External Merge Sort Visualizer

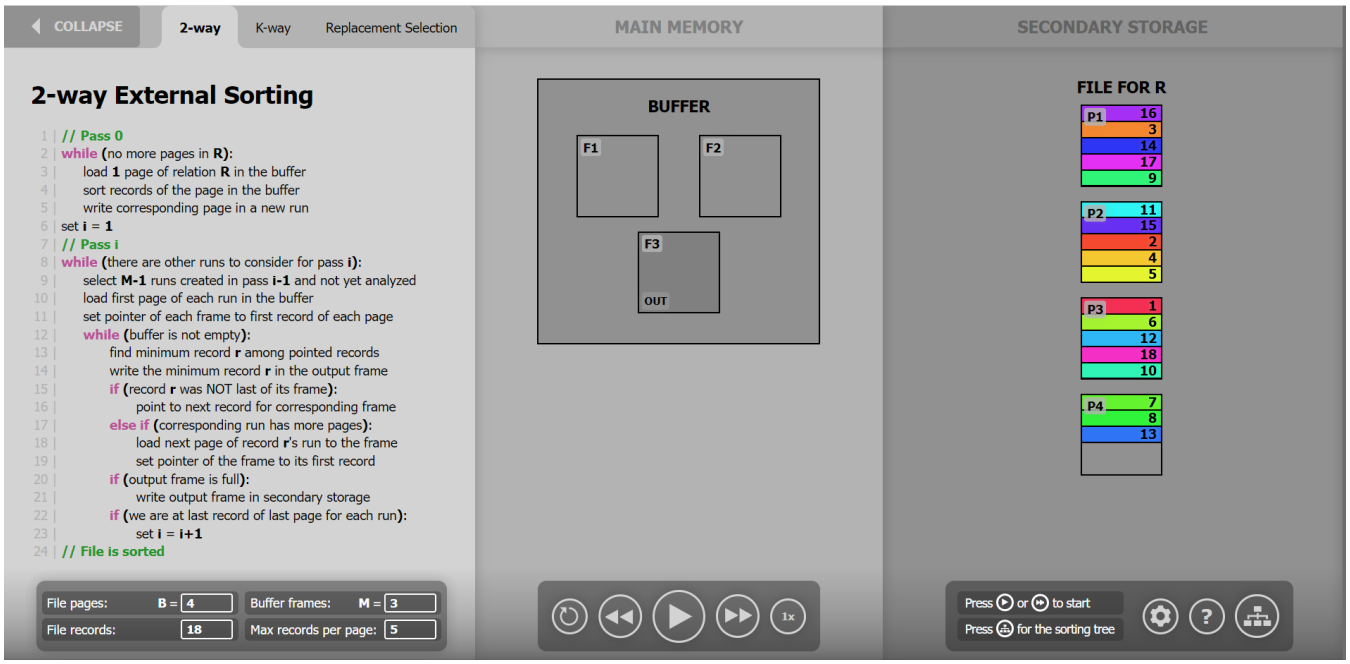
Data Management 2022/2023 - Project Report

1. Introduction **2**
- 1.1. Internal and External Sorting 2
- 1.2. Possible Approaches 2
- 1.3. External Merge Sort 4
  - 1.3.1. Two-Way External Merge Sort 4
  - 1.3.2. K-Way External Merge Sort 5
  - 1.3.3. Replacement Selection Strategy 5
  - 1.3.4. Other Optimizations 6

2. Algorithms Visualization **7**
- 2.1. Implemented System 9
- 2.2. System Features 10
- 2.3. Application Logic 11
  - 2.3.1. Animation Phases 11
  - 2.3.2. Sorting Tree 13
  - 2.3.3. The Undo Problem 14

3. Additional Notes and Conclusions **15**
- 3.1. Development Process 15
- 3.2. Conclusions 15

4. References **16**



The final developed system

1. Introduction

1.1. Internal and External Sorting

External sorting algorithms allow for sorting large amounts of data by only considering a small fraction of that data at a time.
They are used when the data being sorted do not fit into the main memory of a computing device (e.g. RAM), and instead must reside in the external memory (e.g. disk drive): this is known as “external memory model”.

In the "internal sorting" approach, used instead when considering arrays or small data structures in general, the entire data to be sorted is loaded in the main (internal) memory before proceeding.

When larger collections of data need to be sorted, however, this approach is not applicable: this is because algorithms following the "internal sorting" approach (like selection sort, merge sort, quick sort, etc.) can afford to consider and manage the entire data being sorted at each single iteration, and operate based on the order of all the elements to sort in that specific iteration.

Larger collections of data which can’t fit into the available memory of a computing device, and must therefore reside in the "external memory" (secondary storage), make the described approach inapplicable

1.2. Possible Approaches

When sorting records of a large relation contained in a file and stored in a database (in secondary storage), in order to be able to replicate the behavior of the "internal sorting" approach, and thus considering all the elements (records) at each iteration of a sorting algorithm (like in the case of merge sort, which needs to take into account specific elements in the merging phase), a possible approach would be to load each element from external (or secondary) memory, into the main memory, one at a time and each time the algorithm requires to analyze that specific element.

Replicating “internal sorting” algorithms in this way, however, would force the loading in main memory of the entire page of the file in which the single element (e.g. a record of a relation) is located, since a page represents the smallest collection of data that can be loaded from secondary storage (i.e. records can't be retrieved one at a time from the secondary storage).

The described approach is highly impractical, mainly because in the external memory model we are not interested in minimizing the amount of times the data is read (as for merge sort and the other internal sorting approaches), but instead in minimizing the number of page accesses of the sorting algorithm for data stored in the external memory (secondary storage), which act as a bottleneck for the time

complexity of the sorting algorithms (since the loading of a page from secondary storage is very slow).

A second possible approach to implement a sorting algorithm in the external memory model would be to work directly on the disk (i.e. on the external memory itself) and sort records without using the main memory, but this is, once again, very slow.

Accessing data from an external (secondary) memory is slow because of the absence of an index or a way to univocally retrieve single elements of the stored data (since external memory devices, other than offering higher “data retention” than primary storage devices, are intended to store large amounts of data, and indexing this large amount of data would generate a lot of overhead and oftentimes not be efficient).
This renders searching for a single element of a large collection of data directly from the secondary storage very slow (unless we use a “Direct-Access Storage Device” or DASD, as a secondary storage device, which allows for quick access of stored data through unique addresses assigned to each block of memory).

A third, final and practical approach consists of considering only a small portion of data at a time, at each step of sorting, namely as much data as can fit into the available main memory of the computing device.
This is known as “External Sorting”, because we can sort data stored in the external memory (secondary storage) without the need of loading the entire data collection in the internal memory (which is what happens with “internal sorting” approaches).

The described approach is the best solution in terms of number of accesses to the secondary storage, and therefore also in terms of time complexity if we consider access to secondary storage as a time bottleneck for sorting.

A possible implementation of external sorting is the so-called “external merge sort”. The main idea is to divide the data to be sorted into smaller chunks, sort them individually, and then merge them together in the correct order, creating larger and larger sorted chunks at each iteration, and eventually obtaining a single sorted collection of data, in a similar way to the internal merge sort algorithm (but with the previously mentioned differences and constraints).

Another possible implementation of external sorting is the “external distribution sort”, which consists of finding “pivot points”, in a similar way to the internal quick sort algorithm, and then use these “pivots” to recursively divide the data to be sorted into smaller and smaller chunks until we get to a size which can fit into a single memory page.
Once we are done partitioning the data, each single chunk can be loaded into the main memory and therefore efficiently be sorted.
Sorted chunks, taken in the order they were partitioned (and then sorted), make up the sorted collection of data of the initial collection.

1.3. External Merge Sort

The following is an introduction to external merge sorting algorithms (external distribution sorting approaches won't be discussed).

1.3.1. Two-Way External Merge Sort

The simplest implementation of external merge sort is the "two-way" external sorting algorithm.

We start from a relation R containing a certain amount of records and stored into B pages of a file in secondary storage (and therefore in B blocks of memory).
Usually, the number of pages B of the file will be way larger than the available number of blocks in main memory, and therefore, if we assume to have, in the main memory buffer, only M free blocks of memory (called "frames"), we will only be able to consider M pages of the file for relation R at a time (with the corresponding records contained in those pages).

In an initial "pass" of all the records of the file (that we will call "pass 0") we can consider a single page of the file at a time, load it into the main memory, sort its records using an efficient internal sorting algorithm (e.g. quick sort or merge sort), and then write the page back into secondary storage.
The B created pages are called "runs", i.e. sorted portions of the original file.

Once we are done with this initial pass, we can proceed by "merging " the sorted runs in a similar way to the "merge phase" of the internal merge sort algorithm, using another "pass" of the records stored in the runs (called "pass 1"): we can load the first pages of M-1 runs created in the previous pass into M-1 free frames of the buffer and merge their records into a single new run of M-1 pages.
We only consider M-1 frames because we need to reserve one of the M frames, called "output frame", for the generated output of the "merge", hence we need to keep a free frame of the buffer to store the records in order and to then be able to write its content back to the secondary storage.
Whenever the "output frame" is full, we write it into secondary storage and therefore empty it to free space for the next records of the pages we are merging.
Whenever we reach the final record of a page (of a certain run) currently in one of the buffer frames, we load the next page of the corresponding run into the same frame of the buffer, and continue the merging.

After this first pass, we can proceed in the same way for a second pass, and therefore consider a generic "pass i" after "pass 0" as the merge of runs created in "pass i-1", until we reach a point in which we only have a single run of sorted pages and records, which corresponds to the sorted records of relation R.

The cost of this kind of algorithm in terms of accessed secondary storage blocks of memory is $2 \times B \times N$ where N is the total number of passes (since at each pass we read once and write back once each of the B pages of the file storing the relation or of the runs created in the previous pass).

The number of passes N will be $N = \lceil \log_{M-1}(B) \rceil + 1$ (since at each pass we process M-1 runs and merge them into a single run).

The minimum amount of available frames to execute the two-way external merge sorting algorithm is 3 (for which we obtain a cost of $2 \times B \times \lceil \log_{M-1}(B) \rceil + 1$ memory accesses), since at each pass we need to load at least 2 pages of each run to be merged and we need a frame to reserve for the output of their merge.

1.3.2. K-Way External Merge Sort

In pass 0 of the previous algorithm, we only consider a single page of the file at a time, sort it and write it back in secondary storage.
This can be optimized by reading M pages of the file at a time instead (with M being the number of available frames in the buffer).
We can then proceed as before, considering M-1 pages of runs at the generic pass i of the algorithm and merging them until we can produce a single sorted run (i.e. the sorted file).

With this generalization, we start pass 1 of the algorithm with $\lceil B/M \rceil$ runs instead of B runs, and therefore will merge runs and arrive at the final pass of the algorithm quicker.

The cost of the K-way external sorting algorithm in terms of memory accesses is again $2 \times B \times N$ where N is the total number of passes.
The number of passes N, in this case, will be $N = \lceil \log_{M-1}(\lceil B/M \rceil) \rceil + 1$ (since at each pass we process M-1 runs and merge them into a single run, starting from $\lceil B/M \rceil$ runs produced at pass 0).

1.3.3. Replacement Selection Strategy

While producing runs for pass 0 of the aforementioned k-way external merge sorting algorithms, we can produce a maximum total number of runs equal to the number of available frames M in the buffer of the main memory.
We optimize pass 0 by trying to create more than M runs to then merge this smaller amount of runs in the generic i-passes of the algorithm even quicker.

The “replacement selection” strategy aims to do this by using a “priority queue” Q1 to assign a certain priority to records currently loaded in the buffer: we reserve a frame for the output and also a frame for the input, and work with M-2 frames which will hold the records of the file which compose queue Q1.
At each iteration of pass 0, we find the minimum record r1 in queue Q1 and “move it” into the output frame, then choose one of the records in the input frame, which we will name r2, to replace the now free slot of record r1 in its corresponding frame (among the M-2 frames we are using).
If this record r2 is greater than record r1, we insert r2 into the priority queue Q1 to be analyzed in one of the next iterations of pass 0.

If, instead, record r2 is smaller than the record we just wrote into the output frame, we insert r2 into a second priority queue, Q2, which we will use later.

Whenever the input frame is empty, we load the next page of the file into the input frame, and continue with the execution of pass 0.

This procedure allows us to create a sorted run containing all the records we will find in queue Q1, which may be more than just the records we can fit into M pages of the buffer (as instead happens in the k-way variation of the algorithm), and therefore may produce a run of more than M pages.

When we can't find records in queue Q1 (hence Q1 empties out), this means that no more record r1 can be found such that r1 is greater than the last record written in the run we are currently creating (i.e. written in the output frame before copying the output frame in secondary storage to extend the current run).
We will therefore need to turn to a new run, and we can do this by moving what is now the content of queue Q2 into the empty queue Q1.
We can now start the creation of a new run as described before, each time replacing the records we pull out from queue Q1 with records in the input frame, and move Q2 to Q1 each time Q1 is empty.

When no more pages of the file can be loaded into the input frame, we simply continue with the algorithm until queues Q1 and Q2 are empty, and then start pass 1 with the generated runs, which may likely have a dishomogeneous number of pages and records, and merge these runs in the subsequent passes as described for the two-way and k-way versions of the algorithm.

The number of runs created at pass 0 is, on average, half of the runs we can produce with pass 0 of the k-way sorting algorithm, meaning that we will be able to eventually merge these runs into a single sorted collection in about half the number of passes we need for the k-way algorithm.
Note that this is an "average case" complexity in terms of memory accesses, since we cannot accurately predict how many runs will be produced starting only from the number of pages of the file B and the number of available frames in the buffer M (as instead we can do with the two-way and k-way approaches).
In a worst case analysis, instead, the number of runs produced at pass 0 may be as large as B (the same number of runs produced at pass 0 using the two-way version of the algorithm).

1.3.4. Other Optimizations

Various kinds of optimizations and variations of the external merge sort algorithm exist.

One of the main optimizations we can consider is the "double buffering" approach, which makes up for the problem of needing to wait for input/output operations executed on the slower secondary storage before being able to proceed with operations of the faster main memory buffer.

The “double buffering” approach partitions the free frames of the buffer into two separate portions, which can individually be considered as a buffer: while input and output operations need to be performed to load or write pages from and into secondary storage for one of the 2 buffers, the main memory can still operate on the other buffer and manipulate and analyze records that were previously loaded in it.

2. Algorithms Visualization

The original project idea consisted of designing and implementing a software for visualizing the external sorting algorithm, with varying size of the relation and varying number of frames in the buffer.
For the final developed project work, the two-way, k-way, and replacement selection variations of the external sorting algorithm were considered for the visualization, and additional features and customization options were implemented and added.

A common algorithm aggregating the 3 possible considered variations of the external merge sort algorithm was considered, with the following pseudocode:

```
// Pass 0
if (sorting using the two-way approach):
        TwoWayPassZero()
else if (sorting using the k-way approach):
        KWayPassZero()
else if (sorting using the replacement selection approach):
        ReplacementSelectionPassZero()
set i = 1
// Pass i
while (there are other runs to consider for pass i):
        select M-1 runs created in pass i-1 and not yet analyzed
        load first page of each run in the buffer
        set pointer of each frame to first record of each page
        while (buffer is not empty):
                find minimum record r among pointed records
                write the minimum record r in the output frame
                if (record r was NOT last of its frame):
                        point to next record for corresponding frame
                else if (corresponding run has more pages):
                        load next page of record r's run to the frame
                        set pointer of the frame to its first record
                if (output frame is full):
                        write output frame in secondary storage
                if (we are at last record of last page for each run):
                        set i = i+1
// File is sorted
```

In particular, a common version of the generic pass i of the algorithm was used, while pass 0 of each of the algorithms was implemented by the pseudocode of the functions called in pass 0.

We have the following pseudocodes for the 3 variations considered:

```
// Pass 0 of the Two-Way External Sorting Algorithm
function TwoWayPassZero():
        while (no more pages in R):
                load 1 page of relation R in the buffer
                sort records of the page in the buffer
                write corresponding page in a new run

// Pass 0 of the K-Way External Sorting Algorithm
function KWayPassZero():
        while (no more pages in R):
                load M pages of relation R in the M frames of the buffer
                sort records of the M pages in the buffer
                write corresponding pages in a new run

// Pass 0 of the Replacement Selection External Sorting Algorithm
function ReplacementSelectionPassZero():
        load first page of relation R in the input frame F1
        load next M-2 pages of R in the buffer
        set Q1 = queue of records in the M-2 frames starting from F2
        set Q2 = empty queue
        while (F1 or Q1 or Q2 are not empty):
                if (Q1 is empty):
                        move Q2 into Q1
                        if (output frame is not empty):
                                write output frame in secondary storage
                        create new run
                find minimum record r1 in Q1
                move r1 in the output frame
                if (F1 is empty and R has more pages):
                        read next page from R
                if (F1 is NOT empty):
                        point to the next record r2 in input frame F1
                        if (r1 < r2):
                                move r2 in Q1
                        else:
                                move r2 in Q2
                if (output frame is full):
                        write output frame in secondary storage
```

Note that for the considered version of the “replacement selection” algorithm, a simple selection strategy for the record r2 of the input frame (which will replace the minimum record r1 of queue Q1) was used, i.e. we simply select the next record of the input frame not yet analyzed to replace record r1 or to be added to queue Q2.
It is worth nothing, though, that this strategy is not optimal, since we may end up overlooking a record contained in the input frame which may potentially be added to the output frame in the next iteration of the while loop, and may also never be added to the output frame in subsequent iterations for the current run (thus ending up in queue Q2 and being considered only for the next run).
A better strategy would be to keep track of the last record written in the output frame, indicated as LAST, and selecting as the record r2 from the input frame the minimum record such that its value (i.e. the value of its key for which we are sorting relation R) is greater (or equal) than the value of LAST, hence **min {** r2 ∈ F_{IN} s.t. r2 ≥ LAST **}** (where “≥” indicates that r2 should come after LAST in the final sorted file).

2.1. Implemented System

The system was implemented as a single HTML document with internal CSS and JavaScript, in order to:

- provide a high accessibility (the entire system is accessible with a single click, using just a browser: no system installation is required, no additional external modules need to be installed, no need for complicated command prompt lines);
- provide a high portability (because the entire system can be accessed from whatever browser, in whatever operating system, by whatever device);
- reduce system size and overhead (a generic browser acts as an “engine” for the system, without the need to include a standalone and ad-hoc engine for the visualization of the system, which would instead increase its total file size).

HTML and CSS where used for the presentation of the system, while the logic and actual code for the visualization was implemented in plain JavaScript, without the use of external libraries (apart for the JQuery library, which allows to write JS code in a more compact and intuitive version; additional notes can be found in the official JQuery website at the link https://jquery.com/): this, once again, allowed to maintain the system as lightweight and accessible as possible, without the need of installing additional software components and heavy external modules to run the system.

Additional external tools were used to create specific aspects of the system itself, for example Adobe Illustrator, a program used for the creation of SVG elements which act in place of images in the HTML and CSS code of the system (e.g. buttons icons) which were URL-encoded to be embedded directly inside the code of the application, without the need for actual PNG or JPG image files to be included in the system’s file.

The system was also made “responsive” using specific CSS instructions (“media queries”) to adapt to the visualization of many browsers, screen sizes and resolutions.

2.2. System Features

The main page of the system is organized in four main sections:

- On the left, the pseudocode of chosen the algorithm variation is shown
- The center section represent the main memory, where the buffer resides
- The section on the right represents the secondary storage, in which the file for relation R and the sorted runs created at each iteration reside
- At the bottom of the screen, controls for the animation and the visualization as well as additional information, options and features are available.

The system allows to enter as input values the number of pages of the file B, the number of available frames in the buffer M, the total number of records of the file, and the max number of records per page (which is proportional to the number of file pages and file records, but allows for a higher customization of the visualization).

The user can choose which of the three algorithm variations to visualize, and once everything is set up, start the animation to be played automatically (with an adjustable animation speed) or play each step of the animation manually.
The system also allows users to pause the animation and freely rewind it by restoring a previous state, to then resume the animation from that specific step.
Furthermore, the animation can obviously be refreshed or restarted whenever the user needs to, by also being able to provide new values for the controllable parameters or change the algorithm version to visualize.

One of the main feature of the system is the step-by-step description of each phase of the algorithm: while the animation plays, the user can follow what is happening in the animation from the algorithm's pseudocode on the left side of the screen (which can also be hidden, i.e. collapsed, to leave more space for the animation itself), which also highlights the lines of the pseudocode currently being animated by the system.
A richer description of each of the algorithm phases is also available below the chosen algorithm's pseudocode on the left: this description provides an in-depth analysis of the specific actions being executed, records being analyzed or sorted, pages of the file or runs being considered, frames of the buffer being written, ecc…
Information about the number of input/output operations performed so far and the current pass being executed are also always available on the bottom of the screen.

Finally, one of the main features of the system is the possibility to access, at any time, the "sorting tree" of the specific chosen algorithm for the chosen input parameters (file pages, available frames, ecc…).
This "sorting tree" shows the situation of the file, with each of its records, at the start of the algorithm, and the runs generated at the end of each of the passes of the algorithm, in a compact and simple way (these runs are the same runs generated by the chosen algorithm and which can be visualized following the sorting animation of the initial file).

Some additional features of the system include the possibility to control the animation using the keyboard (press the spacebar to play or pause the animation, press the left and arrow keys to move one step backwards or forward in the animation, quickly set the

playback speed of the animation by pressing the “1”, “2” and “3” number buttons), access a quick help guide on the main functionalities of the system, customize the visualization with options to control specific features of the animation, like transforming the mouse pointer to an easier to see pointer to allow for presenting the content of the visualization to others (e.g. students while the animation of the system is being presented to students), or enable/disable animation specific functions (such as pseudocode lines highlight, automatic focus of the element being considered by the animation, disable visual effects, ecc…).

2.3. Application Logic

As mentioned before, the core logic of the system was implemented using JavaScript, the “de-facto” standard, high-level and multi-paradigm language for the web (supporting event-driven, functional and imperative programming styles).

When the page initially loads, a JS script assigns the correct “event listeners”, functionalities and behaviors to the buttons and elements of the screen, and alongside the CSS styles defined for the various elements, the web page is initially built and made ready to be interacted with by the user.

An initialization function is used for the single elements of the visualization for values provided as inputs by the user (number of pages of the file, number of available frames, ecc…), and using these values, the initialization function builds the single HTML elements corresponding to the records and pages of the file to display, as well as the frames of the buffer (the layout of the buffer is also changed to allow for a more effective visualization in some cases, for example in the “replacement selection” algorithm, which shows separate portions of frames to correctly display input and output frames and frames which will contain records for queues Q1 and Q2).

This initialization function correctly updates the visualization of each element each time it needs to be refreshed, hence each time the user changes some of the input parameters, the selected algorithm to visualize, or any customization options provided by the system.

2.3.1. Animation Phases

The core “animation” for the visualization of the algorithm (hence both the automatic playback and the step-by-step behavior of the animation) is built “on-the-fly” when the user starts the animation’s automatic playback or manually steps forward for the first time after inserting the desired values and choosing the desired algorithm to visualize.

At start, the animation is built as an array of animation “phases”, represented using JavaScript objects, each with its own properties, parameters, functions, ecc…
The array of animation phases is reset each time the user resets the animation, to allow for the creation of a new animation phases array when the user changes the input values and parameters or the algorithm type.

The structure of each of the animation phases JS objects stored in the animation phases array is the following:

```
// Single animation phase object (as found in the animation phases array)
let animation_phase = {
   // Function implementing the visualized animation for this phase
   play: function () {
       // ...
   },
   // Function which allows to undo the animation of the "play" function
   //      for this phase and return to the state prior its execution
   rollback: function () {
       // ...
   },
   // Function that returns the duration in milliseconds of the this
   //      phase's animation (as defined in the "play function")
   duration: function () {
       // return ...
   },
   // Nested object containing the information about the phase
   phase_info: {
       // Array of lines of the pseudocode to highlight for this phase
       code_line: [...],
       // Offset for the corresponding code lines of this phase
       //      (used for specific cases of some “pass i”’s phases)
       code_line_offset: Number,
       // Pass number of this phase (pass 0, pass 1, pass 2, ...)
       pass_number: Number,
       // Number of input/output operations executed in this phase
       io_operations: Number,
       // Function that returns the description string for this phase
       description: function () {
           // return ...
       }
   },
}
```

An initial description of each function and attribute is provided as a comment in the above structure, while the following provides a more detailed description of their most important aspects.

The animation phases of the array are used to play the animation of each phase itself (implemented by the “play” function), which manipulates the HTML elements corresponding to records, pages, runs, frames, ecc… to move elements on the screen, make them appear and disappear, animate specific aspects of each element, change their displayed aspect, ecc….
Some of the contextual information of the phase and other visualization-related elements are also manipulated using this specific function.

Furthermore, animation phases stored in the corresponding array contain other functions and attributes used to rollback the animation to a previous state (using the “rollback” function, for when the “step backwards” button is pressed by the user), to calculate the time needed to animate the current phase (using the “duration” function, also based on the current animation playback speed), and update the visualization of the specific information of the phase (using information provided by the “phase_info” attribute, including the detailed description of the phase, the related algorithm’s pseudocode lines, the number of total input output operations, the pass number of the current phase).

Other than the “play”, “rollback” and “duration” functions, the “phase info” attribute of each phase JS object, which is in turn another object, contains various information used by the core animation controller to display the previously mentioned contextual information of each phase, which are visible at the bottom and left of the screen. These information include the highlighted line in the selected algorithm’s pseudocode, the detailed description of the algorithm shown below said pseudocode section on the left, the pass number of the current phase and the total counter of input/output operations executed by the algorithm (computed by summing up the “phase_info.io_operations” attribute at each phase, or subtracting it in case of step backwards).

Each phase of the animation phase array is built and pushed into the array using a function to initialize the animation itself, which corresponds to the “real” implementation of the pseudocode shown in “Chapter 2”, and therefore the implementation of each of the three algorithm variations’ passes.

The initialization function for the animation actually sorts the elements of the file according to the corresponding chosen algorithm version (two-way, k-way or replacement selection), creating the corresponding runs of each pass, merging them in the next pass, and then arriving at the actually sorted file and pages of the file.
All of this is done without considering the visualized elements, but just in order to initialize the animation phases array itself.

Each step of the actual implemented algorithm in JavaScript also builds the JS object of the corresponding animation phase and pushes it into the animation phases array, working “in parallel” with the actual sorting of the records of the file.

2.3.2. Sorting Tree

The “sorting tree” mentioned above is also built using the aforementioned initialization function for the animation phases array.

In particular, at the end of each “pass” of the function implementing the pseudocode shown in “Chapter 2”, the state of the records of the file, and therefore of the created runs, is saved and then used to build the visualized records, pages and runs of the “sorting tree” (accessible using the corresponding button in the bottom right of the screen) in a similar way to the creation of the elements for the initial visualization of the records and pages of the file.

2.3.3. The Undo Problem

In order to correctly implement the “undo” function of the system, a “stateful” version of the animation needed to be implemented.

As mentioned before, each animation phase holds functions and information about the phase itself, and then is stored in an array: this allows to keep track of the index of the current phase in the array which is being animated in the system, and use this index to move the animation forwards and backwards.

While the “play” function of each phase’s associated JS object is used to animate the movement of records and change of appearance of the various elements of the visualization, the “rollback” function allows to move to a state corresponding to the end of the animation of the “play” function, and therefore implements the same actions and operations of said function, plus a set of actions that are needed to rollback from the state of the next animation phase.
This means that the rollback function of a generic phase p will both execute operations and actions in a similar way to the “play” function of this phase (which, however, differ from the “play” function actions themselves because of the lack of an actual animation, e.g. the movement of records on screen, the scale up of elements, ecc…) and also revert any change made by phase p+1 in its corresponding “play” function.

In summary, each animation phase holds all the needed information to display the actual situation of each element of the visualization (records, pages, runs, ecc…) and all the needed information needed to revert the changes which will be made by the next phase of the animation.

3. Additional Notes and Conclusions

3.1. Development Process

The system was developed for the final project work of the course of Data Management (A.Y. 2022/2023) at Sapienza University of Rome, by prof. Maurizio Lenzerini.

360 lines of HTML code
1.830 lines of CSS code
5.260 lines of JS code

Code available on GitHub at https://github.com/valeriodiste/ExternalMergeSortVisualizer

3.2. Conclusions

The design and implementation of this External Merge Sort visualization system was aimed at improving the learning experience of students by providing a simple, intuitive, lightweight and easily accessible system.

I believe visualization tools for algorithms are one of the most efficient ways of learning algorithms themselves, because they provide a fun and interactive way of understanding aspects and details which wouldn't be easy to grasp and comprehend by simply reading the algorithm's pseudocode (which I believe to be extremely important as well, and was in fact provided on the left side of the screen to always be accessible while interacting with the animation). Visual representations of algorithms are also very effective for comparing different versions of specific algorithms themselves (the two-way, k-way and replacement selection strategies in this case).

In developing the system, I also had the opportunity to improve my knowledge and understanding of this type of sorting algorithms, by further researching aspects of these algorithms and interesting variations.
The development of final system also helped me learn and figure out how these algorithms actually work, because of the need to actually implement the algorithms themselves to be able to build an animation for their visualization, and the visuals helped me figure out errors and initial misunderstandings or biases for what I originally understood about these algorithms.
In particular, the "replacement selection" strategy was challenging to learn and deeply understand, and implementing the algorithm itself helped me to understand it at my best.

In the end, the developed external merge sorting visualization system was very helpful for my learning experience, and I hope will be very helpful for both the students willing to learn more about it by using the system and teachers willing to provide an easy to follow description to their students with the help of the system.

4. References

1. Lecture notes for the course of Data Management (2022/2023)
   Chapter 7 - File Organization (Pages 40-75)
   M. Lenzerini

2. Lecture notes for the course of Data Management (2021/2022)
   Chapter 5 - File Organization
   M. Lenzerini

3. Online Lectures Recordings: Data Management (2022/2023)
   Lecures 42, 43 (04 May 2023)
   M. Lenzerini

4. Online Lectures Recordings: Data Management (2021/2022)
   Lectures 31, 32, 33, 34 (08 April 2022)
   M. Lenzerini

5. Wikipedia - External Sorting
   https://en.wikipedia.org/wiki/External_sorting

6. OpenDSA - External Sorting
   Chapter 14 - File Processing
   https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/ExternalSort.html

7. Online Lectures Recordings: Database Management Systems (2011/2021)
   Chapter 8 - External Sorting, Merge Sort, Double Buffering, Replacement Sort
   S. Simonson
   https://youtu.be/YjFI9CJy6x0

8. Database Management Systems. McGraw-Hill, 2004
   Chapter 11 - External Sorting
   R. Ramakrishnan, J. Gehrke.