(c) ***How do the different values of k change the best – average – worst case asymptotic time complexities for this variant?***

To build the graph I took different values of *n* (size of array) and *n/k* (subsequences) as the problem says (a). There are four graphs; One for each cases and also one for all of them (*not much recognizable).

Best time:

For small values of N there is not much change (most of the cases it was 0.00 sth). For greater values of *n* we can see how the value of *k* affects the time complexity *i.e.:* for n=12500 as the k increases the time complexity becomes better. This happens because it is easier for the *insertion sort algorithm* to sort the elements then the *merge sort algorithm* as there are more subsequences where insertion sort can be implemented and the elements are already sorted so there is not much help from the merge sort. (since time complexity of merge sort is nlogn and insertion sort is n for the best case)

Worst time:

In worst case the time complexity of *insertion sort algorithm* is n^2 while *merge sort algorithm* time complexity is *nlogn.* When we increment *k* we implement more insertion sort which affects negatively in the time complexity of algorithm. For small values of *n* this may not be that much visible but for much more greater values there is a more clearly effect of *k* in algorithm. (Didn't include it in the graph as the other values would not be able to be seen)

Average time:

As all the values are generated by a random-number function we know that the time complexity for merge sort is *nlogn* and insertion sort time complexity is n^2. Insertion sort has a factor of *n* in its running time (if we write it as n*n) and merge sort's running time is *logn* (if we write it as n*logn). Although insertion sort usually runs faster for small input sizes N, once the input becomes large enough merge sort's algorithm becomes much faster (as *logn* is smaller than *n*). From the graph we can see that for small values of *n*, the time complexity is better implementing greater values of *k*, and once the *k* becomes larger the time complexity becomes worst. While for big values of *n* merge-sort is more efficient and when *k* increases the time complexity of the algorithm becomes worst. (more insertion sort is implemented)

(d) ***Based on the results from (b) and (c), how would you choose k in practice?***

It always depends on the case. In practice we usually face random numbers and supposing that the input is large big value of *k* is much efficient as the merge-sort is faster than insertion sort for big inputs.

Related to my case where the size varies between 1000 – 12500 and the numbers are random as they mostly are in practice I would choose **80<= k <=100**.

While in most cases related to just the size of input I would choose k bigger for greater inputs and smaller for smaller inputs.