D0012E
Lab 1

Group 86
Henrik Eklund*
Alex Bergdahl†
Emil Magnusson‡

Teacher comments:

December 6, 2021

*henekl-0@student.ltu.se
†alxber-0@student.ltu.se
‡emimag-0@student.ltu.se

Contents

1 Introduction

Let *bSort* be a sorting algorithm that is a variant of *Insertionsort*, where the linear search technique (which is used for locating the position in which to insert the new item) is replaced by a binary search technique.Consider the following two modifications to Mergesort: First, the input is divided into $\frac{n}{k}$ sublists each of length k, where $k < n$ is a value to be determined and n is the length of the input list. Then each sublist is sorted using *insertionsort* and *bSort*, respectively. After that, the sorted sublists are merged using the standard merging mechanism. Your task is to implementand evaluate these two modified mergesorts.

2 Theory

2.1 bSort

Our binary merge sort works by splitting our lists into k many sub-lists and sorting each of them separately. When they are organised we merge them back together using b sort until we reach one list.

This new variant of merge sort is called binary merge sort or bsort. It works by splitting our list into k many sub-lists. As long as our k value is less than the amount of element in the original list.

Then we take each element in our sub-list and using binary search. We start with finding the middle point in its list, then check if our key value is less or greater than it. If it's greater, we do the same on the upper half of our list and if it's lesser then we do it on the lower half. Until we find the position the value should be in. We return this value and replace our new value in this and move the unsorted half up.

2.2 InsertionSort

Insertion sort is quite a simple algorithm that works well with fewer inputs in the array. The algorithm follows the following steps as can be seen in figure 1. Start by choosing the element in the second index, compare the value in the chosen index with the value of each lower index than the chosen index until a bigger value is found. If a value larger than the current element is found the new index for our element is the index before the bigger value. Repeat for the next bigger index until the start of the array is reached.

We observed that the algorithm is relatively quick in small amounts, but as the input gets larger it becomes progressively slower.

2.3 MergeSort

This algorithm works by splitting our list into two sub-lists. Then we will continue splitting our sub-lists by two until each sub-list reach one element. When the algorithm reaches this point, it then starts merging each element and sorting each sub-list two and two. As can be seen in figure 2. And as

each new sub-list is already sorted will the merge give one partially sorted list which also shortens the execution time.

This will be using the advantages the simplicity of insertion sort. The insertion sort will be skipping the startup instruction to split the code into smaller groups and instead directly checking each position and swapping. This will give the algorithm an decreased run time for smaller list due to the smaller instuction set.

3 Results

Comparing Insertion Sort to binary sort and merge sort, during a smaller unsorted list it has the advantage to have a simple logic. Which gives it a shorter execution time but have a huge decrease in performance for bigger lists while binary sort and merge sort is faster with larger lists itself.

During the simulation it was clear to see that if the list is short, less than 30 elements, insertion is more effective, while the larger the list, the worse insertion gets. Which is logical because of how it is written and as it is explained in part 2. The more elements, the more data it has to process and that is where bSort and MergeSort is more effective.

We can see according to figure 3, the k value does have affect on all our algorithms. However, bSort gets much larger effect boosts the higher k value we have, as long as we do not overstep the value of n. Though as the graph shows, when k approach 800 it starts to flattens out and does not give much better results.

4 Summary

We noticed that our algorithms can vary in results depending on the size of the problem. As we can see in figure 4 some of our algorithms really small increase execution time while for others the result drastically changes.

Each of our algorithms have a usage for different problems. Our insertion sort has a short run time for small list while our Binary and Merge sort is more useful for larger lists.

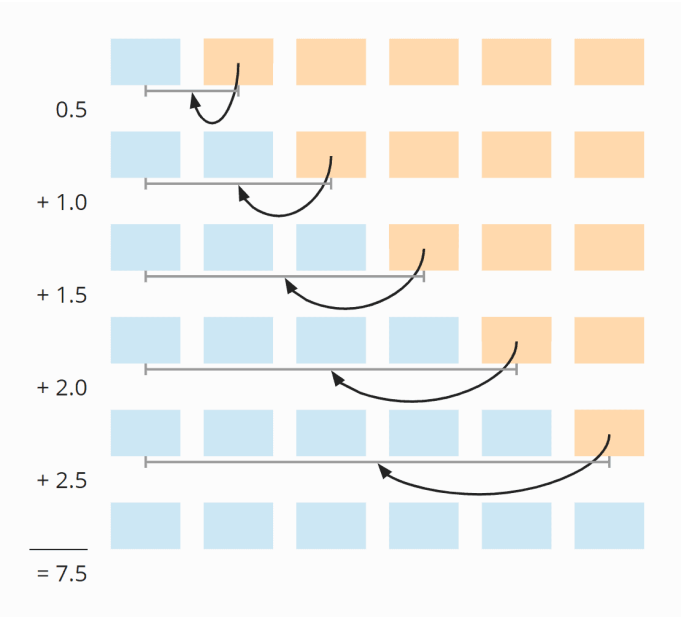

Figure 1: InsertionSort

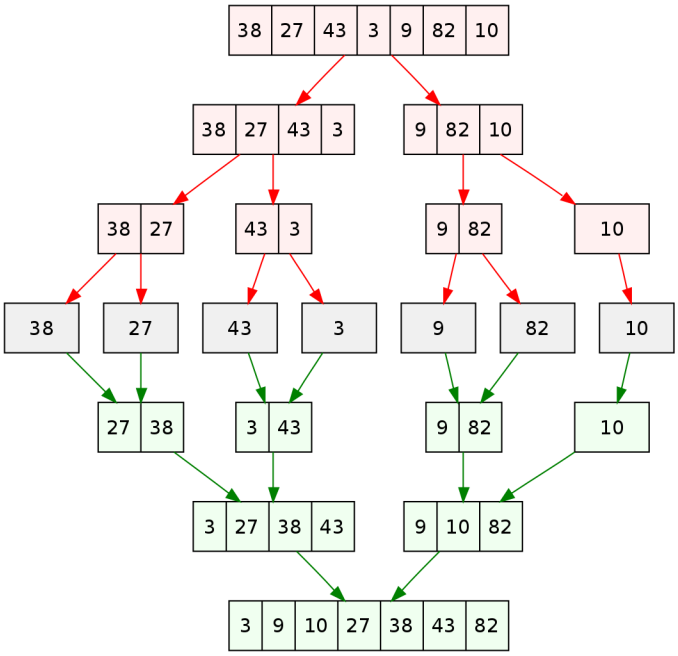

Figure 2: Merge sort

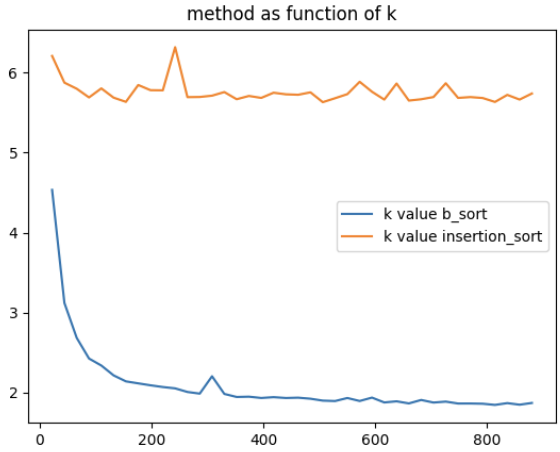

Figure 3: Finding the best k value for b sort and Insertion sort with 10000 item lists

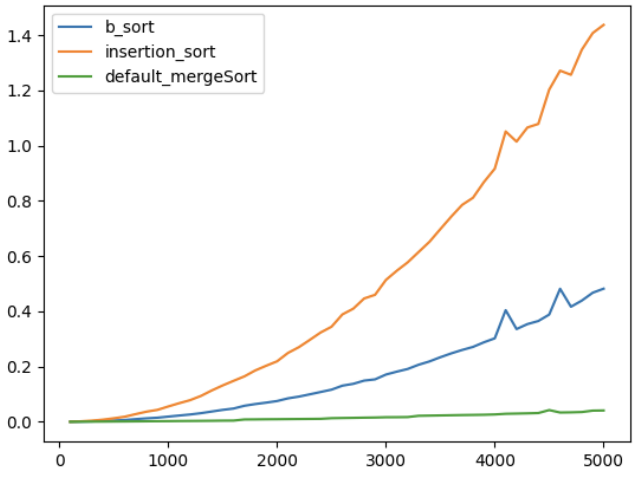

Figure 4: Insertion Sort, Merge Sort and BSort