A.A. 2020/2021

A SUMMARY OF ALL THE PROGRAMS/LIBRARIES IMPLEMENTED

A BRIEF REPORT

Andrea Cacioli

AA 2020/2021

SORTING ALGORITHMS

In the first assignment the request was to create an algorithm that could sort an array as quick as possible using the **merge sort** algorithm with a variation. The variation consisted of using an asymptotically slower algorithm called **insertion sort**. The aim of this variation was to, instead of decelerate, actually increase the speed of the algorithm, avoiding the need to create a new frame in the stack to sort a very small portion of the array, as merge sort is a recursive algorithm.

Roughly speaking, most of the work will be done by a faster algorithm and only when the limits of the machine come in a slightly slower algorithm is used. **The time it would take to prepare to execute the the faster is more than executing the slower one right away**.

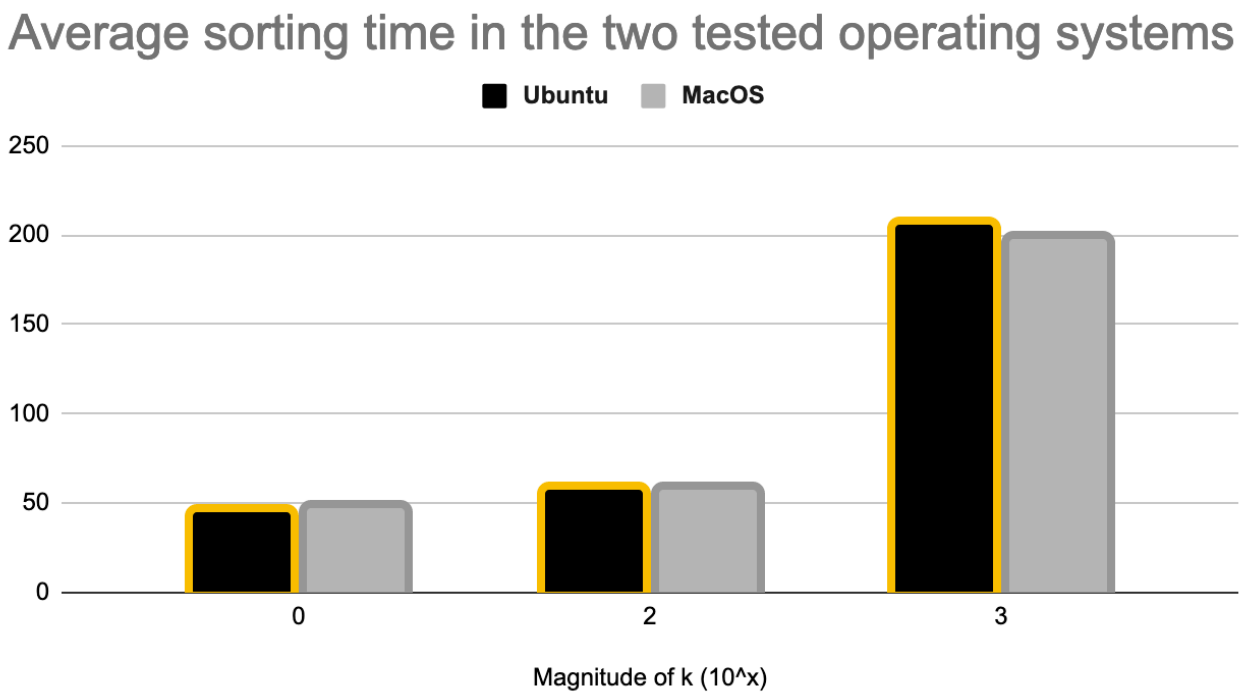
The second part of this assignment was to parameterise the length at which the change of algorithm is performed. The number of elements that the subarray needs to have for us to change the algorithm will from now on be called k.

The program was tested on a rather large sample size (20 000 000 records). The records had to be sorted using the three fields they came with: I string, an integer and a float. Since the implementation of the two algorithm combined needed to be

destructive of the initial array, instead of reloading from the file all the 20 million records, the algorithm was called on a different field keeping the records already sorted from a previous call. This, however, should not change the final results as ordering based on one field doesn't affect the order of the other field.

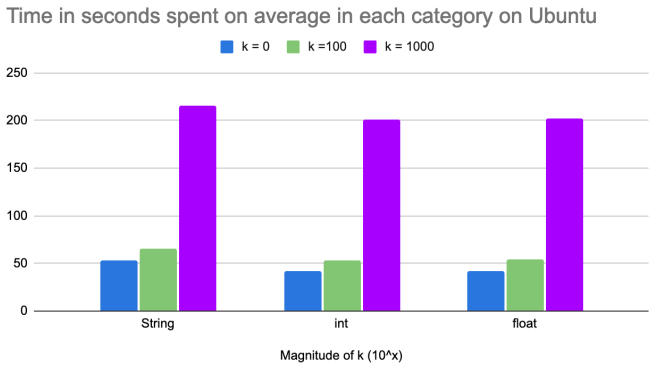
The average time spent by the algorithms is reported in the following graphs. Please keep in mind that these graphs come out of an **average of values** tested on different hardware, different operating systems and with different values of K: as reported in the graphs.

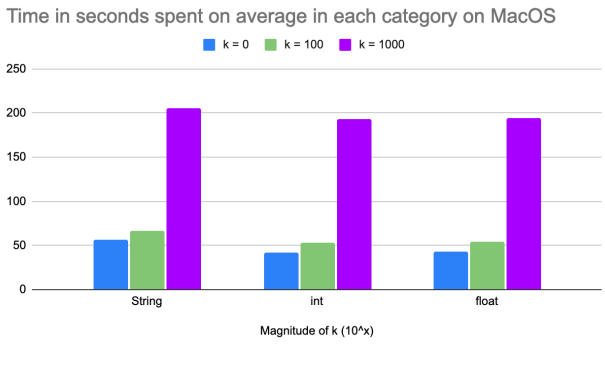
The Y axis represents time expressed in seconds.



As we can see the value of K affects very little when going from the order of 10 to the power of 0 to 10 to the power of 2.

The following two graphs represent the average time spent based on the different field regulating the order.





A wise choice of the value of K should be between 1 and 100 as, basically, after that value the time complexity of the insertion sort takes over the advantage of not creating a new frame in the stack.

EDIT DISTANCE OF STRINGS

The edit distance of two strings is an integer value that represents how many operations: **deletion, insertion and no-operation**, it would take for one string to be transformed into the second one.

To calculate this we have two different approaches: the first one is fully **recursive** and therefore rather time-consuming. The second one is an example of **dynamic programming**: it uses a matrix to store the previously calculated results in order to cut on some calculations. The size of the of the matrix is M x N where M and N are the sizes of the two strings.

The matrix is initialised with all elements being a negative one then every time a new value is calculated the position of that value will be found through the length of the strings and then stored in that location. The next time recursive call of the function refers to the 2 strings being of that length instead of re-calculating every sub problem we just have the value stored in the matrix.

In the example a dictionary containing a lot of words that are spelt correctly is used to check on every word of another file containing instead a quotation filled with grammar mistakes. The objective is to show that calculating the edit distance of the strings actually gives us good **suggestions** of what the writer meant when he made the mistakes.

GRAPHS

GENERAL IMPLEMENTATION

A graph is a data structure composed by **nodes and edges**. The node contains a key which could be of any type while the edges contain a tag that usually represents the weight or the distance between the nodes it connects.

A **sparse** graph is a graph where the amount of connections between each node is not close to the maximum amount of edges that, at any given time, could be in the graph. On the contrary a dance graph is one with a lot of connections. This difference is rather important as the implementation can be more efficient if we consider the graph to be mostly dense or mostly sparse. In a library we implemented the graphs to be optimal for sparse data: we used an arraylist of adjacent nodes for each individual node.

In order to make the graph as general as possible, the type of the key stored in the nodes and the type of the tag of each connection is **general** and is only limited by the fact that the object has to implement the Comparable<> interface, so that it is possible to have an order of magnitude of each edge and key.

Although every variable of the edge and node class is public, the **complexity is hidden** in the class of the graph which only allows the final user to call methods that operate on the public variables.

A graph can be either **oriented or unoriented**, which means that the direction of the connection between the nodes matters and limits the way we can access data only from one note to the other. One could see the unoriented version as one particular case of the oriented one in which every connection going from A to B is duplicated in the opposite direction from B to A.

ALGORITHMS IMPLEMENTED

The most common algorithms are implemented keeping in mind the time complexity that was decided beforehand and a list of these algorithms is shown as follows:

Please keep in mind that I as the grass is supposed to be sparse the complexity of reading from the adjacent nodes list is approximately O(1).

- Creation of an empty graph – O(1)
- Adding a node - O(1)
- Creating a new edge - O(1)
- Verifying if the graph is oriented or not - O(1)
- Verifying if the graph contains a given value - O(1)
- Verifying if the graph contains a connection between two given values - O(1)
- Deleting a node - O(n)
- Deleting an edge - O(1)
- Finding the number of nodes - O(1)
- Finding the number of edges - O(n)
- Getting all the values of the keys or all the values of the edges' tags - O(n)
- Finding the adjacent nodes of one node - O(1)

In addition to these methods are r**ecursive visit of the graph** to check if it has cycles has been added.

KRUSKAL'S ALGORITHMS

In an unoriented graph it is possible to define a minimum spanning tree or a minimum spanning forest that connects every note of the graph but doesn't have any cycle in it.

To do so the kruskal's algorithm uses a data structure called **union-find set.** The idea is to keep in the graph only the connections that do not create cycles. The algorithm is **greedy** so it uses the least "heavy" connections first and then it moves on to the heaviest ones. To detect a cycle we say that every node belongs to their own individual set which is then merged into another one when an edge connects the two nodes. If after this procedure another edge tries to connect to nodes that are already connected and thus in the same set, the algorithm deletes that connection as it would create a cycle and it is not necessary for the minimum spanning tree.

THE UNION-FIND SET DATA STRUCTURE

This data structure is used to store different types of data in **disjoint sets** that are identified by one particular elements called representative. The operations of union, finding the representative and making a new set are very time efficient.

Union by rank: when every union operation is performed the rank of the two sets is taken into account. To identify which one of the two sets is going to be the main set and which of the two representatives will be the representative of the final set a rank system is implemented. So if the two sets have the same rank one of the two is picked randomly as the winner and it **ranks up**.

Path Compression: every element of each set is associated with a pointer to its representative. When a union is performed, though, sometimes some elements point to other elements of the set instead of the representative which makes finding the representative a bit slower. So to avoid this problem every time we find the representative of one value we overwrite the pointer of the input to point to the representative. In this way the next time we tried to access that particular value we can directly find its representative.

THE DEMO

To test both the data structure and the algorithm a file containing data about Italian cities and the distance relative to each other is provided. After reading it all the information is stored into a graph on which then the algorithm is run and the difference in the total weight of the edges is reported and very significant.