It is understood that sorting plays an important task in most of the computer mechanisms. Most importantly, sorting algorithms can be used to solve other problems like counting duplicates, deciding rankings, finding medians in a dataset and many other problems. Commercially talking sorting can be used for tasks like event- driven simulations, searching for some specific information, numerical computations. Various sorting algorithms are in place to handle sorting tasks for a limited size of datasets.

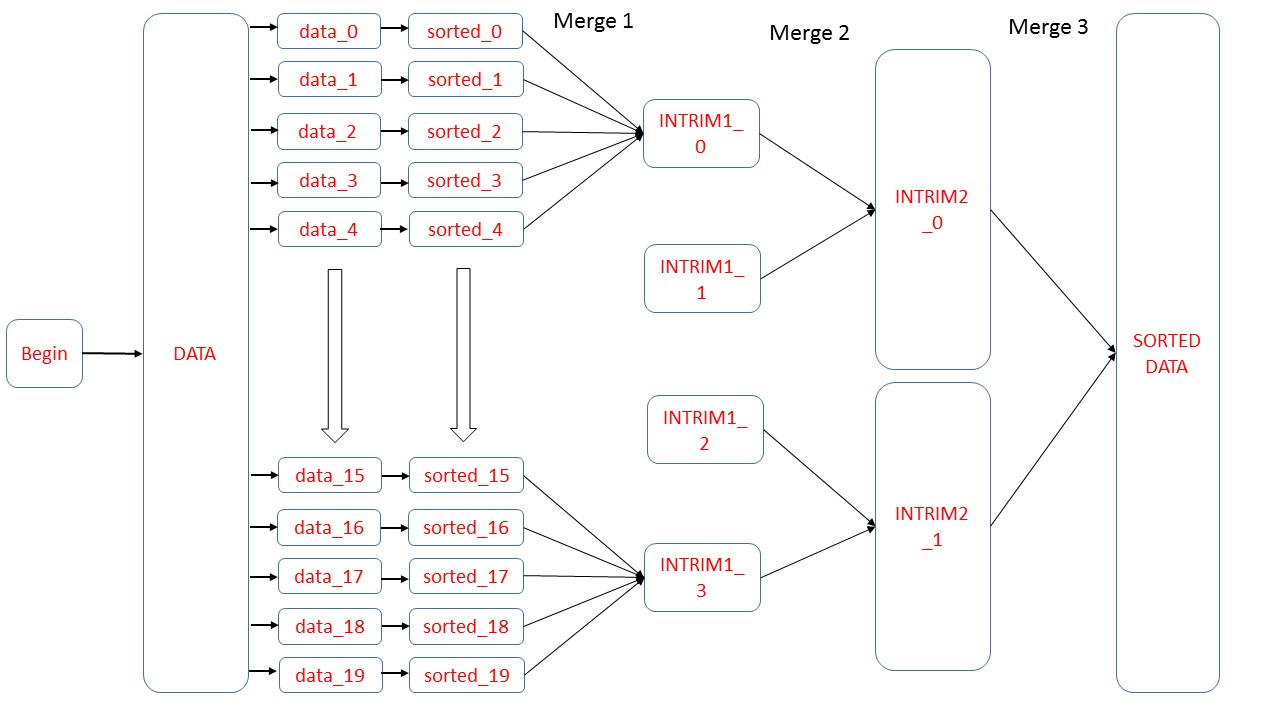
For datasets of huge size, the conventional sorting algorithms cannot work as most the times the values to be compared must be on the RAM at all the times. If the dataset is something like 100GB in size, RAM of that size will be expensive and unaffordable. Here external sorting algorithms come to the rescue.

External sorting algorithms can easily handle large datasets which cannot be sorted by conventional sorting algorithms. Basically, there are many flavors of external sorting algorithms since they are cooked up using a combination of conventional sorting algorithms.

The dataset which is too huge to handle is split up in various partitions of sizes which are manageable. These partitions are then sorted using conventional sorting algorithms like merge sort, heap sort, or quick sort and stored at a buffer. Finally, these buffers are merged again to create the same dataset from which they stem from but in sorted order.

The algorithm that we have implemented which is also a variation of external sort, divides the dataset into 20 equal parts. This is done for the sake of convenience because it is almost impossible to store a data this big in a specific program variable whether it is a string, an array, or a list. It is fine for smaller databases, but as the size of database increases the sorting algorithm can’t store these values in any structure. It may try to do that to a certain level, but once that level is reached the program stops responding.

The algorithm can be explained by fig.1



The 20 partitions created are stored in 20 separate datasets (data\_0, …, data\_19) and where they are not so large, hence at this level they can be sent to a sorting program. The dataset so created will be of size k/20, considering the original data to be of size k. The sorted data is stored in datasets of equal size (sorted\_0, …, sorted\_19). Once the sorting is done, we can combine the data to create the initial dataset but sorted.

The partitioning phase can take O(log(n)) time. Although we are making 20 partitions here, the base for the logarithmic function doesn’t have to change. The second phase which is sorting can take time depending upon the sorting algorithm that is used. e.g. insertion sort can take O(n^2) time, whereas merge sort can take O(n\*log(n)) time which is way faster that insertion sort.

We can do this merge using various methodologies but we have chosen a different approach here. Instead of performing a 20 – way merge, we performed a 5 – way 4 times, which gives us 4 intermediate datasets. These intermediate datasets which are just buffers are named INTRIM1\_0, …, INTRIM1\_3. So, this is the first merging scenario which takes place. As the building blocks for this dataset are of size k/20 and we are using 5 of the partitions to create an intermediate buffer dataset, its size will be k\*5/20 = k/4.

So, at this stage we have 4 sorted datasets pf size k/4 each.

Once the first level of merging is done, the 4 buffer datasets are merged 2 at a time to provide us with 2 buffers which are named INTRIM2\_0 and INTRIM2\_1. This two-way merge can be performed in O(n) time and the buffers so created will be of size k\*2/4 = k/2. Basically, these two buffers have half the size of original dataset. The final merge is performed on these two to give the final output as SORTED which can be performed in O(n) time and the final sorted dataset will have size k\*2/2 = k. That is the size of our original dataset.

Implementation

1. input data generation and storage –

the Python module that is used for this task is creating\_data.py.

the data entries are generated as a random variable within range 0 to sys.maxsize -1, where sys.maxsize stands for maximum integer value in Python which is dependent on the system you are working on. function random.randint() is used for that specific task.

The data so generated is then stored in a file named data.csv. the data is stored in a column of this file. For 100000000 values of integers, the file so created is around 1.25 GB (1,348,273,809 bytes).

1. partition phase –

in this phase file data.csv is split up into 20 equal partitions using Python module named divide.py. the partitions so created are stored in csv files named data\_0, data\_1, …, data\_19 respectively.

1. sorting phase –

in this phase the partitions created in the previous phase are sorted using merge sort and are stored in csv files as sorted\_0, sorted\_1, …, sorted\_19.

The python module used for this task is named sorting.py.

1. merging phases –

this phase consists of three levels in which different merge policies are applied to perform a 5 way merge 4 times, a 2 way merge 2 times, and a 2 way merge once. The Python module used for these tasks is named merging.py. intermediate results of these merge operations are stored in buffers named INTRIM1\_0, INTRIM1\_1, INTRIM1\_2, INTRIM1\_3, INTRIM2\_0, INTRIM2\_1 and are stored as csv files as well.

At the end of the merging operation, we get the final output dataset named as final\_sorted\_data.csv which has all the entries same as that of initial dataset but in a sorted order.