Esercizio 1 Laboratorio Algoritmi e Strutture Dati
Merge-Binary Insertion Sort

912404 - Marco Edoardo Santimaria
926194 - Mia Zimonjic
912759 - Nicolò Vanzo

April 2021

1 Identification of the K value

Given that the complexity of Merge sort is $\mathcal{O}(n \log n)$ and given that the complexity of the Binary Insertion sort is still $\mathcal{O}(n^2)$, we expect that the value of K should be the one in which the cost of Binary Insertion Sort is less than Merge Sort. To do that we must solve the following dis-equation:

$$c_1x^2 \leq c_2n * log(n)$$

Which is equal to solve the following equation

$$c_1x^2 = c_2n * log(n)$$

and, whenever the interval of the array to be sorted is below the threshold identified by the equation, we utilize the Binary Insertion Sort instead of the Merge Sort. The equation has an infinity number of solutions, but all of them are with the following structure (in the real field):

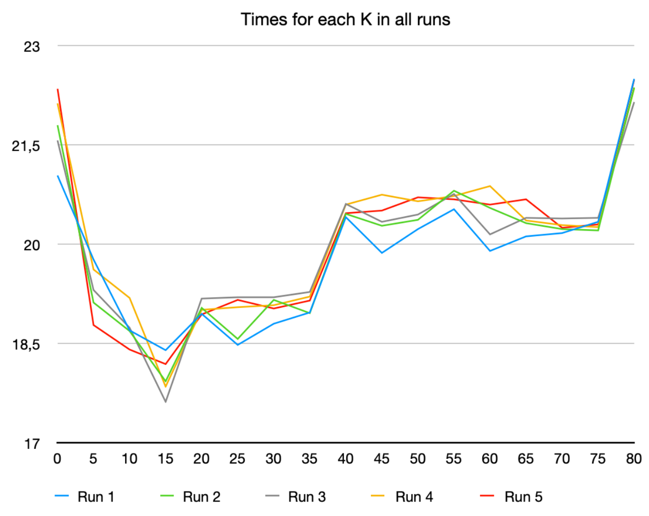
$$C_1 = C_2 \forall C_1, C_2 \in \mathbf{N}^+, n = 0^+$$

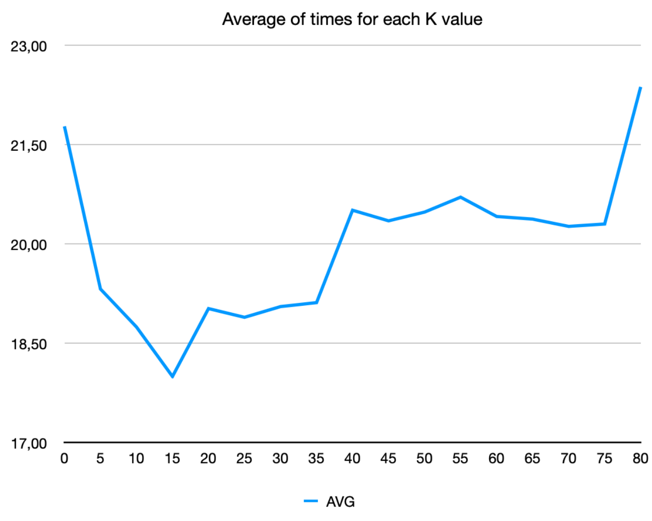
Where 0^+ is intended as the limit of n to 0.
This leads to our conclusion that the parameter K must be found trough testing as it is hardware dependant. Moreover, swapping and scheduling policies of the operating systems (as well as thermal throttling) might affect the results. To try to confine this behaviour, for each candidate of K value, we have decided to run 5 tests and kept into consideration the average of the timings.

1.1 K value for String comparison

The following values has been collected on an i5 8th gen CPU with 16Gb of ram.

K	run1	run2	run3	run4	run5	AVG
0	21,04	21,8	21,57	22,13	22,35	21,78
5	19,77	19,12	19,31	19,62	18,78	19,32
10	18,7	18,69	18,74	19,19	18,41	18,75
15	18,4	17,93	17,62	17,85	18,19	18,00
20	18,95	19,04	19,18	19,01	18,94	19,02
25	18,48	18,57	19,2	19,05	19,16	18,89
30	18,8	19,16	19,2	19,08	19,03	19,05
35	18,97	18,96	19,28	19,21	19,15	19,11
40	20,41	20,46	20,61	20,6	20,47	20,51
45	19,87	20,28	20,34	20,75	20,51	20,35
50	20,23	20,37	20,45	20,65	20,71	20,48
55	20,53	20,81	20,76	20,73	20,68	20,71
60	19,9	20,55	20,15	20,88	20,6	20,42
65	20,12	20,32	20,4	20,36	20,68	20,38
70	20,17	20,23	20,39	20,29	20,25	20,27
75	20,34	20,21	20,4	20,26	20,3	20,30
80	22,5	22,37	22,15	22,36	22,49	22,37

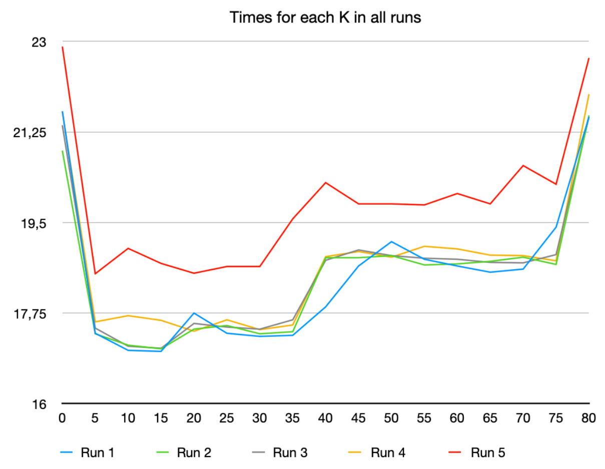


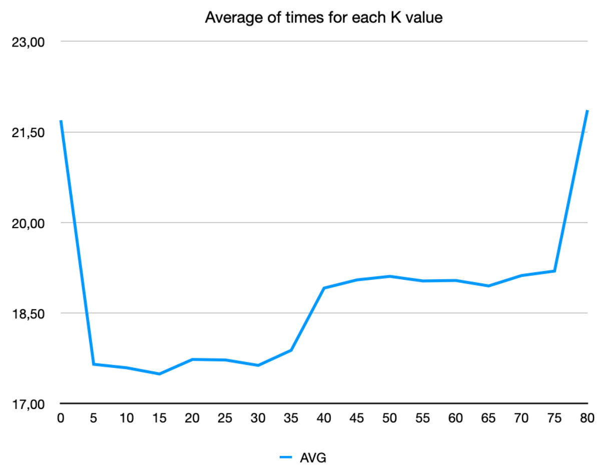


1.2 K value for Integer comparison

The following values has been collected on an i5 4th gen CPU with 4Gb of ram.

K	run1	run2	run3	run4	run5	AVG
0	21,65	20,89	21,38	21,65	22,90	21,69
5	17,36	17,35	17,46	17,58	18,51	17,65
10	17,03	17,13	17,11	17,70	19,00	17,59
15	17,01	17,06	17,07	17,61	18,71	17,49
20	17,75	17,44	17,55	17,40	18,52	17,73
25	17,36	17,51	17,48	17,62	18,65	17,72
30	17,30	17,35	17,44	17,43	18,65	17,63
35	17,32	17,39	17,62	17,52	19,57	17,88
40	18,87	18,82	18,77	18,84	20,27	18,91
45	18,66	18,82	18,97	18,94	19,86	19,05
50	19,13	18,86	18,86	18,83	19,86	19,11
55	18,79	18,68	18,81	19,04	19,84	19,03
60	18,66	18,70	18,79	18,99	20,06	19,04
65	18,54	18,75	18,73	18,87	19,86	19,95
70	18,60	18,83	18,72	18,86	20,60	19,12
75	19,41	18,69	18,88	18,76	20,24	19,20
80	21,54	21,57	21,54	21,98	22,68	21,86

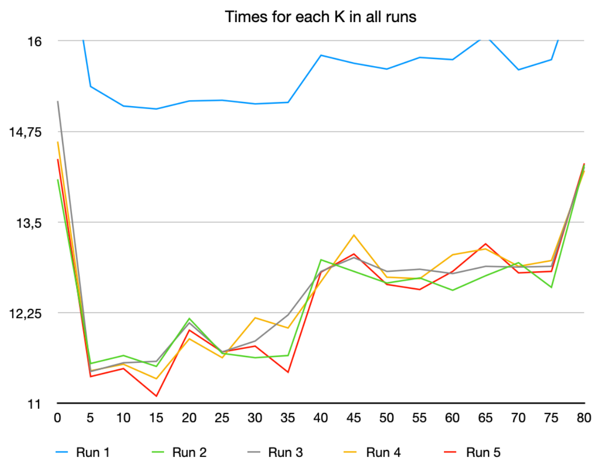


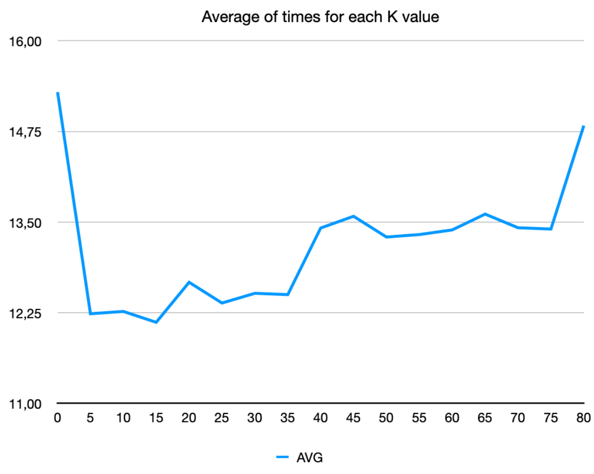


1.3 K value for Float comparison

The following values has been collected on an i5 8th gen CPU with 16Gb of ram.

K	run1	run2	run3	run4	run5	AVG
0	18,22	14,09	15,17	14,61	14,37	15,29
5	15,37	11,55	11,44	11,45	11,37	12,24
10	15,1	11,66	11,56	11,54	11,48	12,27
15	15,06	11,51	11,58	11,34	11,1	12,12
20	15,17	12,17	12,11	11,89	12,01	12,67
25	15,18	11,69	11,71	11,63	11,71	12,38
30	15,13	11,63	11,86	12,18	11,79	12,52
35	15,15	11,66	12,22	12,04	11,43	12,50
40	15,8	12,98	12,82	12,68	12,81	13,42
45	15,69	12,82	13,01	13,32	13,06	13,58
50	15,61	12,66	12,82	12,74	12,64	13,29
55	15,77	12,73	12,85	12,72	12,57	13,33
60	15,74	12,56	12,79	13,05	12,82	13,39
65	16,07	12,76	12,89	13,13	13,2	13,61
70	15,6	12,94	12,88	12,89	12,8	13,42
75	15,74	12,6	12,89	12,97	12,82	13,40
80	17,08	14,29	14,26	14,21	14,31	14,83





1.4 Are the results what we were expecting?

Yes, we were expecting that the curve to the times will be similar for all different CPUs. We were also expecting that when only Merge Sort is used times will be higher than when we use the binary merge insertion sort with the correct K. We were also expecting that as soon as the Binary insertion sort was going to be more prevalent than the Merge Sort, times are higher.

2 How to use the library

In order to use the library, users must define their own comparing function. This is done so that the library can be independent from the data structure or the sorting method. In the example Main.c file, we have defined 6 different comparison methods: one for the non decreasing order and the second for the non increasing method. Whenever a user choose to define it's own comparing method, he must bear in mind that the method MUST return an integer number, which shall be either 0 or one (or greater). The comparison function must have TWO parameters (the two items to be compared).
Once all the comparison functions have been created, The Sorting algorithm must be called in the following way:

```
MergeBinaryInsertionSort(<Data>, <Begin>, <End>,<CompareFunction> );
```

Where:

- <Data> is a pointer to void** to the custom data structure in which the sorting will happen;
- <Begin> is the first valid index to begin the sorting. usually it should be 0, but it can be changed to a non negative integer to sort only a portion of the array;
- <End> is the last valid index of your data structure to be sorted. usually it should be Data.length()-1 but it can be lower. <End> must be < than <Begin> otherwise, the program will not crash, but no sorting will happen.;
- <CompareFunction> is a pointer to the function that will do the comparison between the items to be sorted;

A final note on the crashes the library may cause. The library is meant to not crash the software in any way unless one of the following things happen:

- Incorrect coding of the comparison function by the user;
- The Library was not able to allocate memory for support data structure.

If some other kind of errors occurs, then the library would simply return, without sorting the data structure.

3 How to compile the demo project

In order to compile the demo program (the one that uses the given csv file) you must issue the following command:

```
make
```

Once the program is compiled it will automatically launch and it will automatically look for a file called ”records.csv”. If you want to specify the .csv file manually you can do it by first compiling the program with by issuing again ”make” and then launch the program manually (the main binary is located into the Build folder) in the following way:

```
/Build/BinarySortMainExec <path_to_csv_file>
```

This command (as well as the first one) will read the file, and sort it by Integer, Float and String and save the sorted array into three different csv files. Alternatively, you can issue the following command:

```
make verbose
```

This will compile the program defining the macro ”VERBOSE” which will cause the program to ask you at launch which field you want to sort and in what manner. You can also launch the program specifying the path to the csv file in the following way:

```
Build/BinarySortMainExecVerbose <path_to_csv_file>
```