == high) {break;

}

}

c[0]++; while (less(v, array[--j])) {c[0]++; if (j == low) {break;

}

}

c[0]++; if (i >= j) {break;

}

exch(array, i, j);

}

exch(array, low, j); c[1] = j; return c;

}

/**

* Sorts the given array using k-way merge sort. The implementation can* assume that k is at least 2. k is the number of the number of subarrays* (at each level) that must be separately sorted via a recursive call and* merged via a k-way merge. For example, if k equals 3, then the array must* be subdivided into three subarrays that are each sorted by 3-way merge* sort. After the 3 sub- arrays, these sub-arrays are combined via a 3-way* merge.

*

* Note that if k is larger than the length of the array (or larger than the* length of a sub-array in a recursive call), then the implementation is* allowed sort that sub-array using quick sort.

*

* @return An non-null array of length 2. The first element of this array is* the number of comparisons (i. e. calls to compareTo) performed by* the algorithm, while the second element is the number of data* moves.* @note After solving a lot of problems with indexes I finally managed to* get it right! Hurray!

*/

public > int[] kWayMergeSort(T[] array, int K) {assert K > 1; int[] count = new int[2]; T[] aux = (T[]) new Comparable[array. length];// If K > array. length, it would do exactly the same as when// array. length == K, but that isn't what was asked, I guess.// Note that the return value isn't exactly what it says, though. if(K <= array. length) {count = kWayMergeSortRecursiveStep(array, 0, array. length, aux, K);} else {count[0] = quickSort(array); count[1] = 0; // our implementation of quick sort does not consider data moves

}

// return the result. return count;

}

/**

* This method divides an array into K compartiments (subarrays) and sorts* these seperately.* In the next step these compartiments are " merged" by sorting them within the supercompartiment.* At the end of the recursion, the array from index " low" to index high should be sorted.

*

* @return An non-null array of length 2. The first element of this array is* the number of comparisons (i. e. calls to compareTo) performed by* the algorithm, while the second element is the number of data* moves.

*/

private > int[] kWayMergeSortRecursiveStep(Comparable [] array, int low, int high, Comparable [] aux, int K) {// Local variablesint[] count = new int[2]; count[0] = 0; count[1] = 0;// Determine how many subarrays we can make at this levelint difference = high - low; int interval = difference / K; int[] indices;// Decide when to returnif (high - 1 <= low) {return count;

}

// Determine the boundaries of the subarrays. if(difference >= K) {// We get nice equal subarrays.// determine " K + 1" indices to create K subarrays (includes lower and upper boundaries)indices = new int[K + 1];// boundaries : indices[0] = low; indices[K] = high;// initialize the rest of the array of indices.// The last subarray will have a different number of elements.// if difference % K != 0for (int k = 1; k < K; k++) {indices[k] = indices[k - 1] + interval;

}

} else {// if (difference % K > 0)

// ----------

// we can't further divide into K even subarrays.// we divide into K - 1 equal parts and a last part.// There will be less than K subarraysindices = new int[difference + 1]; indices[0] = low; indices[difference] = high;// All arrays of size 1. for (int k = 1; k < indices. length - 1; k++) {indices[k] = indices[k - 1] + 1;

}

}

// call the recursive methods K - 1 times. for (int k = 0; k < indices. length - 1; k++) {// sort from array[indices[k]] to array[indices[k + 1] - 1]int[] temp = kWayMergeSortRecursiveStep(array, indices[k], indices[k + 1], aux, K); count[0] += temp[0]; count[1] += temp[1];

}

// Merge all subarrays of the array. int temp[] = kWayMerge(array, indices, aux); count[0] += temp[0]; count[1] += temp[1];// Return the result. return count;

}

/**

* Merge subarrays with indices for array_1 " indices[0] to indices[1] - 1" and* array_2 " indices[1] to indices[2] - 1", giving array_1_2.* Then merge

array_1_2 with indices " indices[0] to indices[2] - 1" with* array_3 " indices[2] to indices[3] - 1".* Do this for all K subarrays.

*

* @return An non-null array of length 2. The first element of this array is* the number of comparisons (i. e. calls to compareTo) performed by* the algorithm, while the second element is the number of data* moves.

*/

private > int[] kWayMerge(Comparable [] array, int[] indices, Comparable [] aux) {int[] count = new int[2];// to merge K subarrays, we need K - 1 merges.// or : the number of indices - the index for the high and low indexfor (int k = 0; k < indices. length - 2; k++) {int temp[] = iterativeMergeStep(array, indices[0], indices[k + 1] - 1, indices[k + 2] - 1, aux); count[0] += temp[0]; count[1] += temp[1];

}

return count;

}

/**

* @return An non-null array of length 2. The first element of this array is* the number of comparisons (i. e. calls to compareTo) performed by* the algorithm, while the second element is the number of data* moves.

*/

private > int[] iterativeMergeStep(Comparable [] array, int low, int mid, int high, Comparable [] aux)

{

int[] count = new int[2]; int i = low, j = mid + 1; for (int k = low; k <= high; k++) {aux[k] = array[k];

}

for (int k = low; k <= high; k++) {if (i > mid) {// everything is sorted from " low" to " mid"// add everything after mid. array[k] = aux[j++];// Moving data from subarray (auxilary array)// to the resulting (sorted) arraycount[1] ++;} else if (j > high) {// everything is sorted from " mid" to " high"// add all elements before mid. array[k] = aux[i++];// Moving data from subarray (auxilary array)// to the resulting (sorted) arraycount[1] ++;} else {if (less(aux[j], aux[i])) {// " aux[j]" is less than " aux[i]", so position// " aux[j]" in the array. array[k] = aux[j++];} else {array[k] = aux[i++];

}

// Comparing data. count[0] ++;// Moving data from subarray (auxilary array)// to the resulting (sorted) arraycount[1] ++;

}

}

return count;

}

/**

* Sorts the given array of strings using LSD sort. Each string in the input* array has length W.

*

* @return the number of arrays accesses performed by the algorithm

*/

public int lsdSort(String[] array, int W) {int arrayAccesses = 0; int N = array. length; int R = 256; String[] aux = new String[N]; for (int d = W - 1; d >= 0; d--) {int[] count = new int[R + 1]; for (int i = 0; i < N; i++) {count[array[i]. charAt(d) + 1]++;// we access " array" on position " i" arrayAccesses++;// we access " count" on position " array[i]. charAt(d) + 1" arrayAccesses++;

}

for (int r = 0; r < R; r++) {count[r + 1] += count[r];// we access " count" two timesarrayAccesses++; arrayAccesses++;

}

for (int i = 0; i < N; i++) {aux[count[array[i]. charAt(d)]++] = array[i];// we access " array" on position " i" arrayAccesses++; arrayAccesses++;// we access " count" arrayAccesses++;// we access " aux" arrayAccesses++;

}

for (int i = 0; i < N; i++) {array[i] = aux[i];// we access " array" arrayAccesses++;// we access " aux" arrayAccesses++;

}

}

return arrayAccesses;

}

/**

* Sorts the given array of strings using MSD sort. Do NOT use a cutoff for* small subarrays.

*

* @return the number of characters examined by the algorithm

*/

public int msdSort(String[] array) {int stringsExamined = 0; int R = 256; int N = array. length; int M = 0; // NO CUTOFF ! : OString[] aux = new String[N]; stringsExamined += msdSortRecursiveStep(array, 0, N - 1, 0, aux, R, N, M); return stringsExamined;

}

private int msdSortRecursiveStep(String[] array, int low, int high, int d, String[] aux, int R, int N, int M)

{

int stringsExamined = 0; if (high <= low + M) {// While sorting with selection sort, we also compare strings...// stringsExamined += stringInsertionSort(array, low, high, d);//return stringsExamined; return 0;

}

int[] count = new int[R + 2]; for (int i = low; i <= high; i++) {count[charAt(array[i], d) + 2]++;// the string on position i in the " array"// is examined with charAt()stringsExamined++;

}

for (int r = 0; r < R + 1; r++) {count[r + 1] += count[r];

}

for (int i = low; i <= high; i++) {aux[count[charAt(array[i], d) + 1]++] =

array[i];// the string on position i in the " array"// is examined with

charAt()stringsExamined++;

}

for (int i = low; i <= high; i++) {array[i] = aux[i - low];

}

//

for (int r = 0; r < R; r++) {stringsExamined += msdSortRecursiveStep(array,

low + count[r], low + count[r + 1] - 1, d + 1, aux, R, N, M);

}

return stringsExamined;

}

private int charAt(String s, int d) {if (d < s. length()) {return s. charAt(d);}

else {return -1;

}

}

public int stringInsertionSort(String[] array, int low, int high, int d) {int

stringsExamined = 0;// Sortingfor (int i = low; i <= high; i++) {int j = i; while

```
(j > 0) {stringsExamined++; if (array[j - 1]. substring(d). compareTo(array[i].
substring(d)) <= 0) {break;

}
j--;

}
insert(array, i, j);

}
return stringsExamined;

}
}
```