Lab report 1 - Insertsort

Group 117
Jonatan Trefil, jontre-0@student.ltu.se
Rasmus Jacobsen, rasjac-9@student.ltu.se
Edvin Pettersson, edvpet-0@student.ltu.se

November 18, 2021

1 Introduction

In this lab we are suppose to test different sorting algorithms, and if they are able to become more effective. The different algorithms that we are testing are

merge sorting with a binary sorter for help,

merge sorting with a linear sorter for help,

and a regular merge sorter.

We will test these algorithms with different kinds of large randomized arrays that are completely random, almost sorted, and sorted.

2 Theory

2.1 Insertion Sort

The linear algorithm is Insertionsort. The algorithm works by inserting a number in it correct position by linearly going through the list index. We start from low to high index, where the left side of index j will be sorted and the right side unsorted. A detailed description of the algorithm can be seen in the step by step.

1.	We gather the size of the list n.
2.	We define j as 1. Representing an index number of the list.
3.	Set k as equal to j.
4.	If k is larger than 0 and item in index $k-1 > k$, Perform step 5. Otherwise jump to step 6.
5.	Swap index for item $k-1$ and k. Decrement k by 1. Go to step 4.
6.	Increment j by 1 and if $j < n$ go to step 3.
7.	Done!

Normally an array would be used as it as a higher memory efficiency and we do not require any additional features from more complex memory structures. This is because we only operate with a fixed size of elements and only swap elements in the list. The reason this is not used is for compatibility with Merge Sort and Binary Sort that use double linked list for mentioned reasons in their respective descriptions. The algorithm is based on iterations. This is done as we beforehand know that we will be required to do comparisons with all elements.

2.2 Binary Sort

Binary sort is based upon linear sort where the linear search technique is replaced with a binary search. When we want to sort a selected element, we compare it the middle of the sorted list and if it is larger we compare it to the

middle between the current position and the rightmost index. The opposite is true if it is smaller. This is done until the element to the left is smaller and the element to the right is larger. A better explanation can be seen in the step by step.

1.	bSort takes a list
2.	Check IF the input list is 1, THEN we return the list as is.
3.	ELSE order the 2 first elements by size in sList.
4.	Check IF the input list is 2, THEN we return the sList.
5.	Take out the next element X from the inputed list.
6.	Create 3 pointers: LeftPointer (LP), points to the begining of the sList MiddlePointer (MP), points to the middle of the sList RightPointer (RP), points to the end of the sList
7.	IF X is equal or smaller then MP THEN move RP to MP
8.	ELIF X is equal or larger then MP THEN move LP to MP
9.	Set MP to the middle of LP and RP.
10.	IF LP and RP is 1 from each other ($LP + 1 == RP$), THEN insert X in sList between LP and RP, Skip step 11.
11.	Repeat step $7 - 10$.
12.	Repeat step $5 - 11$ until all elements have been worked through
13.	Return the sorted list (sList)
14.	Done!

We use a double linked list data structure to enable element insertion in the list. This is requirement from Binary sort while the binary search also work with arrays. We need insertion as otherwise we would need to move all other elements when sorting a single element. This would make it inefficient on larger data sets and make it similar to Insertion Sort.

An advantage with using list is that it is the standard data structure in python and using arrays would require more code in the sense of the uncompiled file.

This algorithm is also based on iteration for similar reasons as mentioned with Insertion Sort. The change to this we can think of is separating the algorithm into 2 components. An iterative binary sort function and a recursive binary search function. An recursive implementation might be easier to read and make the binary sort component easier to understand.

2.3 Merge Sort

Merge sort is an algorithm that divides the unsorted list into equal sized sublists that are then sorted by either binary search or linear search. These sublists are then merged sorted after each sublist is created. The different Merge Sort Algorithms are identical except for the differences of what function is called when sorting. Merge sort can be more thoroughly explained in the step by step.

1.	Divide the array into equal sub parts with the size k.
2.	Sort the sub parts with binarySort or linearSort.
3.	Take the sorted sub parts and compare them two by two.
4.	Add the smallest element from the sub parts to a new list
5.	Repeat step 4 until all elements from the sub parts have been added to the new list.
6.	Repeat step $3-5$ until all sub parts have been sorted into the new list.
7.	Return the new list.
8.	Done!

For the Merge algorithms we use lists as our data structure to enable easier implementation as as memory or exact cycle usage is not a requirment for this assigment. We also use it to enable compatability with Binary Sort.

3 Results

The results have been obtained by creating a testing program in python that performs the algorithm with increasing data size and k value size. The test data is then saved unto text file. The following is best case for all following conditions. The full measured test can be viewed in the text file "RESULTS FINAL".

Merge BinarySort Iterative Unsorted

k	Input length	Validated	Hours	Minutes	Seconds
2^{12}	2^{14}	True	0	0	0.381031...

Merge LinearSort Unsorted

k	Input length	Validated	Hours	Minutes	Seconds
2^{9}	2^{14}	True	0	0	2.810414...

Pure MergeSort Unsorted

Input length	Validated	Hours	Minutes	Seconds
2^{14}	True	0	0	0.456032...

Merge BinarySort Iterative Sorted

k	Input length	Validated	Hours	Minutes	Seconds
2^{13}	2^{14}	True	0	0	0.088006...

Merge LinearSort Sorted

k	Input length	Validated	Hours	Minutes	Seconds
2^{13}	2^{14}	True	0	0	0.052005...

Pure MergeSort Sorted

Input length	Validated	Hours	Minutes	Seconds
2^{14}	True	0	0	0.237015...

Merge BinarySort Iterative Partially Sorted

k	Input length	Validated	Hours	Minutes	Seconds
2^{13}	2^{14}	True	0	0	0.4120380...

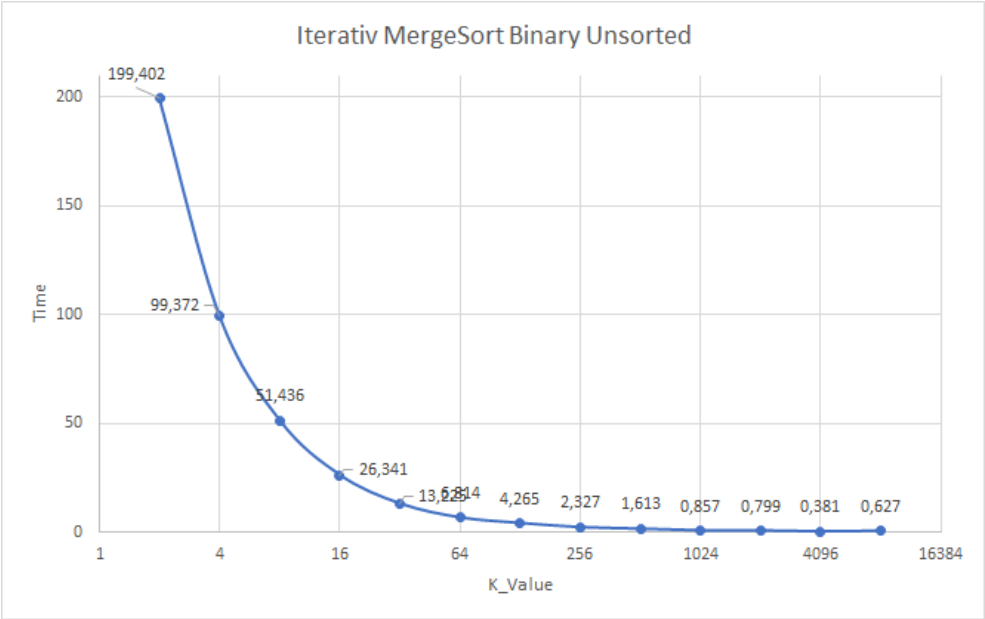
Merge LinearSort Partially Sorted

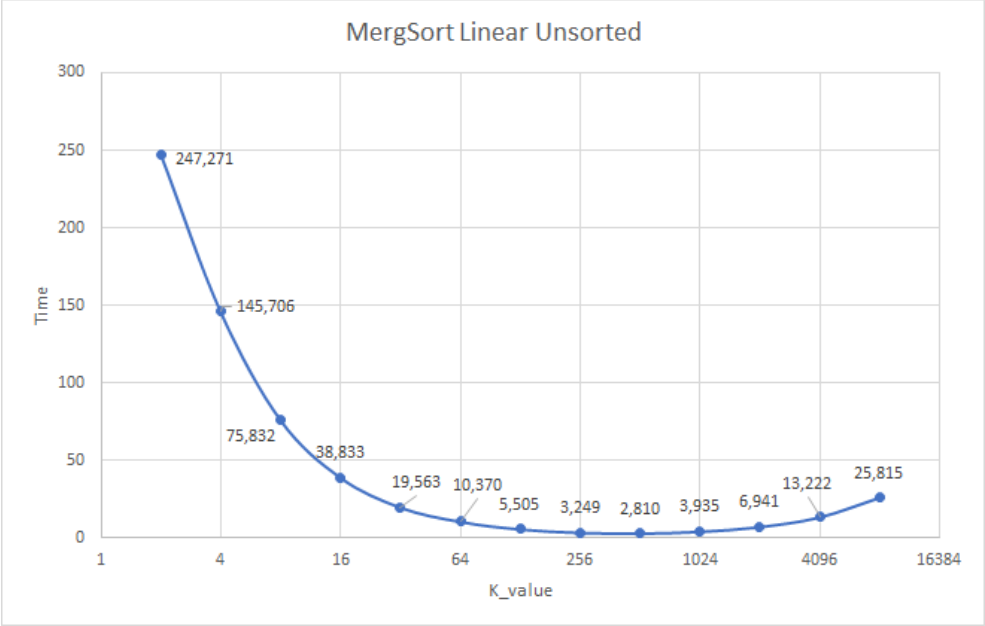
k	Input length	Validated	Hours	Minutes	Seconds
2^{9}	2^{14}	True	0	0	2.3122847...

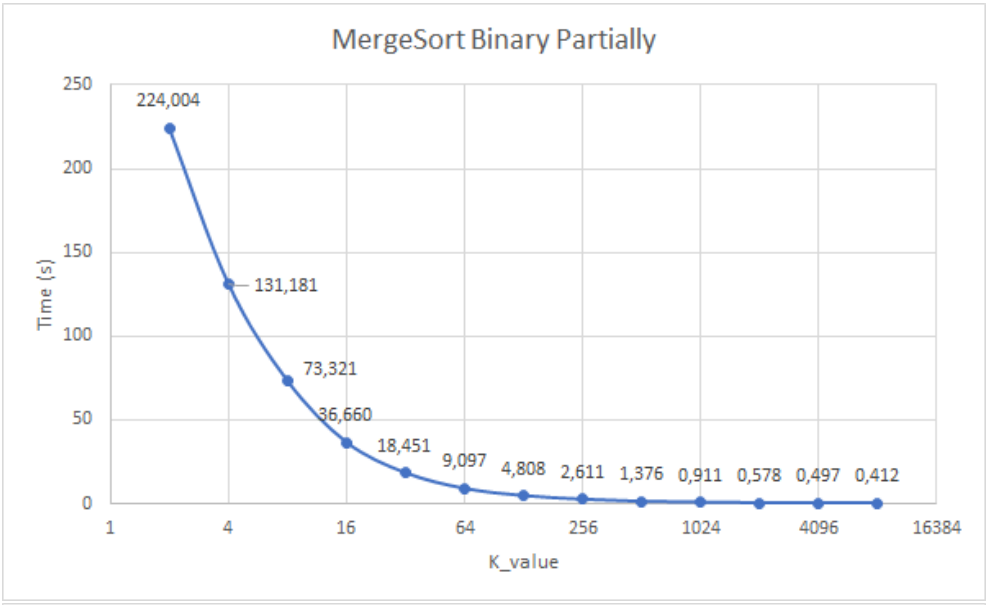
Pure MergeSort Partially Sorted

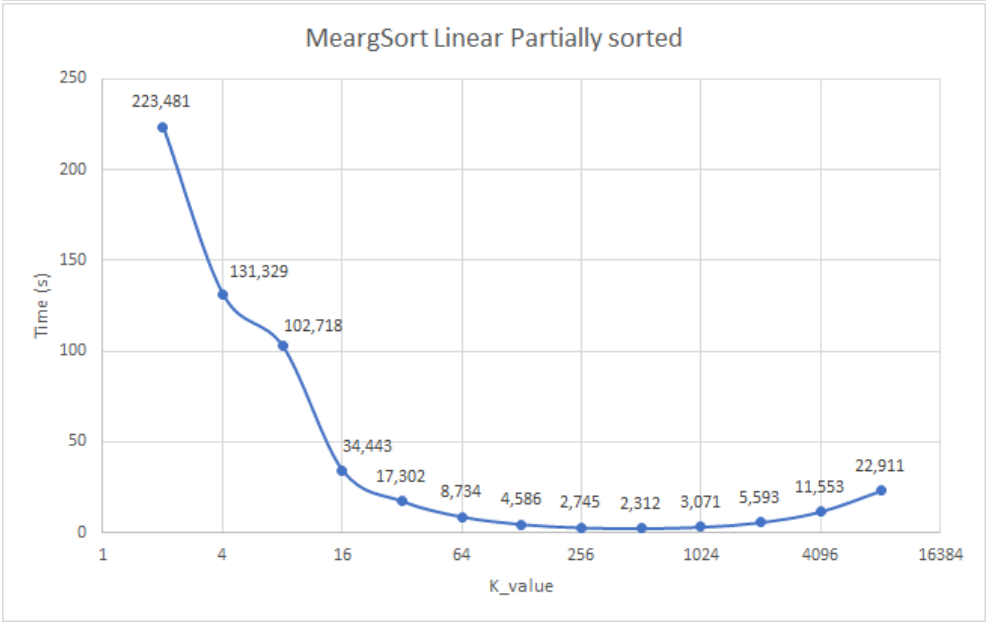
Input length	Validated	Hours	Minutes	Seconds
2^{14}	True	0	0	0.406019...

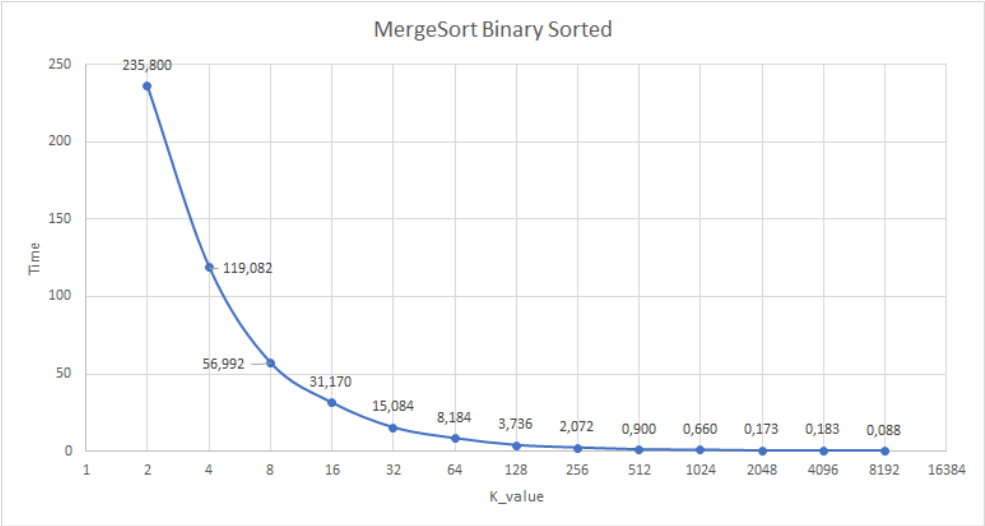
The following is the test data inserted into excel and visualised onto graphs.

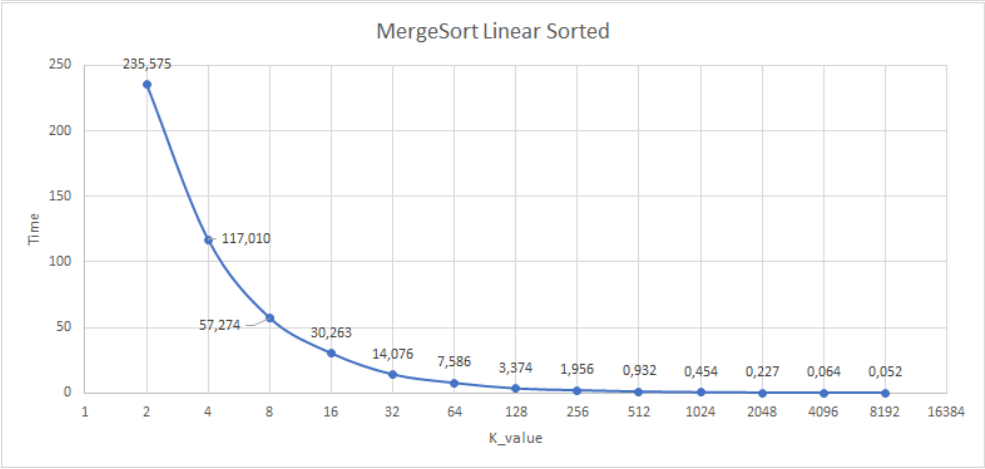

Iterativ MergeSort Binary Unsorted
Time
K_Value
199,402
99,372
51,436
26,341
13,225
5,814
4,265
2,327
1,613
0,857
0,799
0,381
0,627
0
50
100
150
200
1
4
16
64
256
1024
4096
16384


MergSort Linear Unsorted
Time
K_value
247,271
145,706
75,832
38,833
19,563
10,370
5,505
3,249
2,810
3,935
6,941
13,222
25,815
0
50
100
150
200
250
300
1
4
16
64
256
1024
4096
16384


MergeSort Binary Partially
Time (s)
K_value
224,004
131,181
73,321
36,660
18,451
9,097
4,808
2,611
1,376
0,911
0,578
0,497
0,412


MeargSort Linear Partially sorted
Time (s)
K_value
223,481
131,329
102,718
34,443
17,302
8,734
4,586
2,745
2,312
3,071
5,593
11,553
22,911


MergeSort Binary Sorted
Time
K_value
235,800
119,082
56,992
31,170
15,084
8,184
3,736
2,072
0,900
0,660
0,173
0,183
0,088
250
200
150
100
50
0
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384


MergeSort Linear Sorted
Time
K_value
235,575
117,010
57,274
30,263
14,076
7,586
3,374
1,956
0,932
0,454
0,227
0,064
0,052
250
200
150
100
50
0
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384

4 Summary

From the data results we can ascertain that Merge BinarySort Iterative is the fastest algorithm when dealing with a unsorted list. The optimal k value for it is 2^{12}.

When dealing with a sorted list Merge LinearSort is the fastest with a k value of 2^{13}. This is the upper bound of k for our test so the actual value could be higher. This is attributed to the linear sort failing the while loop on the first instance of each element of the list when sorting.

For the partially sorted lists Pure MergeSort was the fastest. We have difficulty with knowing exactly what this is attributed to.