**Nonuniform data**

Write a client that generates test data that is not uniform,

including the following:

■ Half the data is 0s, half 1s.

■ Half the data is 0s, half the remainder is 1s, half the remainder is 2s, and so forth.

■ Half the data is 0s, half random int values.

Develop and test hypotheses about the effect of such input on the performance of the algorithms in this section.

**Insertion Sort Algorithm**

This is an in-place comparison based sorting algorithm. For example, the part of an array is maintained to be sorted. Insertion sort is “Stable”. An element that is going to be inserted has to find its appropriate place and then it has to be inserted there. In insertion sort elements are known beforehand while location to place them is searched. Best case time complexity is O (n).

**Selection Sort Algorithm**

In a selection sort, for each element that is to be added to the sorted section, you have to scan the entire unsorted part of the list to find the minimum remaining element. The process of searching the minimum key and placing it in the proper position is continued until the all the elements are placed at right position. Selection sort is “Unstable”. Best case time complexity is O (n^2).

**Half the data is 0s, half 1s**

When the given test case is exactly half 0’s and half 1’s.

For example **101010101010101010100110101010101010101010010101101010010101101001010101010101010101011010100101011010010101010101010101**

To sort this using insertion sort algorithm we need traverse all the elements in the given array and swap.

Time taken to sort an array of half the data is 0’s and half 1’s is

For Insertion Sort it is **0.021** **Seconds**

For Selection Sort it is **0.023** **Seconds**

As we have only 0’s and 1’s we sort the data easily because we knew the data in insertion sort. But in selection sort though it is 0 and 1 but we need to search all the elements to find the minimum element in the given array.

After Sort:

**000000000000000000000000000000000000000000000000000000000000111111111111111111111111111111111111111111111111111111111111**

**Half the data is 0s, half the remainder is 1s, half the remainder is 2s, and so forth.**

When the given array containing half 0’s and the rest half is the remainders of 1’sm half the remainders of 2’s and so on.

For example

**30000030001000100111000121141001000112021202022303**

For Insertion Sort it is **0.006 Seconds**

For Selection Sort it is **0.007 Seconds**

In general, insertion sort will perform fewer comparisons than selection sort so the time taken by insertion was less compared to selection.

**After Sort:**

**00000000000000000000000001111111111111222222233334**

**Half the data is 0s, half random int values.**

When the given array containing half 0’s and the rest half is random integers from [1-9].

For example

**010203040500102030405040508090810305080904050809081030508090**

For Insertion Sort it is **0.007** Seconds

For Selection Sort it is **0.007** Seconds

For this both insertion and selection sort algorithm works same as we have random numbers we need to compare each and every element with the least element and swap according to it.

After Sort:

**000000000000000000000000000000111122333344445555558888889999**