D0012E Lab 1

Fredrik Larsson
email frelau-1@student.ltu.se

Mikkjal Erik Gardshorn
email mikgar-1@student.ltu.se

Josef Thurfjell
email jostuh-1@student.ltu.se

January 30, 2023

1 Introduction

The purpose of the lab assignment is to create the following algorithms and measure the disparities in their performance: *bSort*, *Insertion Sort* and *Merge Sort*.

2 Code

2.1 Linear Insertion Sort

The *Insertion Sort* algorithm was created like this.

1. Iterate from the second item in the list to the last. The reason it begins at the second element is that we assume that the first element is always sorted
2. Then take the next element in the list and store in a *key* variable
3. Then compare the *key* element to the first element in the list, if the element is larger then place it after, if not place it before.
4. Then continue to compare the key with the previous element until an element that is smaller is found

2.2 Binary search

A *Binary search* algorithm was created since it is needed in the BSort algorithm.

1. A function that takes a list, a number, lowest index, and highest index.
2. Then a mid point variable is created using the sum of the highest and lowest values floor divided with two.
3. Then it proceeds to check if the list value at the mid point index is equal to the number that is being searched for. If it is then it returns the mid number which is the index where the number is located
4. Then it checks if the list value at the mid point index is less than the number that is being searched for. If it is, then recursion is performed with a new lowest value that is the mid value plus one
5. Then if neither of those if statements are true it means that the list value at the mid point index is larger than the number that is being searched for. And a recursion is performed with a new highest value that is the mid value minus one

2.3 bSort

The *bSort* algorithm was created in the following way

1. Goes through the list from 1 to the end since we assume the first index is sorted
2. Then take the next element in the list and store in a *key* variable
3. Then it gets the index for where the key variable should be placed using the binary search method, with the key as the number to be searched for with index 1 as the high index value.
4. Then changes the list like this: list = list from start until the index for key variable, add the key, add the rest of the list

2.4 Modified Merge Sort

The *Merge Sort* algorithm functions in a way to minimize resource uptake through dividing the input into smaller sublists. This is to reduce the amount of repetition the algorithms perform in order to sort a larger dataset, as the time required to perform both bSort and Insertion Sort goes up by a large factor as the input size increases. The function works in the following manner

1. If the dataset is larger than a predetermined size then split the dataset in half
2. Call the function again for each sublist until the requested size has been met
3. When the sublists are of an appropriate size they are sorted using either bSort or Insertion Sort, respectively
4. Proceed to merge each layer of sorted sublists, resulting in a completely sorted dataset when returning to and ending the first iteration

3 Results

3.1 Types of datasets used

The tests used three different types of lists, each of 30000 elements numbering from one to 30000. These were made in the following ways:

1. Sorted lists: A fully sorted list with the lowest number first and the largest number last.
2. Random lists: A randomized list where a list was generated in a sorted order, with a random element of it inserted into a new list to be returned.
3. Unsorted lists: A fully sorted list with an argument determined number of switches between two random elements, thus unsorting the list. The unsorted lists were mixed up at a rate of 10% of the array's size.

3.2 Insertion Sort vs bSort

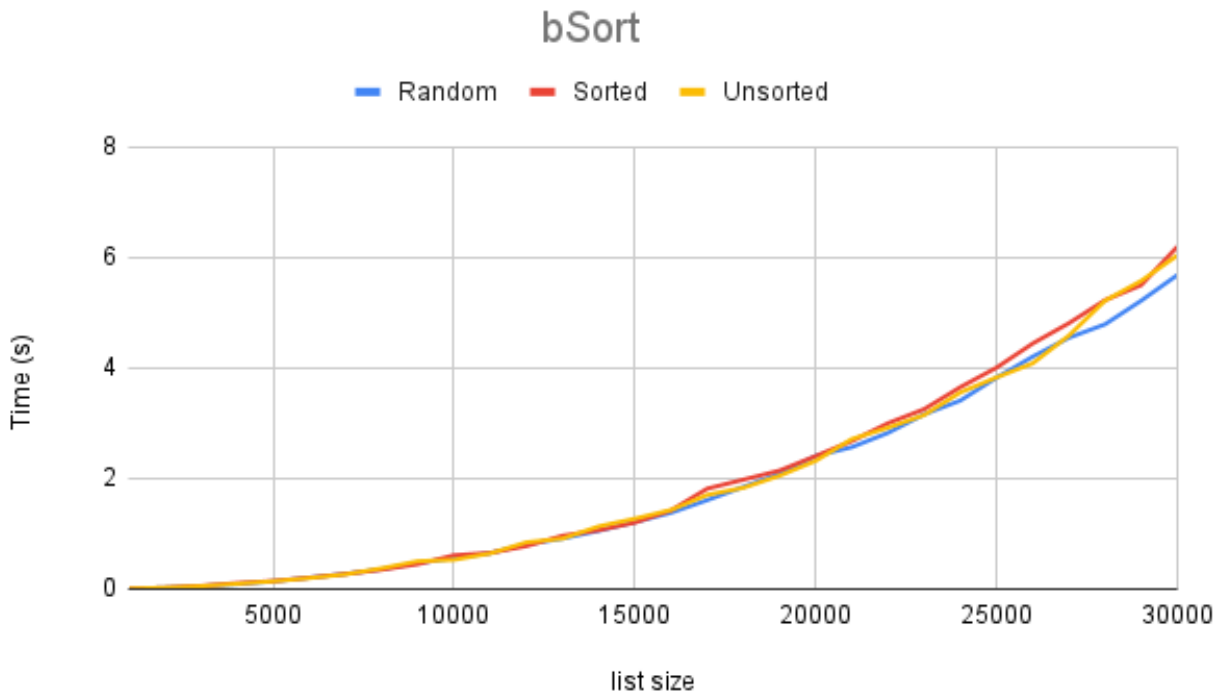

Figure 1: bSort run time

Figure 2: Insertion Sort run time

Comparing the results in Figure 1 and in Figure 2 it shows clearly that *bSort* is faster for a random list, while for Unsorted lists both *bSort* and *Insertion Sort* preformed about the same and lastly for sorted list *Insertion Sort* was clearly the fastest algorithm.

The reason for this is that for **sorted arrays** *insertion sort* only has to go through the list once and it does not need to perform any switch operations since the array was already sorted. On the other hand, when running on *bSort* it has to go through the list and perform binary search (without switching the items) on each element in the array, which makes it slower than only looping through the array. Time complexity wise this means that *Insertion sort* has O(n) while *bSort* has O(N*log(n)).

For **random arrays** *Insertion sort* has to go through the list and swap the unsorted elements. *bSort* has to go through the list and perform binary search and swap them. Time complexity wise them both have $O(n^2)$ in the worst case, while an average for *bSort* would be around O(N*log(n)). However, since *bSort* uses binary search it requires much less comparisons than *Inserion sort* which makes it faster by a large margin, as well as the binary searching mechanism of *bSort* working much faster and much more consistently as the worst or even bad cases are very rare. On the other hand, **random arrays** will often result in *Inserion sort* making many comparisons and performing a large number of backtracking through the array.

For **unsorted arrays** *Insertion sort* has to go through the list and swap some elements, a few elements are already sorted which means that it does not need to swap them. *bSort* has to go through the list and perform binary search and swap them. Both of the algoirthms have up- and

downsides on unsorted arrays and that is why the performance disparity is much lower than on fully randomized arrays. Time complexity wise they both have O(n^2) in the worst case.

3.3 *Merge sort*

Merge sort was tested using both *Insertion Sort* and *bSort*, with various values on the k variable ranging from 10 to 3500 and various length of the randomly generated data sets. The results differed depending on the type of data used and the sorting algorithm. Insertion Sort performed best on fully sorted lists whereas randomized datasets resulted in bSort performing better by over two times at a K-value of 3500.

It should also be noted that *bSort* sorts faster than *Insertion Sort* on large data sets, but it doesn't matter on *Merge Sort* as it always splits the array into smaller pieces. This was concluded from the experiment below.

3.4 *K value*

The results from testing *Merge Sort* with both *Insertion Sort* and *bSort* with a randomly generated array with the sample size of 30000 are illustrated through the following graphs, including graphed lines for the Merge Sort algorithm sorting through *bSort* or *Insertion Sort* respectively, as well as different types of datasets:

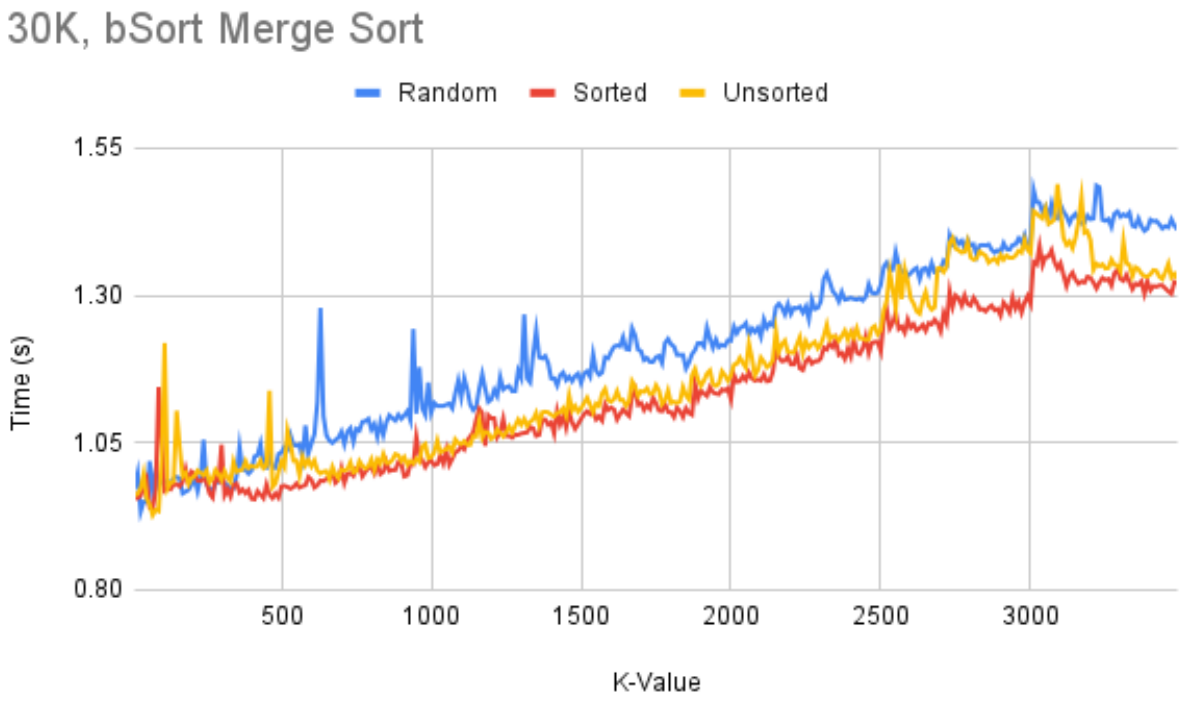

Figure 3: BSort K value

The graph clearly shows that the sorting algorithm increases resource allocation regardless of the type of dataset which is used, although randomized datasets seem to have a larger impact.

Over a steady increase in the value of K the time increased. This coincides with the increase in time the regular *bSort* algorithm has in isolation through an increase in the size of data used.

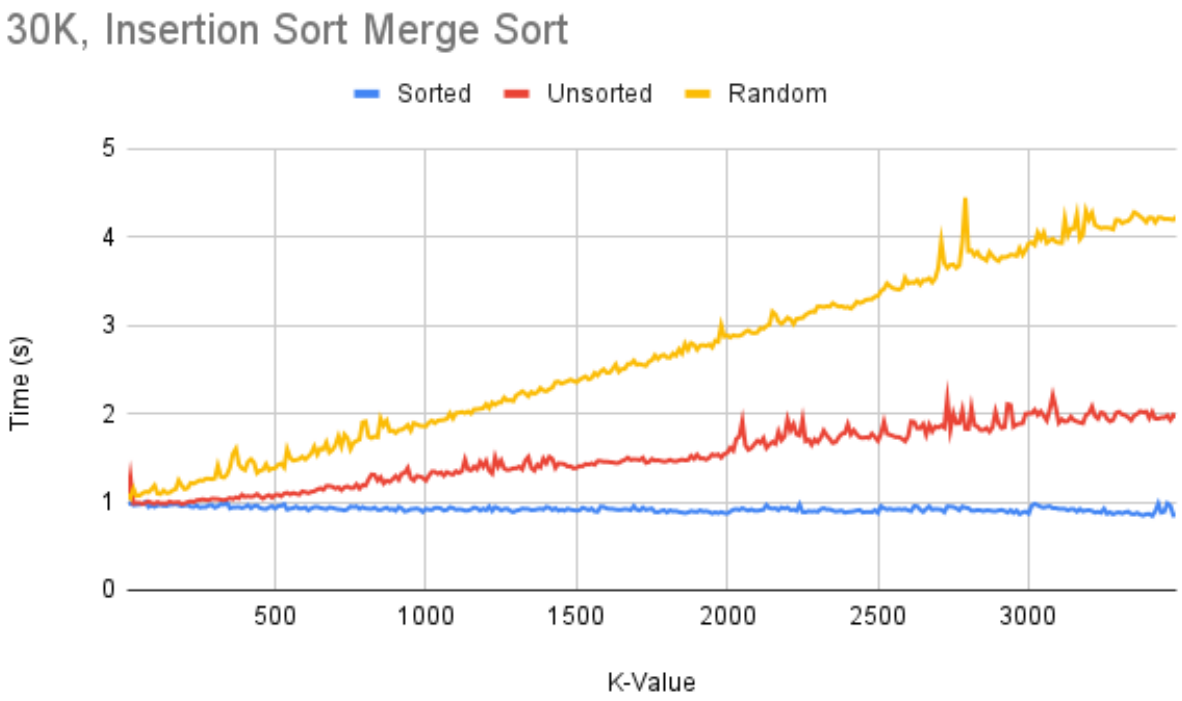

Figure 4: Insertion Sort K value

Insertion Sort resulted in the fastest speeds on a fully sorted list, as well as major increases in time as the *K-value* increases on both random and unsorted lists.

Testing *Merge Sort* using *Insertion Sort* showed that the best *K value* wasn't a clear number, but more a lower number as the following figure 2 shows.

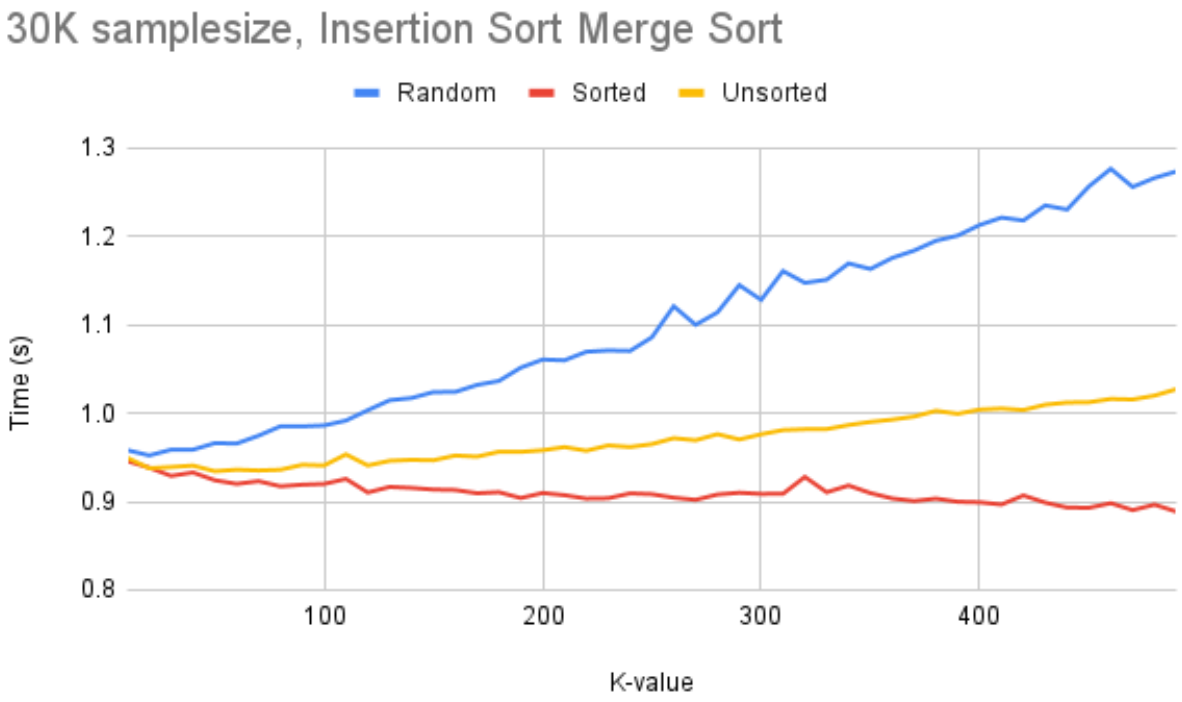

Figure 5: BSort K value

Testing *Merge Sort* using *BSort* showed that the best *K value* was 60, although the range between 30-300 gave negligible differences.

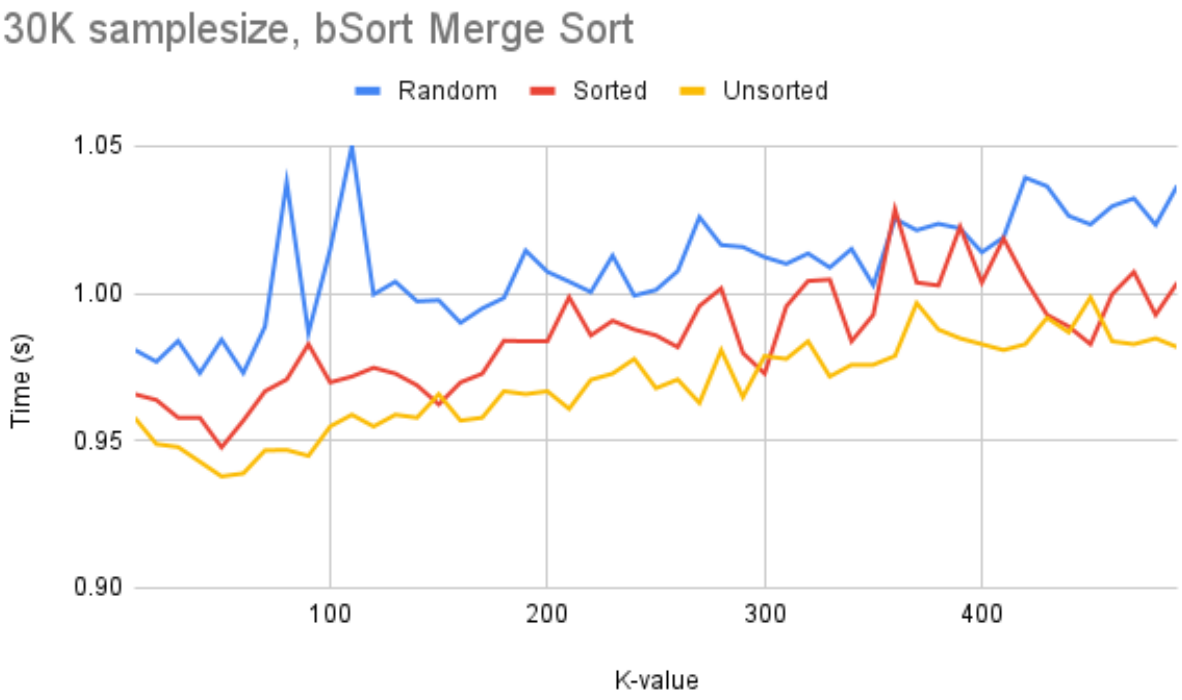

Figure 6: BSort K value

Overall *Merge Sort* with *bSort* was faster than *Merge Sort* with *Insertion Sort*.

The only outlier from the increase over K-value was the sorted list running on Merge Sort through *Insertion Sort*. The following table illustrates the difference between running through a sorted list using *Insertion Sort* in a *Merge Sort* and alone, respectively. The following table illustrates this:

List Size	With Merge	Standalone
6000	0.040864	0.000997
12000	0.160464	0.000996
18000	0.340860	0.001997
24000	0.591023	0.002990
30000	0.904977	0.003985

4 Discussion

In most of the cases the time difference between *Insertion Sort* and *bSort* being used in *Merge Sort* was one of a few seconds, and we came to the conclusion that the difference was due to the algorithm's efficiency and sample size. Splitting the dataset into smaller subsets of data greatly improved performance and resulted in a major reduction in the difference in runtime, as *Insertion Sort* struggled over larger randomized datasets. The difference in execution time for each sorting algorithm within the respective *Merge Sort* functions was visible due to the aforementioned factors.

The outlier from that conclusion would be the insertion sort's run time as K increases, as a fully sorted list will not force the algorithm to sort backwards. This results in the time difference decreasing on average, as the program spends less resources on splitting and combining lists.

We found many variations in the multitude of executions of the algorithms, but with clear trends towards an increase in running time as the K-value increased. Our conclusion was after several runs of the algorithms that the computer will offer differing results each execution, however there will be a clear trend across several executions where a higher K-value resulted in longer running time. As a higher value results in larger sublists, the conclusion was accurate with the ways both bSort and Insertion Sort work, which is a larger dataset taking disproportionately more resources to sort than a smaller one.