SAPIENZA
UNIVERSITÀ DI ROMA

Master of Science of Engineering in Computer Science
A.Y. 2021/2022

Data Management

Design and implementation of a software for visualizing the external multipass sorting algorithm

Submitted to:

Maurizio Lenzerini
Riccardo Valentini

Presented By:

Andrea Morelli
Lorenzo Romagnoli

Summary

External sorting

External sorting is a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory, usually a disk drive. Thus, external sorting algorithms are external memory algorithms and thus applicable in the external memory model of computation. External sorting algorithms generally fall into two types, distribution sorting, which resembles quicksort, and external merge sort, which resembles merge sort. The latter typically uses a hybrid sort-merge strategy.

Model and running time

External memory algorithms are analyzed in an idealized model of computation called the external memory model (or I/O model, or disk access model). The external memory model is an abstract machine similar to the RAM machine model, but with a cache in addition to main memory. The model captures the fact that read and write operations are much faster in a cache than in main memory, and that reading long contiguous blocks is faster than reading randomly using a disk read-and-write head.

The running time of an algorithm in the external memory model is defined by the number of reads and writes to memory required. The model was introduced by Alok Aggarwal and Jeffrey Vitter in 1988. The external memory model is related to the cache-oblivious model, but algorithms in the external memory model may know both the block size and the cache size. For this reason, the model is sometimes referred to as the cache-aware model.

The model consists of a processor with an internal memory or cache of size M, connected to an unbounded external memory. Both the internal and external memory are divided into blocks of size B. One input/output or memory transfer operation consists of moving a block of B contiguous elements from external to internal memory, and the running time of an algorithm is determined by the number of these input/output operations.

K-way merge-sort algorithm

Merge-sort algorithm for external memory uses the sort and merge strategy to sort the huge data file on external memory. It sorts chunks that fit in main memory and then merges the sorted chunks into a single larger file. Thus it can be divided into 2 phases – Run formation Phase (Sorting chunks) and Merging Phase.

In the following sections we will refer to the following Parameters:

- B – Number of pages of the relation R
- F – Number of buffer frames available
- S – Number of runs created at run formation phase, so ($S = B/F$)
- Z – Number of buffer frames used in the merging phase (usually $Z = F-1$)

The idea behind the algorithm

Assuming that we have F frames in the buffer the basic idea behind the algorithm is the following:

- Pass 0: read the file in runs of F pages, and, at each run, internally sort the F pages, and write a file (called run) to disk
  - In all subsequent passes, we reserve *F-1* input frames for input, and 1 frame for output

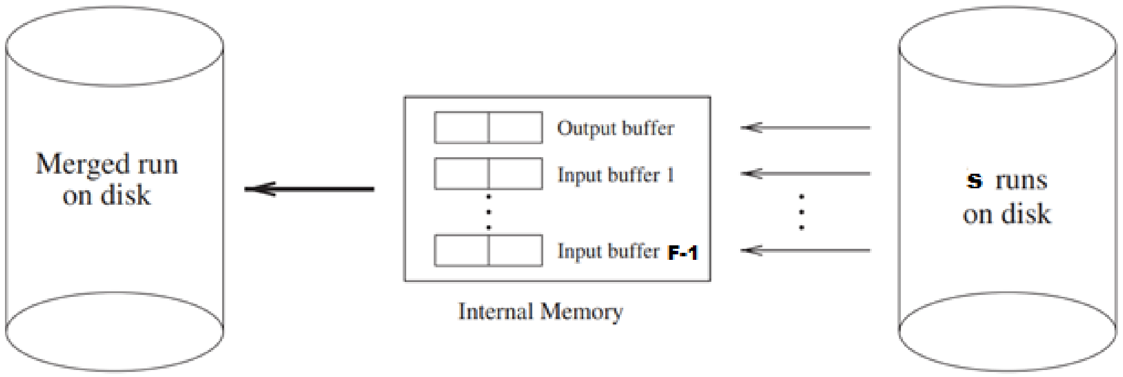
- Pass 1: merge *F-1* runs into one output runs
  - For each input run, read one page in one dedicated frame
  - When a page is used up, read the next page of the input run into the same frame
  - The result of merge is buffered in the output frame: when such frame is full, its content is written in the output run in secondary storage

- Pass n: for all the next passes we compute the same actions as the Pass 1 until we have a single sorted run that is the resulting one.
- Sorted!

Run formation phase

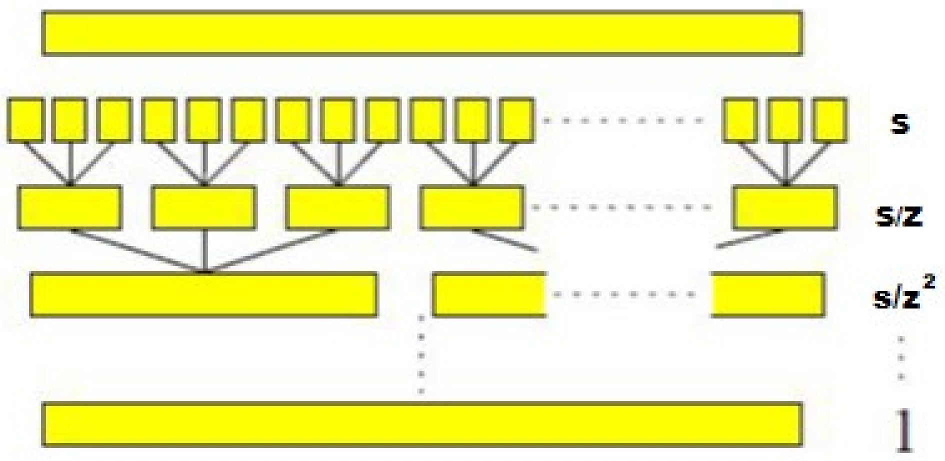
B pages of data are scanned, one memory load at a time. Every time we load F pages in the buffers, and we sort the elements in those pages into a single run, that is written in secondary storage. At the end of this run formation phase of the algorithm (also known as Pass 0), there are S number of sorted runs, given $S= B/F$. Those S runs will be then given as input to the merging phase of the algorithm.
The run formation phase which involves creation S sorted lists takes place in $O(2 * B)$ I/O operations.

Merging phase



Merge Phase

After initial runs are formed, the merging phase begins where groups of Z runs are merged. For each merge, the R runs are scanned and merged in an online manner as they stream through the internal memory.
The number of passes necessary to reach the end of the algorithm is equal to $\boldsymbol{K = log_Z S}$: then the number of runs created in each pass is S/Z^K and we can see visually how the formula is derived from the following picture.



Number of runs created at each pass of the merging phase

During the merging phase the S runs are merged together repeatedly.
As seen in the I/O complexity diagram, it forms a recursion tree with S elements at leaves and height of the tree equal to $\boldsymbol{K = log_Z S}$. The algorithm stops when we reach just one final sorted list at the end , therefore $1 = S/Z^K$ and it follows that $\boldsymbol{K = log_Z S}$.

Overall complexity

At every pass including pass 0 we read and write every page of the relation therefore the cost in terms of page accesses is equal to $\boldsymbol{2 * B * (log_Z S + 1)}$ that is equal to $\boldsymbol{2 * B * \left(log_Z \frac{B}{F} + 1\right)}$

Then since usually $Z = F - 1$ then we have: $\boldsymbol{log_Z \frac{B}{F} = log_{F-1} B - log_{F-1} F}$ then since F is a constant this doesn't depend on the data and we can remove this constant term from the cost.
So in this way the cost in terms of page accesses can be approximated to: $\boldsymbol{2 * B * (log_{F-1} B)}$
That is $O(\boldsymbol{B * (log_{F-1} B)})$

Design and implementation of the program to visualize the algorithm

To implement the visualization of the algorithm we decided to develop a python script that shows all the steps performed by the algorithm, printing out all the related information directly in the command line.

In the first part of the script, we managed to ask to the user to insert all the “parameters” related to the algorithm. So, we ask to insert: the number of pages to be sorted, the number of buffer frames available to the algorithm, the number of elements that should be in each page, and the end of the range from which we create randomly the values to be inserted in the pages. These preliminary steps are coded in the python code below.

```
#insert nOfPages
nOfPages=int(input("insert nOfPages: "))
#insert nOfBufferFrames
nOfBufferFrames=int(input("insert nOfBufferFrames: "))
#insert nOfElementsInAPage
nOfElementsInAPage=int(input("insert nOfElementsInAPage: "))
#insert rangeOfValues
rangeOfValues=int(input("insert rangeOfValues: "))

print("number of pages: "+str(nOfPages))
print("number of buffer frames: "+str(nOfBufferFrames))
print("number of elements in a page: "+str(nOfElementsInAPage))
print("range of values: "+str(rangeOfValues))
print()
```

Then for visualization and presentation purposes we created a class that extends the class list of python on order to override the method __str__ in order to, given the number of elements in each page defined at the start of the script, divide the various elements in the list (that contains all the values to sort) between each page. With that is more clear to see how the elements are divided between the various pages on which the algorithm operates onto since are divided with a “ | “ when the user look at them in the command line.

```
class myList(list):
    def __str__(self):
        lista=list(self)
        r="["
        for i,l in enumerate(lista):
            if(i%(nOfElementsInAPage)==0 and i!=0):
                r+="| "
            r+=str(l)+", "
        if len(r)>1:
            r=r[0:-2]
        r+="]"
        return r
```

After those initial steps, the script creates randomly the values of the number to be used in the execution of the algorithm given the maximum range defined initially by the user and appends them in the list “listOfNumbers”. It also initializes the list “runs” that will contains the various runs produced at the various steps of the algorithm so that will be utilized in the next parts of the code.

```
runs=[myList([])]
listOfNumbers=myList([])
for i in range(0,nOfPages*nOfElementsInAPage):
    listOfNumbers.append(random.randint(0,rangeOfValues))
print("this is the initial list: "+str(listOfNumbers))
```

After those steps, we have the part of the script that is in charge of actually sorting the elements created at the beginning performing so the pass 0 of the algorithm. The “for” loop performs a number of steps equal to the number of buffer frames available multiplied by the number of elements in a page, since in the merge sort algorithm at each step, the algorithm, performs the sorting and the merge of M pages into 1 resulting run. Inside the loop we can then observe the temporary variable “l” that is used in order to keep the values of the M pages (or runs) to take into consideration at the actual step and perform on those the sorting. Once those values are sorted, the resulting run, so its values are written in the array “runs”. While performing those operations, in the CMD are showed the related information to the user also.

```
for i in range(0,len(listOfNumbers),nOfBufferFrames*nOfElementsInAPage):

    l=myList(listOfNumbers[i:i+nOfBufferFrames*nOfElementsInAPage])
    print("load in the buffer frames the  n "
+str(int(i/(nOfBufferFrames*nOfElementsInAPage)))+" part of the list")
    print(l)
    l.sort()

    input()
    print("i sort the "+str(int(i/(nOfBufferFrames*nOfElementsInAPage)))+" part of the
list in the buffer frames and i write it in the second storage")
    print(l)
    print("\n")

    input()
    runs[0].append(l)

print("those are the sorted lists written in the second storage: ")

for i,l in enumerate(runs[0]):
    print("list "+str(i)+": "+str(l))
```

After that initial pass in which the runs are created, we start to execute the successive passes of the algorithm. The pass number is indicated by the value of the variable “level”. Given the M buffer frames available, this time the algorithm will compute the run computing M-1 runs (or sublists), since one frame will be reserved for the output. In order to show how the sort and merge are computed by the algorithm we created the function “removemin” that together with the rest of the script, shows and removes the least value of the elements in the runs/pages on which the algorithm is currently working on, showing also the related output run that that is computed step by step. When there are no more lists on which to perform the sort and merge, the script goes to the next pass of the algorithm, until there is only one single run left, indicating that all the elements of the pages have been sorted and has been produced the final sorted run as output. Those computations are coded in piece of code below.

```
level=0
while True:
    runs.append(myList([]))
    conta=0
    for i in range(0,len(runs[level]),nOfBufferFrames-1):
        sublistsToSort=runs[level][i:i+nOfBufferFrames-1]
        newList=myList([])

        if len(sublistsToSort) == 1:
            print("there is one only list left to sort so we skip the sort and write it
directly in secondary storage")
            runs[level+1].append(sublistsToSort[0])
            continue

        print("we are in the "+str(level)+" pass of the algorithm and those are the
current lists to sort: \n"+str(sublistsToSort))
        while True:
            res=removeMin(sublistsToSort,newList)
            if res==None:
                conta+=1
                print("this is the produced sorted list n "+str(conta)+"
"+str(newList)+" that will be written in the second storage\033[J")
                break
        runs[level+1].append(newList)
    level+=1
    if len(runs[level])==1:
        break
```

In the next piece of code is showed the function “removemin” that is actually in charge of sorting the elements of the runs and so produce the sorted output run for each pass and step. It takes as parameters the values related to the runs to sort and the list which will be updated as the sorted resulting run of the pass. Each time the function removes the minimum elements between the runs and adds it to the output run. The function is called until there are no more values left in the runs so, they have been all computed and sorted; this is checked simply letting the function return the last valued “removed” from the runs, that when it will be equal to null it means that all the values has been processed.

```
def removeMin(toSort,newList):
    minimum=rangeOfValues+1
    argmin=-1
    input()
    print("\033[J",end="")
    print("those are the current lists to sort: ")

    for i,l in enumerate(toSort):
        print("list "+str(i)+": "+str(l))
    print("this is the newlist being produced: ")
    print(newList)

    goBack="\033[A"*(len(toSort)+4)+"\033[F"
    print(goBack)

    for i in range(len(toSort)):
        if len(toSort[i])==0:
            continue
        if toSort[i][0]<minimum:
            minimum=toSort[i][0]
            argmin=i

    if argmin==-1:
        return None

    r=toSort[argmin].pop(0)
    newList.append(r)

    return r
```

In the code, we decided to insert a command that is executed in order to clear the cmd console at each iteration of the code, during the “removemin” function, so that when are shown the single elements that are being sorted and placed in the new output list, we see only the last iteration performed so that the command line interface will be more readable, without a lot of steps of execution shown. The following is the command required to clear the console in case of a Windows system.

```
os.system("cls")
```

In case instead the system on which the software is executed, is a Unix one, the previous command should be changed with the following one.

```
os.system("clear")
```

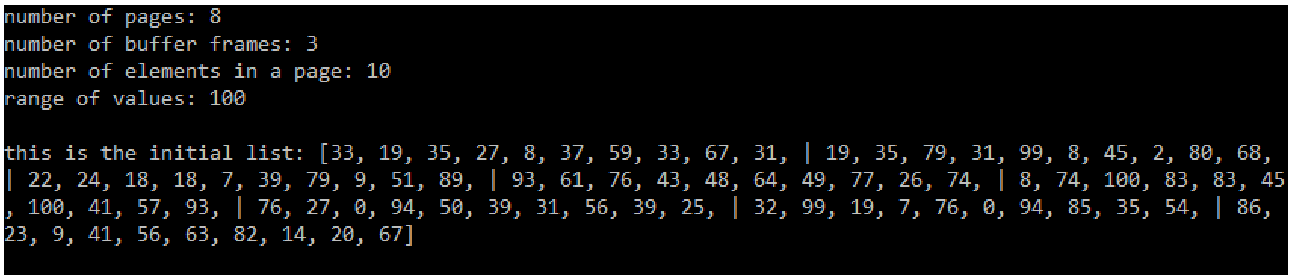
Moreover we also added the following line of code that simply deletes from the cmd everything that is after the cursor, again, to make the visualization better.

```
print("\033[J",end="")
```

Example of execution of the script

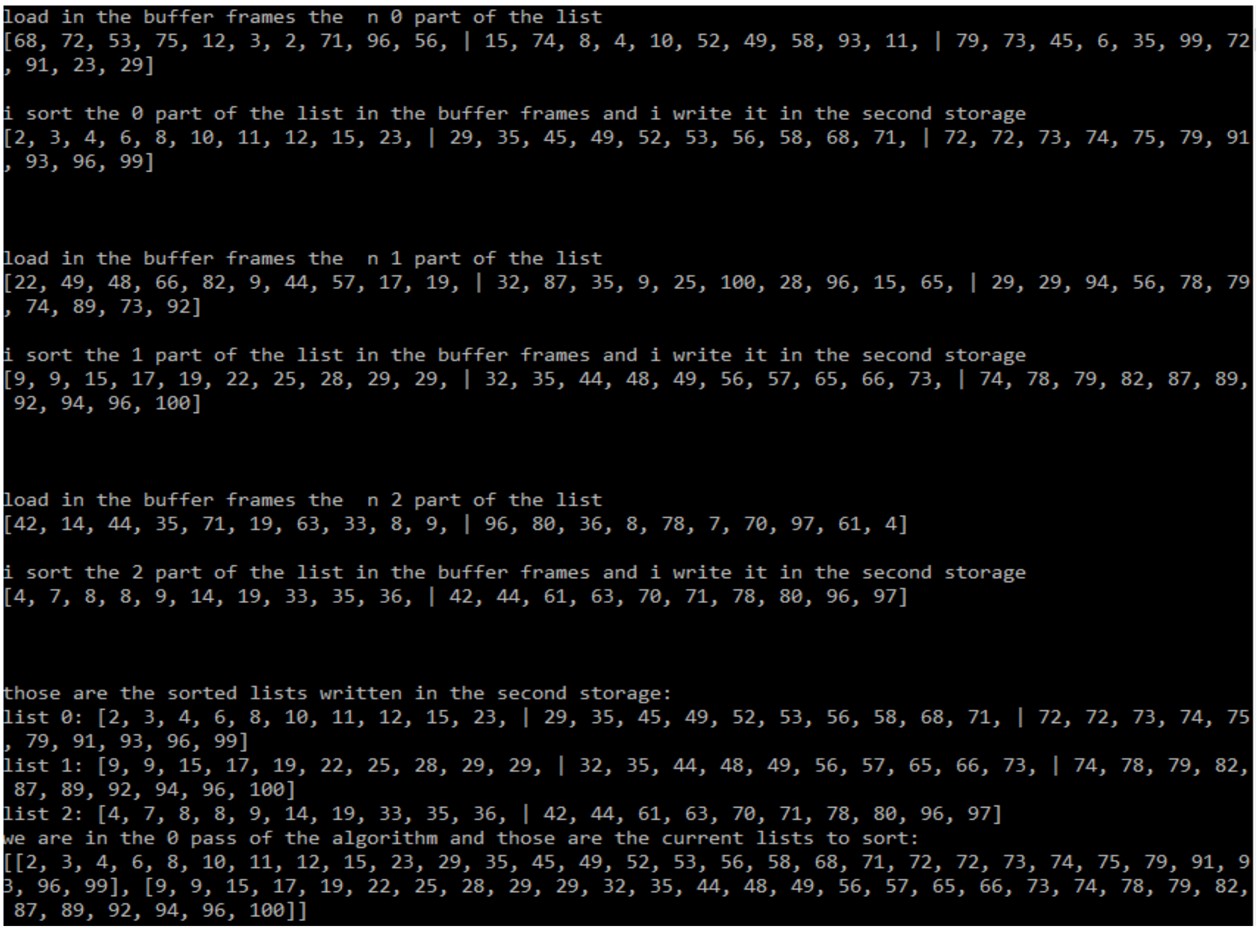
In this final part, we will show some screenshots related to the running script in order to show how the various steps and computations of the algorithm are presented to the user.

Firstly, the script asks the user about the number of pages, the number of buffer frames, and the number of elements per page to use in the algorithm. Also, the max number of the range from which the values are created randomly is asked in input to the user. After that in the cmd those values inserted by the user are confirmed and prompted out before the execution of the algorithm together with the various values created and divided in the various pages at random as shown in the screenshot below.



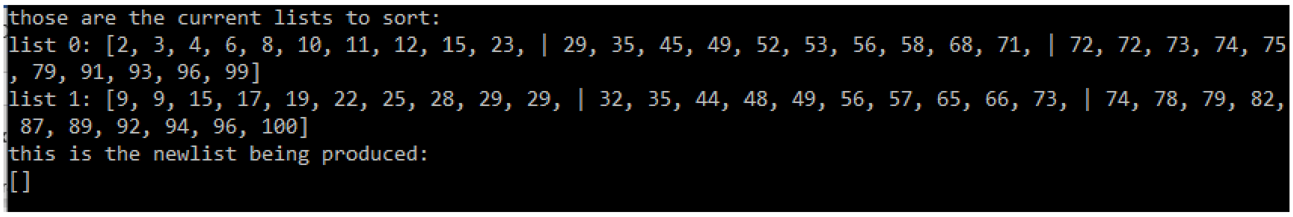
```
number of pages: 8
number of buffer frames: 3
number of elements in a page: 10
range of values: 100

this is the initial list: [33, 19, 35, 27, 8, 37, 59, 33, 67, 31, | 19, 35, 79, 31, 99, 8, 45, 2, 80, 68,
| 22, 24, 18, 18, 7, 39, 79, 9, 51, 89, | 93, 61, 76, 43, 48, 64, 49, 77, 26, 74, | 8, 74, 100, 83, 83, 45
, 100, 41, 57, 93, | 76, 27, 0, 94, 50, 39, 31, 56, 39, 25, | 32, 99, 19, 7, 76, 0, 94, 85, 35, 54, | 86,
23, 9, 41, 56, 63, 82, 14, 20, 67]
```

After that, the algorithm starts, showing the various pages taken into consideration for the first pass. After the computation is completed, the resulting run are also prompted out.



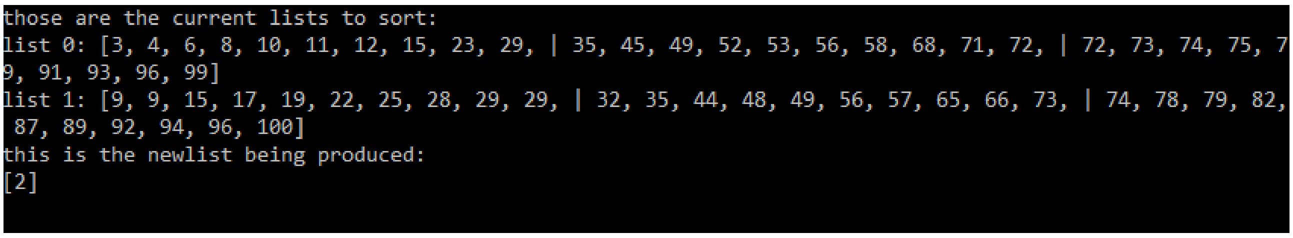
```
load in the buffer frames the  n 0 part of the list
[68, 72, 53, 75, 12, 3, 2, 71, 96, 56, | 15, 74, 8, 4, 10, 52, 49, 58, 93, 11, | 79, 73, 45, 6, 35, 99, 72
, 91, 23, 29]

i sort the 0 part of the list in the buffer frames and i write it in the second storage
[2, 3, 4, 6, 8, 10, 11, 12, 15, 23, | 29, 35, 45, 49, 52, 53, 56, 58, 68, 71, | 72, 72, 73, 74, 75, 79, 91
, 93, 96, 99]



load in the buffer frames the  n 1 part of the list
[22, 49, 48, 66, 82, 9, 44, 57, 17, 19, | 32, 87, 35, 9, 25, 100, 28, 96, 15, 65, | 29, 29, 94, 56, 78, 79
, 74, 89, 73, 92]

i sort the 1 part of the list in the buffer frames and i write it in the second storage
[9, 9, 15, 17, 19, 22, 25, 28, 29, 29, | 32, 35, 44, 48, 49, 56, 57, 65, 66, 73, | 74, 78, 79, 82, 87, 89,
 92, 94, 96, 100]



load in the buffer frames the  n 2 part of the list
[42, 14, 44, 35, 71, 19, 63, 33, 8, 9, | 96, 80, 36, 8, 78, 7, 70, 97, 61, 4]

i sort the 2 part of the list in the buffer frames and i write it in the second storage
[4, 7, 8, 8, 9, 14, 19, 33, 35, 36, | 42, 44, 61, 63, 70, 71, 78, 80, 96, 97]



those are the sorted lists written in the second storage:
list 0: [2, 3, 4, 6, 8, 10, 11, 12, 15, 23, | 29, 35, 45, 49, 52, 53, 56, 58, 68, 71, | 72, 72, 73, 74, 75
, 79, 91, 93, 96, 99]
list 1: [9, 9, 15, 17, 19, 22, 25, 28, 29, 29, | 32, 35, 44, 48, 49, 56, 57, 65, 66, 73, | 74, 78, 79, 82,
 87, 89, 92, 94, 96, 100]
list 2: [4, 7, 8, 8, 9, 14, 19, 33, 35, 36, | 42, 44, 61, 63, 70, 71, 78, 80, 96, 97]
we are in the 0 pass of the algorithm and those are the current lists to sort:
[[2, 3, 4, 6, 8, 10, 11, 12, 15, 23, 29, 35, 45, 49, 52, 53, 56, 58, 68, 71, 72, 72, 73, 74, 75, 79, 91, 9
3, 96, 99], [9, 9, 15, 17, 19, 22, 25, 28, 29, 29, 32, 35, 44, 48, 49, 56, 57, 65, 66, 73, 74, 78, 79, 82,
 87, 89, 92, 94, 96, 100]]
```

After this first steps, given the two runs on which the algorithm is being applied at the moment, the algorithm will show in the cmd, how the various values are taken from their pages and “inserted” in the resulting run. That resulting run will step by step contain all the sorted values from the initial runs taken into consideration. In the screenshots below we can see some of the steps that are performed in order to understand also how the various values are taken from the runs to “compose” the output sorted run.



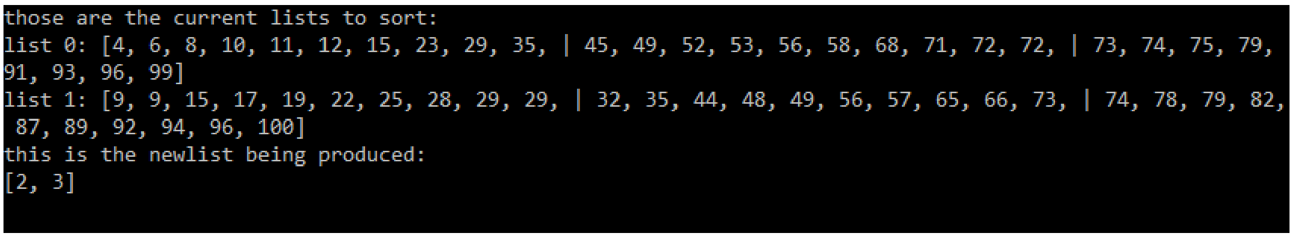
```
those are the current lists to sort:
list 0: [2, 3, 4, 6, 8, 10, 11, 12, 15, 23, | 29, 35, 45, 49, 52, 53, 56, 58, 68, 71, | 72, 72, 73, 74, 75
, 79, 91, 93, 96, 99]
list 1: [9, 9, 15, 17, 19, 22, 25, 28, 29, 29, | 32, 35, 44, 48, 49, 56, 57, 65, 66, 73, | 74, 78, 79, 82,
 87, 89, 92, 94, 96, 100]
this is the newlist being produced:
[]
```

In this step we can see the lists 0 and 1 that are being merged together in a single resulting run. As we can see the first merged value is “2”.

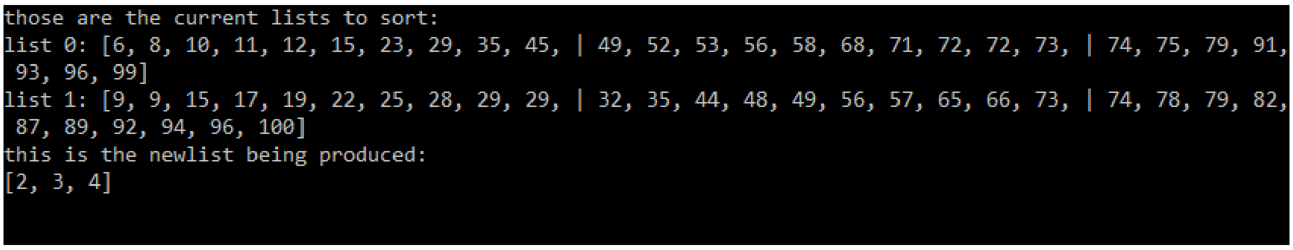


```
those are the current lists to sort:
list 0: [3, 4, 6, 8, 10, 11, 12, 15, 23, 29, | 35, 45, 49, 52, 53, 56, 58, 68, 71, 72, | 72, 73, 74, 75, 7
9, 91, 93, 96, 99]
list 1: [9, 9, 15, 17, 19, 22, 25, 28, 29, 29, | 32, 35, 44, 48, 49, 56, 57, 65, 66, 73, | 74, 78, 79, 82,
 87, 89, 92, 94, 96, 100]
this is the newlist being produced:
[2]
```

The same as before is shown in this step that adds “3” to the resulting list. And so on for the successive values in the lists.

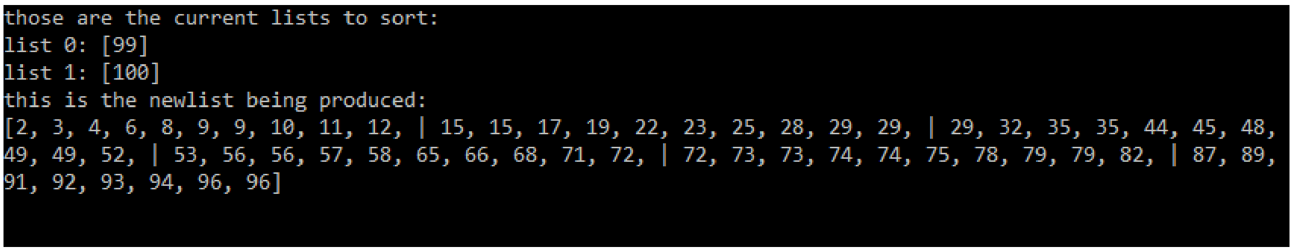


```
those are the current lists to sort:
list 0: [4, 6, 8, 10, 11, 12, 15, 23, 29, 35, | 45, 49, 52, 53, 56, 58, 68, 71, 72, 72, | 73, 74, 75, 79,
91, 93, 96, 99]
list 1: [9, 9, 15, 17, 19, 22, 25, 28, 29, 29, | 32, 35, 44, 48, 49, 56, 57, 65, 66, 73, | 74, 78, 79, 82,
 87, 89, 92, 94, 96, 100]
this is the newlist being produced:
[2, 3]
```

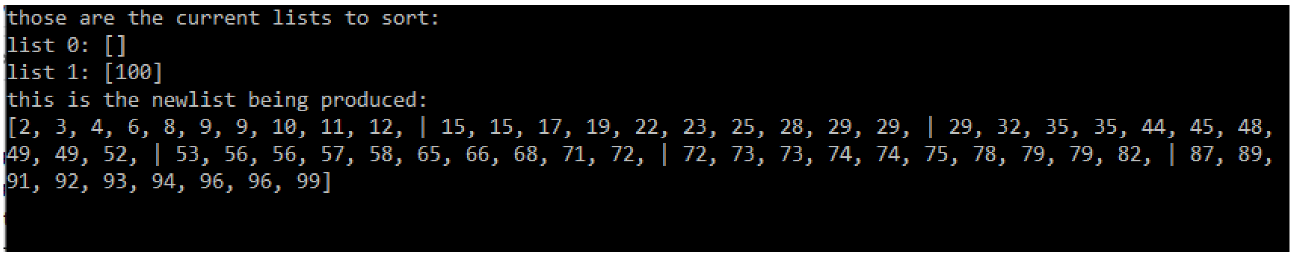


```
those are the current lists to sort:
list 0: [6, 8, 10, 11, 12, 15, 23, 29, 35, 45, | 49, 52, 53, 56, 58, 68, 71, 72, 72, 73, | 74, 75, 79, 91,
 93, 96, 99]
list 1: [9, 9, 15, 17, 19, 22, 25, 28, 29, 29, | 32, 35, 44, 48, 49, 56, 57, 65, 66, 73, | 74, 78, 79, 82,
 87, 89, 92, 94, 96, 100]
this is the newlist being produced:
[2, 3, 4]
```

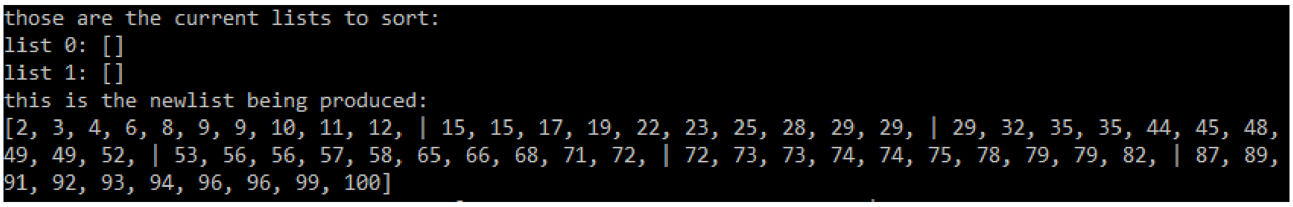
…..



```
those are the current lists to sort:
list 0: [99]
list 1: [100]
this is the newlist being produced:
[2, 3, 4, 6, 8, 9, 9, 10, 11, 12, | 15, 15, 17, 19, 22, 23, 25, 28, 29, 29, | 29, 32, 35, 35, 44, 45, 48,
49, 49, 52, | 53, 56, 56, 57, 58, 65, 66, 68, 71, 72, | 72, 73, 73, 74, 74, 75, 78, 79, 79, 82, | 87, 89,
91, 92, 93, 94, 96, 96]
```



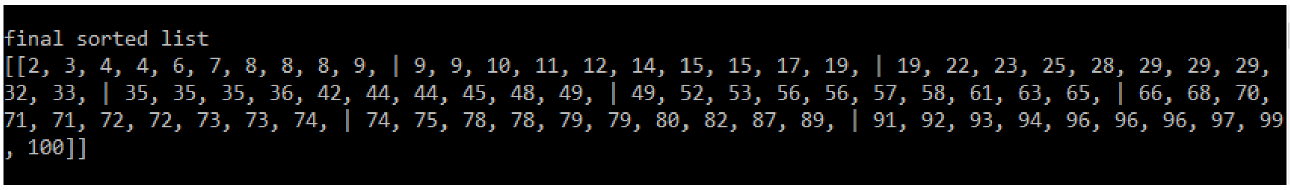
```
those are the current lists to sort:
list 0: []
list 1: [100]
this is the newlist being produced:
[2, 3, 4, 6, 8, 9, 9, 10, 11, 12, | 15, 15, 17, 19, 22, 23, 25, 28, 29, 29, | 29, 32, 35, 35, 44, 45, 48,
49, 49, 52, | 53, 56, 56, 57, 58, 65, 66, 68, 71, 72, | 72, 73, 73, 74, 74, 75, 78, 79, 79, 82, | 87, 89,
91, 92, 93, 94, 96, 96, 99]
```



```
those are the current lists to sort:
list 0: []
list 1: []
this is the newlist being produced:
[2, 3, 4, 6, 8, 9, 9, 10, 11, 12, | 15, 15, 17, 19, 22, 23, 25, 28, 29, 29, | 29, 32, 35, 35, 44, 45, 48,
49, 49, 52, | 53, 56, 56, 57, 58, 65, 66, 68, 71, 72, | 72, 73, 73, 74, 74, 75, 78, 79, 79, 82, | 87, 89,
91, 92, 93, 94, 96, 96, 99, 100]
```

The previous steps are performed until the two “input” lists are merged all together.

In the end, we will have a single, sorted resulting run, that will be printed out and the algorithm will be concluded.



```
final sorted list
[[2, 3, 4, 4, 6, 7, 8, 8, 8, 9, | 9, 9, 10, 11, 12, 14, 15, 15, 17, 19, | 19, 22, 23, 25, 28, 29, 29, 29,
32, 33, | 35, 35, 35, 36, 42, 44, 44, 45, 48, 49, | 49, 52, 53, 56, 56, 57, 58, 61, 63, 65, | 66, 68, 70,
71, 71, 72, 72, 73, 73, 74, | 74, 75, 78, 78, 79, 79, 80, 82, 87, 89, | 91, 92, 93, 94, 96, 96, 96, 97, 99
, 100]]
```

So this list will be the result of the execution of the algorithm that will be written in secondary storage.

How to execute the software

The software, is available in the github at the following link: https://github.com/bbooss97/externalmultipasssorting

Once downloaded the script “algo.py” it can be executed with a Windows cmd using python calling the command “python algo.py” in the same directory in which the script is saved in the computer.
Once inserted the number of pages, the number of buffer frames, and the number of elements per page to use in the algorithm and also, the max number of the range from which the values are created randomly. Then the cmd will show the various steps of execution of the algorithm and, in order to go on with the various steps and their visualization the user will just need to press the “Enter” key until the algorithm execution ends.