**//steven guo**

**//insertion sorting program**

**//10/20/19**

**import** java.util.Random;

**public** **class** insertionSort {

**static** **void** sort(**int** arr[])

{

**int** n = arr.length;

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

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

**int** j = i - 1;

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

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

j = j - 1;

}

arr[j + 1] = key;

}

}

//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];

*sort*(*randArray*(SIZE, arr1));

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

time = System.*nanoTime*();

*sort*(*randArray*(SIZE1, arr2));

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

time = System.*nanoTime*();

*sort*(*randArray*(SIZE2, arr3));

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

time = System.*nanoTime*();

*sort*(*sortedArray*(SIZE, arr1));

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

time = System.*nanoTime*();

*sort*(*sortedArray*(SIZE1, arr2));

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

time = System.*nanoTime*();

*sort*(*sortedArray*(SIZE2, arr3));

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

time = System.*nanoTime*();

*sort*(*almostSortedArray*(SIZE, arr1));

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

time = System.*nanoTime*();

*sort*(*almostSortedArray*(SIZE1, arr2));

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

time = System.*nanoTime*();

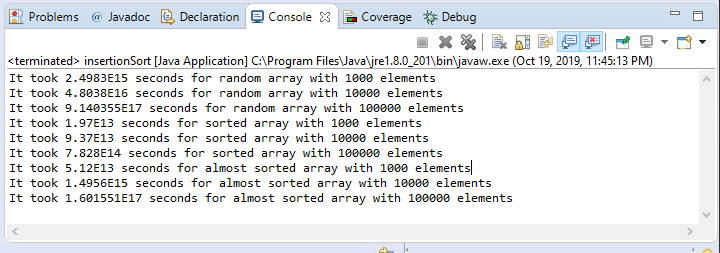
*sort*(*almostSortedArray*(SIZE2, arr3));

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

time = System.*nanoTime*();

}

}



The time complexity of insertion sort is O(n\*2). The insertion sort takes an element from the input array and removes it. It then finds the location where the element belongs within the sorted list. And it keeps repeating until there are no elements from the input array left. The worst case is when the input elements are in reverse order(descending order). The best case is when the elements are already sorted. Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array. In theory, the insertion sort should run faster in all cases when the input array is already sorted. My results however are true in some cases. When the input array had 1000 elements and is sorted, it ran faster than if the input array was randomized and if the array was almost sorted. When the input array had 10000 elements and is sorted, it ran slower than if the input array was randomized and if the array was almost sorted. When the input array had 100000 elements and is sorted, it ran faster than if the input array was randomized and if the array was almost sorted. I think the reason it took 9 second for a sorted array of 10000 elements is because it could be my lagging or something. There is no way it should take longer than a sorted array with 100000 elements.