1. The order would be Since in the worst case scenario, the inversions would be opposite to each other.

public long merge(int[] a, int[] aux, int lo, int mid, int hi) {

long inversions = 0;

*// copy to aux[]*

for (int k = lo; k <= hi; k++) {

aux[k] = a[k];

}

*// merge back to a[]*

int i = lo, j = mid+1;

for (int k = lo; k <= hi; k++) {

if (i > mid)

a[k] = aux[j++];

else if (j > hi)

a[k] = aux[i++];

else if (aux[j] < aux[i]) {

a[k] = aux[j++]; inversions += (mid - i + 1); }

else

a[k] = aux[i++];

}

return inversions;

}

public long count(int[] a, int[] b, int[] aux, int lo, int hi) {

long inversions = 0;

if (hi <= lo) return 0;

int mid = lo + (hi - lo) / 2;

inversions += count(a, b, aux, lo, mid);

inversions += count(a, b, aux, mid+1, hi);

inversions += merge(b, aux, lo, mid, hi);

assert inversions == brute(a, lo, hi);

return inversions;

public static long distance(int[] a, int[] b) {

if (a.length != b.length) {

throw new IllegalArgumentException("Array dimensions disagree");

}

int n = a.length;

int[] ainv = new int[n];

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

ainv[a[i]] = i;

Integer[] bnew = new Integer[n];

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

bnew[i] = ainv[b[i]];

return Inversions.count(bnew);}

1. To sort S, do a radix sort on the n elements, viewing them as pairs (i, j) such that i and j are integers in the range [0, n − 1].
2. We will assume that the priority queue can be considered a min heap (though it is not neccessarily so) where each node stores a distinct number in S, called its key. And, each node's key is always greater than its parents.

Therefore, to we will need to perform a series of comparisons to ensure the new node is placed appropriately within the 'heap'.

To we can use an operation of pulling off our 'heap's root and bubbling as neccessary.

Therefore we will attempt to prove that, in a comparison based implementation following from the above, that requires time.