LULEÅ UNIVERSITY OF TECHNOLOGY

# Algorithms and Data Structures - Laboration 1

## Course: D0012E

Marcus Lund (amuulo-4)
Edvin Åkerfeldt (edvker-4)
Samuel Karlsson (samkar-4)
November 23, 2015

First submission.

**Abstract**

We have implemented the two sorting algorithms, insertionsort and binsertionsort. Binsertionsort is a variation of insertionsort where you use a binary search to find the place for any new number. The regular insertionsort use linear search instead of binary search. We also implemented a merge sort that divide the array into sub-arrays of size equal or less the $k$ ($k$ is a positive integer). The sub-arrays are then sorted by insertionsort or binsertionsort. Finally we merge the sorted sub-arrays back together using standard merge sort algorithm.

We compared the run-times when using insertinonsort and binsertionsort. We found that insertionsort's runtime increases logarithmic while the binsetrionsort takes less time, it also resembles a logarithmic function but it increases much slower. The runtime of insertionsort increases by a factor of two for each time $k$ has been doubled. This is dune because of the merge sort that split the array in half and give insertionsort twice the amount of work. In small array sizes, insertionsort may be faster than binsertionsort. These cases has an array of a very small size. Both algorithms runs in $\theta(nk + n\lg(\frac{n}{k}))$ worst case time. Generally, binsertionsort is much faster than insertionsort in average case.

# CONTENTS

# 1 Introduction

We began by looking into how the two algorithms would be implemented. Then both algorithms were then implemented in our language of choice, Python. The first implementation was sloppy and slow, but after a few improvements to the code (reducing function calls and comparisons) we managed to get a faster and better implementation overall while keeping the functionality of the algorithms intact.

# 2 Language

We choose Python as our language of choice because of the easy implementation of array splitting. Another contributing factor was the general easy to use syntax. We are aware of the fact that it is ugly compared to other languages due to the fact that there is little structure in the code syntax itself. Although we decided to choose this language we are also aware of it's drawbacks, such as: slow execution time compared to C++ or java, low recursion depth compared to other languages and the lack of variable declarations (implies more error checks). All together we decided that Python was a good choice due to the big advantages the array splitting gives in terms of easy implementation.

# 3 Implementation

By following the convention of merge sort, we recursively split the array into k-sized segments by splitting the array in half and continued until each segment is of size k or less. These segments are then sorted using either insertionsort or binsertionsort. The sorted segments are then merged using the standard merging algorithm.

# 4 Test procedure

We decided to run the first tests several times and take the average values from the results of each k and array size. With a larger $k$ and array we ran the tests fewer times to reduce the total runtime.

The test we conducted were based a single array generated for each arraysize with the numbers in the array randomly generated from 0 to $10^6$. If we would have used a fixed array we would get the same results each run. We wanted to test different arrays and compare the average to get as close real-life results as possible.

# 5 Result

We desided to do fore tests one with a array of size 1000 wher the $k$ values ($k$ is a pisetiv integer) is $1-1000$ with the interval 1 (graf 5.2). The second test were with a array size $50,000$ and the $k$ value were $100-50,000$ with the interval 1000 (graf 5.3). The therd thest whare with a array size $200,000$ and the $k$ values were $1-200,000$ with logaritmic intervals (graf 5.1). the final test

where with a array size of $2,000,000$ and the $k$ value where betven $1-500,000$ with a logaritmic intervals (graf 5.4)

Ass we can see in graf 5.3 is the time incresing in steps. the time rises is hapening in logaritmig intervals. Bouth insertionsort and binsertionsort increses thime in the same plases but how mutch thay arr rising differs.

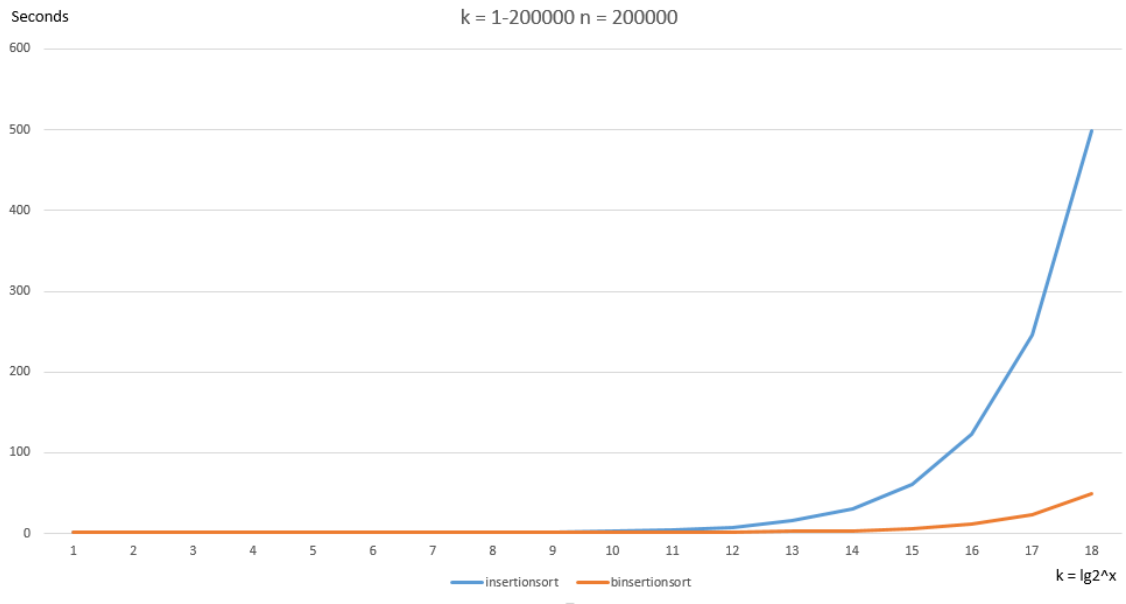


Figure 5.1: A graf of a test whith a array size of $200,000$ and the $k$ walue warius betwem 1 and $200,000$ wher the time metserd for evry logaritmic increse of $k$.

In graf 5.1 wher $k$ is incresing logarity can we se that the time is incresing exponentiol wen $k$ grows big.

Bouth algoritms is incresing wen $k$ passes a outer power of 2. This id done becos of the merge thet first splits the arrays. Wen $k$ increse ower a power of 2 the mearge nedade to split the arrat oune mor time withe mend it get tow arrays of the size of the prevues. Obesly tow arrays take the double time of one.

Sense it is the algoritms insertionsort and binsertionsort that take time is the time barly incresing wen the array size increse and the $k$ value is low. When the $k$ value gets higer the array that reatche the algoritms (insertionsort and binsertionsort)is biger ant it tace time for the algoritms to sovel the problem.

## 5.1 Insertionsort

From the grafes is it clear that insertionsort gets rely bad wen deling wid huge data sets. From graf 5.1 and graf 5.4 is it clear that the time get redicules hige.

A intresting thing that can be sen clearly in graf 5.3 thet the time dubels in logaritmi intervals. This geiv as the nice star loging graf.

## 5.2 BINSERTIONSORT

Ass insertionsort seam binsertionsort to incrise i logaritmik intervals, but binsertionsort do not increse as mutch. Sens binsertionsort splitt the array ín tow multible times is it lodgik tht it increses wen an pasing the line of a power of 2, becos it the neds to split the array in tow equal long arrays that prevues wer the howl array.

## 5.3 Graphs

### 5.3.1 Test 1, K 1-1000

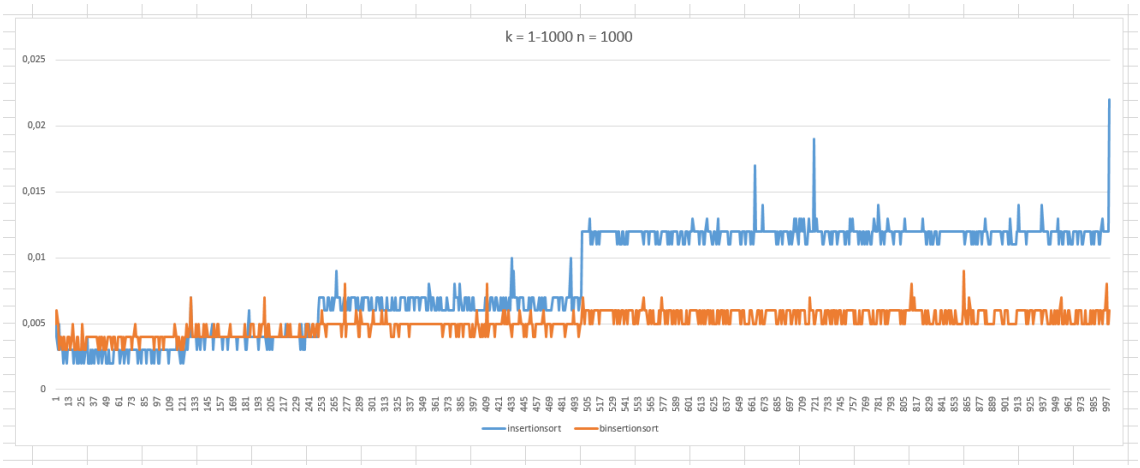


Figure 5.2: A graf of a test whith a array size of 50000 and the $k$ walue warius betwem 100 and 50000 wher the time metserd ouns evry 1000.

Here we see that small values are in some cases worse with binsertionsort. There is however a volume in witch binsertionsort switches as the better choise than regular insertionsort. Although the arraysize is only 1000, we see the small difference. We decided to run even bigger arrays with values from 1 to 2000000.

### 5.3.2 Test 2, K 100-50000

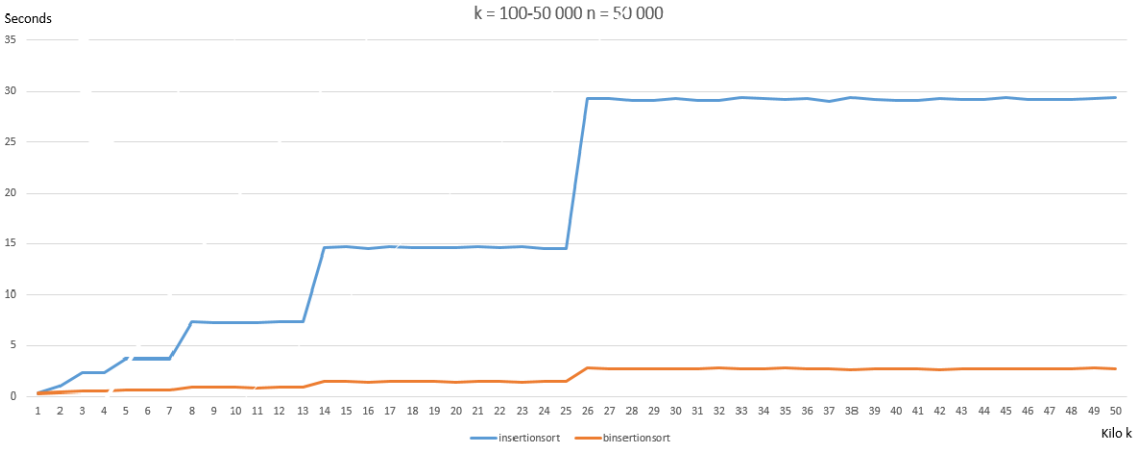


Figure 5.3: A graf of a test whith a array size of 50000 and the $k$ walue warius betwem 100 and 50000 wher the time metserd ouns evry 1000.

Here we can figure 5.3 ease see the increase of both algorithms and the difference of the two. while insertionsort quickly increases, binsertionsort doesn't increase nearly as much. This is when K=100-50000 and an array of size 50000.

### 5.3.3 Test 3, K 1-524288

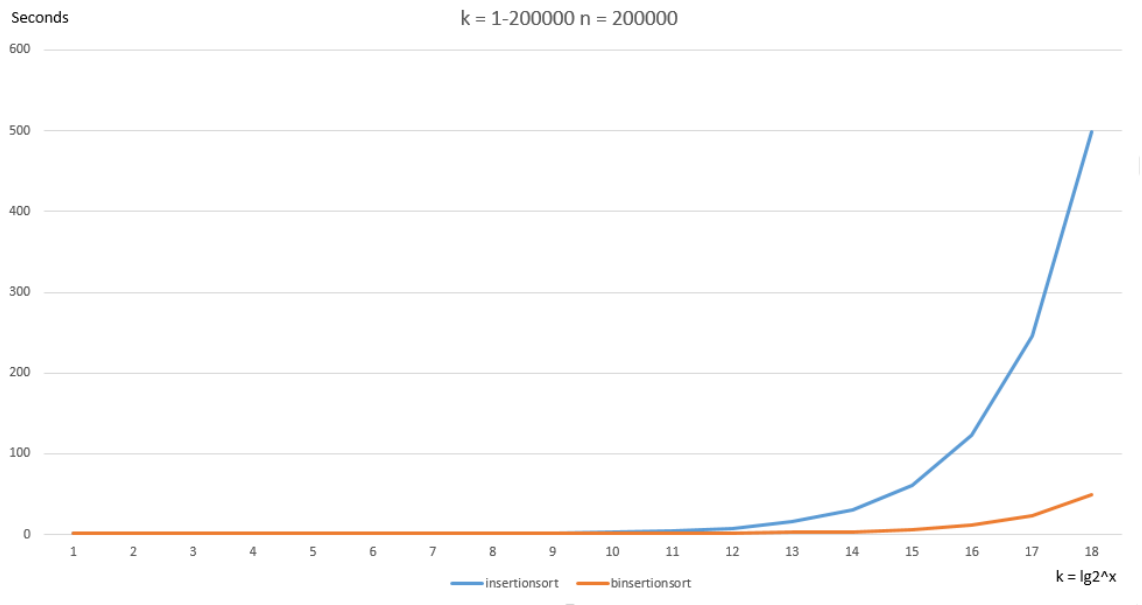


Figure 5.4: A graf of a test whith a array size of 200000 and the $k$ walue warius betwem 1 and 524288.

| K | insertionsort | binsertionsort |
|---|---|---|
| 1 | 15.2799999714 | 16.0329999924 |
| 2 | 13.4079999924 | 13.9700000286 |
| 4 | 12.1639997959 | 13.2150001526 |
| 8 | 11.5429999828 | 12.6840000153 |
| 16 | 10.5320000648 | 12.4670000076 |
| 32 | 10.8150000572 | 13.3180000782 |
| 64 | 11.7960000038 | 14.0509998798 |
| 128 | 14.1679999828 | 14.9419999123 |
| 256 | 19.7150001526 | 16.2039999962 |
| 512 | 30.8680000305 | 17.6510000229 |
| 1024 | 53.9000000954 | 19.7840001583 |
| 2048 | 100.339999914 | 23.3710000515 |
| 4096 | 194.943000078 | 30.0620000362 |
| 8192 | 384.326999903 | 44.7890000343 |
| 16384 | 752.871000051 | 79.0769999027 |
| 32768 | 1526.60899997 | 152.967000008 |
| 65536 | 3139.0769999 | 303.378000021 |
| 131072 | 6468.52200007 | 667.940000057 |
| 262144 | 13433.013 | 1536.90400004 |
| 524288 | 25904.4170001 | 5946.81500006 |

Table 5.1: K from 1 to 524288, Array size 2000000

The graph 5.4 and its values of table 5.1 is generated by starting with K=1 and array size=2000000. The values in the array are between 1 to 2000000. In this test we double the K value each step and watching the diffrences in boath algorithms. The results are ploted in graph 5.4. The table shows us that insertionsort doubles its time to sort by a factor of two while binsertionsort takes less time but also starts to (almost) double its values at a larger K value.

# 6 Discussion

The results we got were not surprising. We suspected in pre-test that insertionsort was worse than binsertionsort. Though we had not foreseen that there would be such clear steps in the graphs. (clear seen in graf 5.3). Although it was surprising, when you think about it it is fairly obvious that it should be that clear steps due to the algorithms. When we look at merge sort we can clearly see this. For example an array split into segments of 4 will take, roughly, half the time of an array split into segments of 8 due to the fact that bigger arrays take longer to sort. It was a bit surprising that binsertionsort increased runtime so slowly however.

Over implementation and chose to use random arrays may had inflicted the result so it did not end up in a clean,nice & exact graphs. The OS might have had an influences on the test result aswell due to other process running in the background. We have minimised the risk of

that by using a computer that had enough computer power to not be interrupted. We where able to find a trend that where similar to the function $\theta(nk + n\lg(\frac{n}{k}))$.

In the lab we learned that python is a really bad optimised language. When we talked to friends about their solution and their array sizes, they where much bigger. Yet their algorithms where faster aswell. We conclude that it was because of other language selections. The longest test we did took 7 hours and it had an array size of only $2,000,000$. If the algorithms is to be analysed in bigger datasets it would be necessary to rewrite them in a different language.