Sorting code:

Used Interface

========================================================================

**public** **interface** ISortingAlgo {

/\*\* 0. GuruJi Sort O(n) \*/

**public** **void** guruJiSort(**int** a[]);

/\*\* 1. Selection Sort \*\*/

**public** **void** selectionSort(**int** a[]);

/\*\* 2. Bubble Sort \*\*/

**public** **void** bubbleSort(**int** a[]);

/\*\* 3. Recursive Bubble Sort \*\*/

**public** **void** bubbleSortRecursive(**int** arr[], **int** n);

/\*\* 4. Insertion Sort \*\*/

**public** **void** insertionSort(**int**[] a);

/\*\* 5. Recursive Insertion Sort \*\*/

**public** **void** insertionSortRecursive(**int** arr[], **int** n);

/\*\* 6. Merge Sort \*\*/

**public** **void** mergeTwoPartSorted(**int** a[], **int** l, **int** m, **int** r);

**public** **void** mergeSort(**int** arr[], **int** l, **int** r);

/\*\* 7. Iterative Merge Sort \*\*/

**public** **void** mergeSortIterative(**int** a[]);

/\*\* 8. Quick Sort \*\*/

**public** **void** quickSort(**int**[] a, **int** l, **int** h);

/\*\* 9. Iterative Quick Sort \*\*/

/\*\* 10. Heap Sort \*\*/

/\*\*

\* arrange i index element greater than 2\*i+1 index and 2\*i+2 index element

\*\*/

**public** **void** maxHeapify(**int** a[], **int** n, **int** i);

/\*\*

\* arrange i index element less than 2\*i+1 index and 2\*i+2 index element

\*\*/

**public** **void** minHeapify(**int** a[], **int** n, **int** i);

**public** **void** maxHeapSort(**int** a[]);

**public** **void** minHeapSort(**int** a[]);

/\*\* 11. Counting Sort \*\*/

**public** **void** countingSort(**int**[] a);

/\*\* 12. Radix Sort \*\*/

/\*\* 11. Counting Sort for Radix sort \*\*/

**public** **void** countingSortRadix(**int**[] a, **int** exp);

**public** **void** radixSort(**int** a[]);

/\*\* 13. Bucket Sort \*\*/

**public** **void** bucketSort(**double**[] a);

/\*\* 14. ShellSort \*\*/

**public** **void** shellSort(**int** arr[]);

/\*\* 15. TimSort \*\*/

**public** **void** timeSort(**int** a[]);

/\*\* 16. Comb Sort \*\*/

**public** **void** combSort(**int** a[]);

/\*\* 17. Pigeonhole Sort \*\*/

**public** **void** pigeonholeSort(**int** a[]);

/\*\* 18. Cycle Sort \*\*/

**public** **void** cycleSort(**int** a[]);

/\*\* 19. Cocktail Sort \*\*/

**public** **void** cocktailSort(**int** a[]);

/\*\* 20. Strand Sort \*\*/

**public** **void** strandSort(**int** a[]);

/\*\* 21. Bitonic Sort \*\*/ // **TODO**

**public** **void** compAndSwap(**int** a[], **int** i, **int** j, **int** dir);

**public** **void** bitonicMerge(**int** a[], **int** low, **int** cnt, **int** dir);

**public** **void** bitonicSort(**int** a[], **int** low, **int** cnt, **int** dir);

/\*\* 22. Pancake sorting \*\*/

**public** **void** pankakeSorte(**int** a[]);

/\*\* 23. Binary Insertion Sort \*\*/

**public** **void** binaryInsertionSort(**int** a[]);

/\*\* 24. BogoSort or Permutation Sort \*\*/

**public** **boolean** isSorted(**int** a[], **int** n);

// To generate permuatation of the array

**public** **void** shuffle(**int** a[], **int** n);

// Sorts array a[0..n-1] using Bogo sort

**public** **void** bogosort(**int** a[]);

/\*\* 25. Gnome Sort \*\*/

**public** **void** gnomeSort(**int** a[]);

/\*\* 26. Sleep Sort – The King of Laziness / Sorting while Sleeping \*\*/

**public** **void** sleepSort(**int** a[]);

/\*\* 27. Structure Sorting (By Multiple Rules) in C++ \*\*/

**public** **void** structureSort(Student s[]);

/\*\* 28. Stooge Sort \*\*/

**public** **void** stoogeSort(**int** a[], **int** l, **int** h);

/\*\* 29. Tag Sort (To get both sorted and original) \*\*/

**public** **void** tagSort(Person persons[], **int** tag[]);

/\*\* 30. Tree Sort \*\*/

/\*\* create tree and print data in inorder traverse \*/

/\*\* 31. Cartesian Tree Sorting \*\*/

/\*\* 32. Odd-Even Sort / Brick Sort \*\*/

**public** **void** oddEvenSort(**int** arr[]);

/\* 33. QuickSort on Singly Linked List \*/

/\* 34. QuickSort on Doubly Linked List \*/

/\* 35. 3-Way QuickSort (Dutch National Flag) \*/

/\* 36. Merge Sort for Linked Lists \*/

/\* 37. Merge Sort for Doubly Linked List \*/

/\* 38. 3-way Merge Sort \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\* swap two element of array \*\*/

**public** **void** swap(**int** a[], **int** i, **int** j);

/\*\* min element in a array \*\*/

**public** **int** min(**int** a[]);

/\*\* max element in a array \*\*/

**public** **int** max(**int** a[]);

===============================================================================

Interface Implementation

=============================================================================

**public** **class** SortingAlgoImpl **implements** ISortingAlgo {

/\*\* 0. GuruJi Sort O(n) \*/

/\* this is useful when max is not much more bigger than no of element \*/

/\*

\* it is useful for not only uniue but also for duplicate but not for -ve

\* element

\*/

**public** **void** guruJiSort(**int** a[]) {

**int** n = a.length;

**int** i;

**int** max = 0;

// O(n)

**for** (i = 0; i < n; i++) {

**if** (max < a[i])

max = a[i];

}

**int** b[][] = **new** **int**[max + 1][2];

// O(n)

**for** (i = 0; i < n; i++) {

b[a[i]][0] = a[i];

b[a[i]][1] += 1;

}

**int** k = 0;

// O(max)

**for** (i = 0; i < max + 1; i++) {

**for** (**int** j = 0; j < b[i][1]; j++)

a[k++] = b[i][0];

}

}

/\*\* 1. Selection Sort \*\*/

@Override

**public** **void** selectionSort(**int**[] a) {

**if** (a == **null** || a.length == 0 || a.length == 1)

**return**;

**else** {

**for** (**int** i = 0; i < a.length; i++) {

**for** (**int** j = i + 1; j < a.length; j++) {

**if** (a[i] > a[j]) {

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

a[i] = a[j];

a[j] = temp;

}

}

}

}

}

/\*\* 2. Bubble Sort \*\*/

@Override

**public** **void** bubbleSort(**int**[] a) {

**if** (a == **null** || a.length == 0 || a.length == 1)

**return**;

**else** {

**for** (**int** i = 0; i < a.length; i++) {

**for** (**int** j = 0; j < a.length - i - 1; j++) {

**if** (a[j] > a[j + 1]) {

**int** temp = a[j];

a[j] = a[j + 1];

a[j + 1] = temp;

}

}

}

}

}

/\*\* 3. Recursive Bubble Sort \*\*/

@Override

**public** **void** bubbleSortRecursive(**int** arr[], **int** n) {

// Base case no need to sort

**if** (n == 1)

**return**;

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

**if** (arr[i] > arr[i + 1]) {

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

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

arr[i + 1] = temp;

}

/\* Largest element is fixed,recur for remaining array \*/

bubbleSortRecursive(arr, n - 1);

}

/\*\* 4. Insertion Sort \*\*/

@Override

**public** **void** insertionSort(**int**[] a) {

**if** (a.length == 0 || a.length == 1)

**return**;

**else** {

**int** n = a.length;

**for** (**int** i = 1; i < n; ++i) {

**int** key = a[i];

**int** j = i - 1;

/\*

\* Move elements of arr[0..i-1], that are greater than key, to one position

\* ahead of their current position

\*/

**while** (j >= 0 && a[j] > key) {

a[j + 1] = a[j];

j = j - 1;

}

a[j + 1] = key;

}

}

}

/\*\* 5. Recursive Insertion Sort \*\*/

**public** **void** insertionSortRecursive(**int** arr[], **int** n) {

// Base case

**if** (n <= 1)

**return**;

// Sort first n-1 elements

insertionSortRecursive(arr, n - 1);

// Insert last element at its correct position

// in sorted array.

**int** last = arr[n - 1];

**int** j = n - 2;

/\*

\* Move elements of arr[0..i-1], that are greater than key, to one position

\* ahead of their current position

\*/

**while** (j >= 0 && arr[j] > last) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = last;

}

/\*\* 6. Merge Sort \*\*/

**public** **void** mergeTwoPartSorted(**int** a[], **int** l, **int** m, **int** r) {

**int** n1 = m - l + 1;

**int** n2 = r - m;

**int** x[] = **new** **int**[n1];

**int** y[] = **new** **int**[n2];

**for** (**int** i = 0; i < n1; i++)

x[i] = a[l + i];

**for** (**int** j = 0; j < n2; j++)

y[j] = a[m + 1 + j];

/\* Merge the temp arrays \*/

// Initial indexes of first and second subarrays

**int** i = 0, j = 0;

**int** k = l;

**while** (i < n1 && j < n2) {

**if** (x[i] <= y[j]) {

a[k] = x[i];

i++;

} **else** {

a[k] = y[j];

j++;

}

k++;

}

/\* copy remaining of x[] \*/

**while** (i < n1) {

a[k] = x[i];

k++;

i++;

}

/\* copy remaining of y[] \*/

**while** (j < n2) {

a[k] = y[j];

k++;

j++;

}

}

@Override

**public** **void** mergeSort(**int** arr[], **int** l, **int** r) {

**if** (l < r) {

// Find the middle point

**int** m = (l + r) / 2;

// Sort first and second halves

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

// Merge the sorted halves

mergeTwoPartSorted(arr, l, m, r);

}

}

/\*\* 7. Iterative Merge Sort \*\*/

@Override

**public** **void** mergeSortIterative(**int** a[]) {

**if** (a == **null** || a.length == 0 || a.length == 1)

**return**;

**if** (a.length > 1) {

**int** n = a.length;

**int** m = n / 2;

/\* split left \*/

**int** x[] = **new** **int**[m];

**for** (**int** i = 0; i < m; i++)

x[i] = a[i];

/\* split right \*/

**int** y[] = **new** **int**[n - m];

**for** (**int** j = 0; j < n - m; j++)

y[j] = a[m + j];

mergeSortIterative(x);

mergeSortIterative(y);

**int** i = 0, j = 0;

**int** k = 0;

**int** n1 = x.length;

**int** n2 = y.length;

**while** (i < n1 && j < n2) {

**if** (x[i] <= y[j]) {

a[k] = x[i];

i++;

} **else** {

a[k] = y[j];

j++;

}

k++;

}

/\* copy remaining of x[] \*/

**while** (i < n1) {

a[k] = x[i];

k++;

i++;

}

/\* copy remaining of y[] \*/

**while** (j < n2) {

a[k] = y[j];

k++;

j++;

}

}

}

/\*\* 8. Quick Sort \*\*/

@Override

**public** **void** quickSort(**int**[] a, **int** l, **int** h) {

**int** pvt = l + (h - l) / 2;

**int** i = l;

**int** j = h;

**while** (i <= j) {

**while** (a[i] < a[pvt])

i++;

**while** (a[j] > a[pvt])

j--;

**if** (i <= j) {

swap(a, i, j);

i++;

j--;

}

}

**if** (l < j)

quickSort(a, l, j);

**if** (i < h)

quickSort(a, i, h);

}

/\*\* 9. Iterative Quick Sort \*\*/

/\*\* 10. Heap Sort \*\*/

/\*\*

\* arrange i index element greater than 2\*i+1 index and 2\*i+2 index element

\*\*/

@Override

**public** **void** maxHeapify(**int** a[], **int** n, **int** i) {

**int** lar = i;

**int** l = 2 \* i + 1;

**int** r = 2 \* i + 2;

**if** (l < n && a[l] > a[lar])

lar = l;

**if** (r < n && a[r] > a[lar])

lar = r;

**if** (lar != i) {

**int** swap = a[i];

a[i] = a[lar];

a[lar] = swap;

maxHeapify(a, n, lar);

}

}

/\*\*

\* arrange i index element less than 2\*i+1 index and 2\*i+2 index element

\*\*/

@Override

**public** **void** minHeapify(**int** a[], **int** n, **int** i) {

**int** min = i;

**int** l = 2 \* i + 1;

**int** r = 2 \* i + 2;

**if** (l < n && a[l] < a[min])

min = l;

**if** (r < n && a[r] < a[min])

min = r;

**if** (min != i) {

**int** swap = a[i];

a[i] = a[min];

a[min] = swap;

minHeapify(a, n, min);

}

}

@Override

**public** **void** maxHeapSort(**int** a[]) {

**int** n = a.length;

**for** (**int** i = n / 2 - 1; i >= 0; i--) // build head so max is at 0

maxHeapify(a, n, i);

// swap the 0 to end and hipify the rest

**for** (**int** i = n - 1; i >= 0; i--) {

**int** temp = a[0];

a[0] = a[i];

a[i] = temp;

maxHeapify(a, i, 0);

}

}

@Override

**public** **void** minHeapSort(**int** a[]) {

**int** n = a.length;

**for** (**int** i = n / 2 - 1; i >= 0; i--) // build head so max is at 0

minHeapify(a, n, i);

// swap the 0 to end and hipify the rest

**for** (**int** i = n - 1; i >= 0; i--) {

**int** temp = a[0];

a[0] = a[i];

a[i] = temp;

minHeapify(a, i, 0);

}

}

/\*\* 11. Counting Sort \*\*/

@Override

**public** **void** countingSort(**int**[] a) {

**int** max = max(a);

**int** output[] = **new** **int**[a.length];

**int** count[] = **new** **int**[max + 1];

**for** (**int** i = 0; i < a.length; i++) {

++count[a[i]];

}

// add the previous to current

**for** (**int** i = 1; i < max + 1; i++)

count[i] += count[i - 1];

// build output array

**for** (**int** i = 0; i < a.length; i++) {

output[count[a[i]] - 1] = a[i];

--count[a[i]];

}

**for** (**int** i = 0; i < a.length; i++)

a[i] = output[i];

}

/\*\* 12. Radix Sort \*\*/

/\*\* 11. Counting Sort for Radix sort \*\*/

**public** **void** countingSortRadix(**int**[] a, **int** exp) {

**int** n = a.length;

**int** i;

**int** output[] = **new** **int**[n];

**int** count[] = **new** **int**[10];

/\* add the count occurrences in count[] \*/

**for** (i = 0; i < n; i++)

count[(a[i] / exp) % 10]++;

/\* add the previous to current \*/

**for** (i = 1; i < 10; i++)

count[i] += count[i - 1];

/\* Build the output array \*/

**for** (i = n - 1; i >= 0; i--) {

output[count[(a[i] / exp) % 10] - 1] = a[i];

count[(a[i] / exp) % 10]--;

}

/\* copy the out array to actual array \*/

**for** (i = 0; i < n; i++)

a[i] = output[i];

}

@Override

**public** **void** radixSort(**int** a[]) {

**int** m = max(a);

/\* Do counting sort for every digit of numbers \*/

**for** (**int** exp = 1; m / exp > 0; exp \*= 10)

countingSortRadix(a, exp);

}

/\*\* 13. Bucket Sort \*\*/

@Override

**public** **void** bucketSort(**double**[] a) {

@SuppressWarnings("unchecked")

List<Double>[] arList = **new** List[10];

**for** (**int** i = 0; i < arList.length; i++)

arList[i] = **new** ArrayList<Double>();

// O(n)

**int** n = a.length;

**for** (**int** i = 0; i < n; i++) {

**int** bucket = (**int**) ((**int**) n \* a[i]);

arList[bucket].add(a[i]);

}

**int** k = 0;

**for** (**int** j = 0; j < arList.length; j++) {

Collections.*sort*(arList[j]);

**for** (Double fl : arList[j])

a[k++] = fl;

}

}

/\*\* 14. ShellSort \*\*/

**public** **void** shellSort(**int** arr[]) {

**int** n = arr.length;

// Start with a big gap, then reduce the gap

**for** (**int** gap = n / 2; gap > 0; gap /= 2) {

**for** (**int** i = gap; i < n; i += 1) {

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

**int** j;

**for** (j = i; j >= gap && arr[j - gap] > temp; j -= gap)

arr[j] = arr[j - gap];

arr[j] = temp;

}

}

}

/\*\* 15. TimSort \*\*/

**public** **static** **int** *RUN* = 32;

**public** **void** timeSort(**int** a[]) {

**int** n = a.length;

**for** (**int** i = 0; i < n; i += *RUN*)

AlgoUtils.*insertionSort*(a, i, Math.*min*((i + 31), (n - 1)));

**for** (**int** size = *RUN*; size < n; size = 2 \* size) {

**for** (**int** left = 0; left < n; left += 2 \* size) {

**int** mid = left + size - 1;

**int** right = Math.*min*((left + 2 \* size - 1), (n - 1));

AlgoUtils.*mergeTwoPartSorted*(a, left, mid, right);

}

}

}

/\*\* 16. Comb Sort \*\*/

// To find gap between elements

**public** **int** getNextGap(**int** gap) {

// Shrink gap by Shrink factor

gap = (gap \* 10) / 13;

**if** (gap < 1)

**return** 1;

**return** gap;

}

**public** **void** combSort(**int** a[]) {

**int** n = a.length;

// Initialize gap

**int** gap = n;

// Initialize swapped as true to make sure that

// loop runs

**boolean** swapped = **true**;

// Keep running while gap is more than 1 and last

// iteration caused a swap

**while** (gap != 1 || swapped == **true**) {

// Find next gap

gap = getNextGap(gap);

// Initialize swapped as false so that we can

// check if swap happened or not

swapped = **false**;

// Compare all elements with current gap

**for** (**int** i = 0; i < n - gap; i++) {

**if** (a[i] > a[i + gap]) {

AlgoUtils.*swap*(a, i, i + gap);

swapped = **true**;

}

}

}

}

/\*\* 17. Pigeonhole Sort \*\*/

**public** **void** pigeonholeSort(**int** a[]) {

**int** max = max(a);

**int** min = min(a);

**int** i, j, index, range;

range = max - min + 1;

**int** count[] = **new** **int**[max + 1];

**for** (i = 0; i < a.length; i++) {

++count[a[i] - min];

}

/\*

\* // add the previous to current for (int i = 1; i < max + 1; i++) count[i] +=

\* count[i - 1];

\*/

index = 0;

**for** (j = 0; j < range; j++)

**while** (count[j]-- > 0)

a[index++] = j + min;

/\*

\* // build output array for (int i = 0; i < a.length; i++) { output[count[a[i]]

\* - 1] = a[i]; --count[a[i]]; } for (int i = 0; i < a.length; i++) a[i] =

\* output[i];

\*/

}

/\*\* 18. Cycle Sort \*\*/

**int** writes = 0;

@Override

**public** **void** cycleSort(**int** a[]) {

**int** n = a.length;

**for** (**int** cl = 0; cl <= n - 2; cl++) {

**int** item = a[cl];

**int** pos = cl;

**for** (**int** i = cl + 1; i < n; i++)

**if** (a[i] < item)

pos++;

// If item is already in correct position

**if** (pos == cl)

**continue**;

// ignore all duplicate elements

**while** (item == a[pos])

pos += 1;

// put the item to it's right position

**if** (pos != cl) {

**int** temp = item;

item = a[pos];

a[pos] = temp;

writes++;

}

// Rotate rest of the cycle

**while** (pos != cl) {

pos = cl;

// Find position where we put the element

**for** (**int** i = cl + 1; i < n; i++)

**if** (a[i] < item)

pos += 1;

// ignore all duplicate elements

**while** (item == a[pos])

pos += 1;

// put the item to it's right position

**if** (item != a[pos]) {

**int** temp = item;

item = a[pos];

a[pos] = temp;

writes++;

}

}

}

}

/\*\* 19. Cocktail Sort \*\*/

**public** **void** cocktailSort(**int** a[]) {

**int** l = 0;

**int** e = a.length;

**boolean** swaped = **true**;

**while** (swaped) {

swaped = **false**;

// iteration last will be biggest one

**for** (**int** i = l; i < e - 1; i++) {

**if** (a[i] > a[i + 1]) {

AlgoUtils.*swap*(a, i, i + 1);

swaped = **true**;

}

}

// arr is already sorted

**if** (!swaped)

**break**;

swaped = **false**;

e = e - 1;

**for** (**int** i = e - 1; i >= l; i--) {

**if** (a[i] > a[i + 1]) {

AlgoUtils.*swap*(a, i, i + 1);

swaped = **true**;

}

}

l = l + 1;

}

}

/\*\* 20. Strand Sort \*\*/

**public** **void** strandSort(**int** a[]) {

List<Integer> in = **new** ArrayList<>();

List<Integer> op = **new** ArrayList<>();

**for** (**int** i = 0; i < a.length; in.add(a[i]), i++)

;

op = strandSort(in, op);

**for** (**int** i = 0; i < a.length; a[i] = op.get(i), i++)

;

}

**public** List<Integer> strandSort(List<Integer> in, List<Integer> op) {

**if** (in.isEmpty())

**return** op;

List<Integer> subList = **new** ArrayList<>();

subList.add(in.get(0));

in.remove(0);

**for** (**int** i = 0; i < in.size(); i++) {

**if** (in.get(i) > subList.get(subList.size() - 1)) {

subList.add(in.get(i));

in.remove(i);

}

}

op = SortingUtils.*mergeSortedList*(op, subList);

**return** strandSort(in, op);

}

/\*\* 21. Bitonic Sort \*\*/

@Override

**public** **void** compAndSwap(**int** a[], **int** i, **int** j, **int** dir) {

**if** ((a[i] > a[j] && dir == 1) || (a[i] < a[j] && dir == 0)) {

AlgoUtils.*swap*(a, i, j);

}

}

@Override

**public** **void** bitonicMerge(**int** a[], **int** low, **int** cnt, **int** dir) {

**if** (cnt >= 1) {

**int** k = cnt / 2;

**for** (**int** i = low; i < low + k; i++)

compAndSwap(a, i, i + k, dir);

bitonicMerge(a, low, k, dir);

bitonicMerge(a, low + k, k, dir);

}

}

@Override

**public** **void** bitonicSort(**int** a[], **int** low, **int** cnt, **int** dir) {

**if** (cnt >= 1) {

**int** k = cnt / 2;

// sort in ascending order since dir here is 1

bitonicSort(a, low, k, 1);

// sort in descending order since dir here is 0

bitonicSort(a, low + k, k, 0);

// Will merge wole sequence in ascending order

// since dir=1.

bitonicMerge(a, low, cnt, dir);

}

}

/\*\* 22. Pancake sorting \*\*/

**public** **void** pankakeSorte(**int** a[]) {

**int** n = a.length - 1;

**for** (**int** i = n; i > 0; i--) {

**int** maxInd = AlgoUtils.*maxIndex*(a, i + 1);

AlgoUtils.*rvereseArray*(a, maxInd, i);

}

}

/\*\* 22. Binary Insertion Sort \*\*/

**public** **int** loc;

@Override

**public** **void** binaryInsertionSort(**int** a[]) {

**if** (a.length == 0 || a.length == 1)

**return**;

**else** {

**int** n = a.length;

**for** (**int** i = 1; i < n; ++i) {

**int** key = a[i];

**int** j = i - 1;

loc = AlgoUtils.*binarySearch*(a, 0, j, key);

**while** (j >= 0 && a[j] > key) {

a[j + 1] = a[j];

j = j - 1;

}

a[j + 1] = key;

}

}

}

/\*\* 24. BogoSort or Permutation Sort \*\*/

@Override

**public** **boolean** isSorted(**int** a[], **int** n) {

**while** (--n >= 1)

**if** (a[n] < a[n - 1])

**return** **false**;

**return** **true**;

}

// To generate permuatation of the array

@Override

**public** **void** shuffle(**int** a[], **int** n) {

**for** (**int** i = 0; i < n; i++)

AlgoUtils.*swap*(a, i, (**int**) (Math.*random*() \* i));

}

// Sorts array a[0..n-1] using Bogo sort

@Override

**public** **void** bogosort(**int** a[]) {

**int** n = a.length;

// if array is not sorted then shuffle

// the array again

**int** cnt = 0;

**while** (!isSorted(a, n)) {

shuffle(a, n);

cnt++;

}

System.***out***.println(cnt);

}

/\*\* 25. Gnome Sort \*\*/

**public** **void** gnomeSort(**int** a[]) {

**int** n = a.length;

**int** ind = 0;

**while** (ind < n) {

**if** (ind == 0)

ind++;

**if** (a[ind] >= a[ind - 1])

ind++;

**else** {

AlgoUtils.*swap*(a, ind, ind - 1);

ind--;

}

}

**return**;

}

/\*\* 26. Sleep Sort – The King of Laziness / Sorting while Sleeping \*\*/

@Override

**public** **void** sleepSort(**int** a[]) {

**for** (**int** i = 0; i < a.length; i++) {

Thread thread = **new** Thread(**new** SleepsortThread(a[i] \* 10));

thread.start();

}

}

/\*\* 27. Structure Sorting (By Multiple Rules) in C++ \*\*/

@Override

**public** **void** structureSort(Student a[]) {

**int** n = a.length;

// To calculate total marks for all Students

**for** (**int** i = 0; i < n; i++)

a[i].total = a[i].math + a[i].phy + a[i].che;

List<Student> list = **new** ArrayList<>();

**for** (**int** i = 0; i < n; list.add(a[i]), i++)

;

Collections.*sort*(list, **new** Comparator<Student>() {

@Override

**public** **int** compare(Student a, Student b) {

// returns true for higher total

**if** (a.total != b.total)

**return** a.total - b.total;

// If marks in Maths are not same then

// returns true for higher marks

**if** (a.math != b.math)

**return** a.math - b.math;

**return** (a.phy - b.phy);

}

});

// Assigning ranks after sorting

**for** (**int** i = 0; i < n; i++)

a[i].rank = i + 1;

}

/\*\* 28. Stooge Sort \*\*/

**public** **void** stoogeSort(**int** arr[], **int** l, **int** h) {

**if** (l >= h)

**return**;

// If first element is smaller than last,

// swap them

**if** (arr[l] > arr[h])

AlgoUtils.*swap*(arr, l, h);

// If there are more than 2 elements in

// the array

**if** (h - l + 1 > 2) {

**int** t = (h - l + 1) / 3;

// Recursively sort first 2/3 elements

stoogeSort(arr, l, h - t);

// Recursively sort last 2/3 elements

stoogeSort(arr, l + t, h);

// Recursively sort first 2/3 elements

// again to confirm

stoogeSort(arr, l, h - t);

}

}

/\*\* 29. Tag Sort (To get both sorted and original) \*\*/

**public** **void** tagSort(Person persons[], **int** tag[]) {

**int** n = persons.length;

**for** (**int** i = 0; i < n; i++) {

**for** (**int** j = i + 1; j < n; j++) {

**if** (persons[tag[i]].getSalary() > persons[tag[j]].getSalary()) {

// Note we are not sorting the

// actual Persons array, but only

// the tag array

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

tag[i] = tag[j];

tag[j] = temp;

}

}

}

}

/\*\* 30. Tree Sort \*\*/

/\*\* create tree and print data in inorder traverse \*/

/\*\* 31. Cartesian Tree Sorting \*\*/

/\*\* 32. Odd-Even Sort / Brick Sort \*\*/

**public** **void** oddEvenSort(**int** arr[]) {

**boolean** isSorted = **false**; // Initially array is unsorted

**int** n = arr.length;

**while** (!isSorted) {

isSorted = **true**;

**int** temp = 0;

// Perform Bubble sort on odd indexed element

**for** (**int** i = 1; i <= n - 2; i = i + 2) {

**if** (arr[i] > arr[i + 1]) {

temp = arr[i];

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

arr[i + 1] = temp;

isSorted = **false**;

}

}

// Perform Bubble sort on even indexed element

**for** (**int** i = 0; i <= n - 2; i = i + 2) {

**if** (arr[i] > arr[i + 1]) {

temp = arr[i];

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

arr[i + 1] = temp;

isSorted = **false**;

}

}

}

**return**;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\* Sleep Sort use this class \*\*/

**class** SleepsortThread **implements** Runnable {

**private** **int** val;

**public** SleepsortThread(**int** val) {

**this**.val = val;

}

@Override

**public** **void** run() {

**try** {

Thread.*sleep*(val);

System.***out***.println(val);

} **catch** (InterruptedException e) {

// Oops...

}

}

}

/\*\* swap two element of array \*\*/

@Override

**public** **void** swap(**int** a[], **int** i, **int** j) {

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

a[i] = a[j];

a[j] = temp;

}

/\*\* min element in a array \*\*/

@Override

**public** **int** min(**int** a[]) {

**int** min = a[0];

**for** (**int** i = 1; i < a.length; i++) {

**if** (min > a[i])

min = a[i];

}

**return** min;

}

/\*\* max element in a array \*\*/

@Override

**public** **int** max(**int** a[]) {

**int** max = a[0];

**for** (**int** i = 1; i < a.length; i++) {

**if** (max < a[i])

max = a[i];

}

**return** max;

}

}

Test cases

=============================

**public** **class** ISortingAlgoTest {

**public** ISortingAlgo isa = **null**;

@Before

**public** **void** init() {

isa = **new** SortingAlgoImpl();

}

/\*\* 0. GuruJi Sort O(n) \*/

@Test

**public** **void** guruJiSortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

**int** b[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

isa.guruJiSort(a);

**for** (**int** i = 0; i < a.length; i++)

Assert.*assertTrue*(a[i] == b[i]);

**int** a1[] = { 1, 9, 2, 8, 1, 2, 5, 4, 3, 6, 8, 9, 6, 7, 3, 4, 5, 6, 4, 3, 2, 1, 8, 9, 7 };

isa.guruJiSort(a1);

System.***out***.println(Arrays.*toString*(a1));

}

/\*\* 1. Selection Sort \*\*/

@Test

**public** **void** selectionSortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.selectionSort(a);

System.***out***.println(Arrays.*toString*(a));

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

}

/\*\* 2. Bubble Sort \*\*/

@Test

**public** **void** bubbleSortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.bubbleSort(a);

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

// System.out.println(Arrays.toString(a));

}

/\*\* 3. Recursive Bubble Sort \*\*/

**public** **void** bubbleSortRecursiveTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.bubbleSortRecursive(a, a.length - 1);

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

// System.out.println(Arrays.toString(a));

}

/\*\* 4. Insertion Sort \*\*/

**public** **void** insertionSortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.insertionSort(a);

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

// System.out.println(Arrays.toString(a));

}

/\*\* 5. Recursive Insertion Sort \*\*/

@Test

**public** **void** insertionSortRecursiveTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.insertionSortRecursive(a, a.length);

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

// System.out.println(Arrays.toString(a));

}

/\*\* 6. Merge Sort \*\*/

@Test

**public** **void** mergeTwoPartSortedTest() {

**int** a[] = { 4, 5, 6, 7, 8, 9, 1, 2, 3 };

isa.mergeTwoPartSorted(a, 0, 5, a.length - 1);

// System.out.println(Arrays.toString(a));

Assert.*assertTrue*(a[0] == 1);

Assert.*assertTrue*(a[4] == 5);

Assert.*assertTrue*(a[8] == 9);

**int** b[] = { 1, 2, 3, 7, 8, 9, 4, 5, 6 };

isa.mergeTwoPartSorted(b, 4, 6, b.length - 1);

// System.out.println(Arrays.toString(b));

Assert.*assertTrue*(a[0] == 1);

Assert.*assertTrue*(a[5] == 6);

Assert.*assertTrue*(a[8] == 9);

}

@Test

**public** **void** mergeSortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.mergeSort(a, 0, a.length - 1);

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

// System.out.println(Arrays.toString(a));

}

/\*\* 7. Iterative Merge Sort \*\*/

@Test

**public** **void** mergeSortIterativeTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.mergeSortIterative(a);

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

// System.out.println(Arrays.toString(a));

}

/\*\* 8. Quick Sort \*\*/

@Test

**public** **void** quickSortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.quickSort(a, 0, a.length - 1);

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

// System.out.println(Arrays.toString(a));

}

/\*\* 9. Iterative Quick Sort \*\*/

/\*\* 10. Heap Sort \*\*/

/\*\*

\* arrange i index element greater than 2\*i+1 index and 2\*i+2 index element

\*\*/

@Test

**public** **void** maxHeapifyTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

**for** (**int** i = a.length / 2 - 1; i >= 0; i--) // build head so max is at 0

isa.maxHeapify(a, a.length, i);

// System.out.println("Max Heap:"+Arrays.toString(a));

Assert.*assertTrue*(a[0] == 9);

Assert.*assertTrue*(a[3] == 5);

Assert.*assertTrue*(a[8] == 0);

}

/\*\*

\* arrange i index element less than 2\*i+1 index and 2\*i+2 index element

\*\*/

@Test

**public** **void** minHeapifyTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.minHeapify(a, a.length, 0);

// System.out.println("Min heafipy"+Arrays.toString(a));

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[3] == 9);

Assert.*assertTrue*(a[8] == 5);

}

@Test

**public** **void** maxHeapSortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.maxHeapSort(a);

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

// System.out.println("Max Heap sort" + Arrays.toString(a));

}

@Test

**public** **void** minHeapSortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.maxHeapSort(a);

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

// System.out.println("Min heap sort" + Arrays.toString(a));

}

/\*\* 11. Counting Sort \*\*/

@Test

**public** **void** countingSortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.countingSort(a);

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

// System.out.println("Count sort" + Arrays.toString(a));

}

/\*\* 12. Radix Sort \*\*/

@Test

**public** **void** countingSortRadixTest() {

**int** a[] = { 170, 45, 75, 90, 802, 24, 2, 66 };

isa.countingSortRadix(a, 1);

Assert.*assertTrue*(a[0] == 170);

Assert.*assertTrue*(a[1] == 90);

Assert.*assertTrue*(a[2] == 802);

Assert.*assertTrue*(a[3] == 2);

Assert.*assertTrue*(a[7] == 66);

// System.out.println("Radix sort for for 1 digit:"+Arrays.toString(a));

}

@Test

**public** **void** radixSortTest() {

**int** a[] = { 170, 45, 75, 90, 802, 24, 2, 66 };

isa.radixSort(a);

Assert.*assertTrue*(a[0] == 2);

Assert.*assertTrue*(a[1] == 24);

Assert.*assertTrue*(a[2] == 45);

Assert.*assertTrue*(a[3] == 66);

Assert.*assertTrue*(a[7] == 802);

// System.out.println("Radix sort:"+Arrays.toString(a));

}

/\*\* 13. Bucket Sort \*\*/

@Test

**public** **void** bucketSortTest() {

**double** a[] = { 0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434 };

**double** b[] = { 0.1234, 0.3434, 0.565, 0.656, 0.665, 0.897 };

isa.bucketSort(a);

**for** (**int** i = 0; i < a.length; i++)

Assert.*assertTrue*(a[i] == b[i]);

}

/\*\* 14. ShellSort \*\*/

@Test

**public** **void** shellSortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

**int** b[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

isa.shellSort(a);

**for** (**int** i = 0; i < a.length; i++)

Assert.*assertTrue*(a[i] == b[i]);

}

/\*\* 15. TimSort \*\*/

@Test

**public** **void** timeSortTest() {

**int** a[] = { 5, 21, 7, 23, 19 };

**int** b[] = { 5, 7, 19, 21, 23 };

isa.timeSort(a);

**for** (**int** i = 0; i < a.length; i++)

Assert.*assertTrue*(a[i] == b[i]);

}

/\*\* 16. Comb Sort \*\*/

@Test

**public** **void** combSortTest() {

}

/\*\* 17. Pigeonhole Sort \*\*/

@Test

**public** **void** pigeonholeSortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.pigeonholeSort(a);

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

// System.out.println("Pigeonhole Sort" + Arrays.toString(a));

}

/\*\* 18. Cycle Sort \*\*/

@Test

**public** **void** cycleSortTest() {

**int** a[] = { 1, 8, 3, 9, 10, 10, 2, 4 };

isa.cycleSort(a);

System.***out***.println(Arrays.*toString*(a));

}

/\*\* 19. Cocktail Sort \*\*/

@Test

**public** **void** cocktailSortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.cocktailSort(a);

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

}

/\*\* 20. Strand Sort \*\*/

@Test

**public** **void** strandSortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.strandSort(a);

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

}

/\*\* 21. Bitonic Sort \*\*/

@Test

**public** **void** bitoicSortTest() {

**int** a[] = { 3, 7, 4, 8, 6, 2, 1, 5 };

// not working for below rray **TODO**

// int a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.bitonicSort(a, 0, a.length, 1);

System.***out***.println(Arrays.*toString*(a));

}

/\*\* 22. Pancake sorting \*\*/

@Test

**public** **void** pankakeSorteTest() {

**int** a[] = { 10, 5, 2, 3, 1, 0 };

isa.pankakeSorte(a);

System.***out***.println(Arrays.*toString*(a));

}

/\*\* 23. Binary Insertion Sort \*\*/

@Test

**public** **void** binaryInsertionSortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.binaryInsertionSort(a);

System.***out***.println(Arrays.*toString*(a));

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

}

/\*\* 24. BogoSort or Permutation Sort \*\*/

@Test

**public** **void** bogosortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.bogosort(a);

System.***out***.println(Arrays.*toString*(a));

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

}

/\*\* 25. Gnome Sort \*\*/

@Test

**public** **void** gnomeSortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.gnomeSort(a);

System.***out***.println(Arrays.*toString*(a));

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

}

// **TODO** not give proper result

/\*\* 26. Sleep Sort – The King of Laziness / Sorting while Sleeping \*\*/

@Test

**public** **void** sleepSortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.sleepSort(a);

}

/\*\* 27. Structure Sorting (By Multiple Rules) in C++ \*\*/

@Test // **TODO** for correct result

**public** **void** structureSortTest() {

**int** n = 5;

// array of structure objects

Student a[] = **new** Student[n];

**for** (**int** i = 0; i < a.length; i++)

a[i] = **new** Student();

// Details of Student 1

a[0].name = "Bryan";

a[0].math = 80;

a[0].phy = 95;

a[0].che = 85;

// Details of Student 2

a[1].name = "Kevin";

a[1].math = 95;

a[1].phy = 85;

a[1].che = 99;

// Details of Student 3

a[2].name = "Nick";

a[2].math = 95;

a[2].phy = 85;

a[2].che = 80;

// Details of Student 4

a[3].name = "AJ";

a[3].math = 80;

a[3].phy = 70;

a[3].che = 90;

// Details of Student 5

a[4].name = "Howie";

a[4].math = 80;

a[4].phy = 80;

a[4].che = 80;

isa.structureSort(a);

// computeRanks(a, n);

**for** (**int** i = 0; i < a.length; i++)

System.***out***.println("Rank : " + a[i].rank + " Name :" + a[i].name);

}

/\*\* 28. Stooge Sort \*\*/

@Test

**public** **void** stoogesortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.stoogeSort(a, 0, a.length - 1);

System.***out***.println(Arrays.*toString*(a));

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

}

/\*\* 29. Tag Sort (To get both sorted and original) \*\*/

@Test

**public** **void** tagSortTest() {

**int** n = 5;

Person persons[] = **new** Person[n];

persons[0] = **new** Person(0, 233.5f);

persons[1] = **new** Person(1, 23f);

persons[2] = **new** Person(2, 13.98f);

persons[3] = **new** Person(3, 143.2f);

persons[4] = **new** Person(4, 3f);

**int** tag[] = **new** **int**[n];

**for** (**int** i = 0; i < n; i++)

tag[i] = i;

// Every Person object is tagged to

// an element in the tag array.

System.***out***.println("Given Person and Tag ");

**for** (**int** i = 0; i < n; i++)

System.***out***.println(persons[i] + " : Tag: " + tag[i]);

// Modifying tag array so that we can access

// persons in sorted order.

isa.tagSort(persons, tag);

System.***out***.println("New Tag Array after " + "getting sorted as per Person[] ");

**for** (**int** i = 0; i < n; i++)

System.***out***.println(tag[i]);

// Accessing persons in sorted (by salary)

// way using modified tag array.

**for** (**int** i = 0; i < n; i++)

System.***out***.println(persons[tag[i]]);

}

/\*\* 30. Tree Sort \*\*/

/\*\* create tree and print data in inorder traverse \*/

/\*\* 31. Cartesian Tree Sorting \*\*/

/\*\* 32. Odd-Even Sort / Brick Sort \*\*/

@Test

**public** **void** oddEvenSortTest() {

**int** a[] = { 1, 0, 2, 9, 3, 8, 7, 4, 5, 6 };

isa.oddEvenSort(a);

System.***out***.println(Arrays.*toString*(a));

Assert.*assertTrue*(a[0] == 0);

Assert.*assertTrue*(a[5] == 5);

Assert.*assertTrue*(a[9] == 9);

}