4

Sorting Atomic Items

This lecture will focus on the very-well know problem of sorting a set of *atomic* items, the case of *variable-length* items (aka strings) will be addressed in the following chapter. Atomic means that they occupy $O(1)$ space, typically they are integers or reals represented with a fixed number of bytes, such as 4 (32 bits) or 8 (64 bits) bytes each.

The sorting problem. *Given a sequence of n atomic items $S[1,n]$ and a total ordering $\leq$ between each of them, sort S in increasing order.*

We will consider two complemental sorting paradigms: the *merge-based* paradigm, which underlies the design of Mergesort, and the *distribution-based* paradigm which underlies the design of Quicksort. We will adapt them to work in the disk model (see Chapter 1), analyze their I/O-complexities and propose some useful tools that can allow to speed up their execution in practice, such as the Snow Plow technique and Data compression. We will also demonstrate that these disk-based adaptations are *I/O-optimal* by proving a sophisticated lower-bound on the number of I/Os any external-memory sorter must execute to produce an ordered sequence. In this context we will relate the Sorting problem with the so called *Permuting* problem, typically neglected when dealing with sorting in the RAM model.

The permuting problem. *Given a sequence of n atomic items $S[1,n]$ and a permutation $\pi[1,n]$ of the integers $\{1,2,\ldots,n\}$, permute S according to π thus obtaining the new sequence $S[\pi[1]], S[\pi[2]], \ldots, S[\pi[n]]$.*

Clearly Sorting includes Permuting as a sub-task: to order the sequence S we need to determine its sorted permutation and implement it (possibly these two phases are intricately intermingled). So Sorting should be more difficult than Permuting. And indeed in the RAM model we know that sorting n atomic items takes $\Theta(n \log n)$ time (via Mergesort or Heapsort) whereas permuting them takes $\Theta(n)$ time. The latter time bound can be obtained by just moving one item at a time according to what is indicated in the array π. But surprisingly we will show that this *complexity gap* does not exist in the disk model, in that they exhibit the same I/O-complexity under some reasonable

conditions on the input and model parameters n, M, B. This elegant and deep result was obtained by Aggarwal and Vitter in 1998 [1], and it is surely the result that spurred the huge amount of algorithmic literature thereafter produced on the I/O-subject. Philosophically speaking, AV's result formally proves the intuition that *moving* items in the disk is the real efficiency *bottleneck*, rather than the underlying computation needed to determine those movements. And indeed researchers and software engineers typically speak about the *I/O-bottleneck* to characterize this issue in their (slow) algorithms.

We will conclude this lecture by briefly mentioning at two solutions for the problem of sorting items on D-disks: the disk-striping technique, which is at the base of RAID systems and turns any efficient/optimal 1-disk algorithm into an efficient D-disk algorithm (typically loosing its optimality, if any), and the Greed-sort algorithm, which is specifically tailored for the sorting problem and achieves I/O-optimality.

4.1 The merge-based sorting paradigm

We recall the main features of the external-memory model introduced in Chapter 1: it consists of an internal memory of size M and it allows blocked-access to disk data by reading/writing B items at once.

Algorithm 4.1 The binary merge-sort: MergeSort(S, i, j)

```
1: if (i < j) then
2:     m = (i + j)/2;
3:     MergeSort(S, i, m − 1);
4:     MergeSort(S, m, j);
5:     Merge(S, i, m, j);
6: end if
```

Mergesort is based on the Divide&Conquer paradigm. Step 1 checks if the array to be sorted consists of at least two items, otherwise it is already ordered and nothing has to be done. Otherwise it splits the input array S into two halves, according to the position m of the *middle* element, and then recurse on each part. As recursion ends, the two halves $S[i, m-1]$ and $S[m, j]$ are sorted so that Step 5 invokes procedure Merge that fuses their contents and produce a *unique* sorted sequence that is stored in $S[i, j]$. This merging needs an auxiliary array of size n, so that MergeSort is not an *in-place* sorting algorithm (unlike Heapsort and Quicksort) but needs $O(n)$ extra working space. Given that at each recursive call we halve the size of the input array to be sorted, the total number of recursive calls is $O(\log n)$. The Merge-procedure can be implemented in $O(j - i + 1)$ time by using two pointers, say x and y, that start at the beginning of the two halves $S[i, m-1]$ and $S[m, j]$. Then $S[x]$ is compared with $S[y]$, the smaller is written out in the fused sequence, and its pointer is advanced. Given that each comparison advance one pointer, the total number of steps is bounded above by the total number of pointer's advancements, which is upper bounded by the length of $S[i, j]$. So the time complexity of MergeSort$(S, 1, n)$ can be modeled via the recurrent relation $T(n) = 2T(n/2) + O(n) = O(n \log n)$, as well known from any basic algorithm course.[1]

Let us assume now that $n > M$, so that S must be stored on disk and I/Os because the most im-

[1] In all our lectures when the base of the logarithm is not indicated, it means 2.

portant resource to be analyzed. In practice every I/O takes 5ms on average, so one could think that every item comparison takes one I/O and thus estimate the running time of Mergesort on massive data as: 5ms $\times\Theta(n \log n)$. If n is of the order of few Gigabytes, say $n \approx 2^30$ which is actually not much massive for the current-size of commodity PCs, the previous time estimate would be of about $5 \times 2^30 \times 30 > 10^8$ms, namely more than 1 day of computation. However, if we run Mergesort on a commodity PC it completes in few hours. This is not surprising because the previous evaluation totally neglected the existence of the internal memory, of size M, and the sequential pattern of memory-accesses induced by Mergesort. Let us therefore analyze the Mergesort algorithm in a more precise way within the disk model.

First of all we notice that $O(z/B)$ I/Os is the cost of merging two ordered sequences of z items in total. This holds if $M \geq 2B$, because the Merge-procedure in Algorithm **??** can keep in internal memory the 2 pages that contain the two pointers scanning $S[i, j]$. Every time a pointer advances into another disk page, an I/O-fault occurs, the page is fetched in internal memory, and the fusion continues. Given that S is stored contiguously on disk, it occupies $O(z/B)$ pages and this is the I/O-bound for merging two sub-sequences of total size z. Similarly, the I/O-cost for writing the merged sequence is $O(z/B)$ because it occurs sequentially from the smallest to the largest item of $S[i, j]$. As a result the recurrent relation for the I/O-complexity of Mergesort can be written as $T(n) = 2T(n/2) + O(n/B) = O(\frac{n}{B} \log n)$ I/Os. But this formula does not explain completely the good behavior of Mergesort in practice, because it does not account for the memory hierarchy yet. In fact as Mergesort recursively splits the sequence S, smaller and smaller sub-sequences are generated that have to be sorted. So when a subsequence of length ℓ fits in internal memory, namely $\ell = \Theta(M)$, then it will be entirely cached by the underlying operating system using $O(\ell/B)$ I/Os and thus, the subsequent sorting steps would not incur in any I/Os. The net result of this simple observation is that the I/O-cost of sorting a sub-sequence of $\ell = O(M)$ items is no longer $T(\ell) = \Theta(\frac{\ell}{B} \log \ell)$ but $O(\ell/B)$. This saving applies to all S's subsequences of size $\Theta(M)$ on which Mergesort is recursively run, which are $\Theta(n/M)$ in total. So the overall saving is $\Theta(\frac{n}{B} \log M)$, which leads us to re-formulate the Mergesort's complexity as $\Theta(\frac{n}{B} \log \frac{n}{M})$ I/Os. This bound is particularly interesting because relates the I/O-complexity of Mergesort not only to the disk-page size B but also to the internal-memory size M, and thus to the *caching* available for the computation. Moreover this bounds suggests three immediate optimizations to the classic pseudocode of Algorithm 4.3 the we discuss below.

Optimization #1: Taking care of internal-memory size. The first optimization consists of introducing a threshold on the subsequence size, say $j - i < cM$, which triggers the stop of the recursion, the fetching of that subsequence entirely in internal-memory, and the application of an internal-memory sorter on this sub-sequence (see Figure 4.1). The value of the parameter c depends on the space-occupancy of the sorter, which must be guaranteed to work entirely in internal memory. As an example, c is 1 for in-place sorters such as Insertionsort and Heapsort, it is much close to 1 for Quicksort (because of its recursion), and it is less than 0.5 for Mergesort (because of the extra-array used by MERGE). As a result, we should write cM instead of M into the I/O-bound above, because recursion is stopped at cM items: thus obtaining $\Theta(\frac{n}{B} \log \frac{n}{cM})$. This substitution is useless when dealing with asymptotic analysis, given that c is a constant, but it is important when considering the real performance of algorithms. In this setting it is desirable to make c as closer as possible to 1, in order to reduce the logarithmic factor in the I/O-complexity thus preferring in-place sorters such as Heapsort or Quicksort. We remark that Insertionsort could also be a good choice (and indeed it is) whenever M is small, as it occurs when considering the sorting of items over the 2-levels: L1 and L2 caches, and the internal memory. In this case M would be few Megabytes.

Optimization #2: Virtually enlarging M. Looking at the I/O-complexity of mergesort, i.e. $\Theta(\frac{n}{B} \log \frac{n}{M})$, is clear that the larger is M the smaller is the number of merge-passes over the data. These are clearly the bottleneck to the efficient execution of the algorithm especially in the presence of disks with low

bandwidth. In order to circumvent this problem we can either buy a larger memory, or try to deploy as much as possible the one we have available. As algorithm engineer we opt for the second possibility and thus propose two techniques that can be combined together in order to enlarge (virtually) M.

The first technique is based on data compression and builds upon the observation that the runs are increasingly sorted. So, instead of representing items via a fixed-length coding (e.g. 4 or 8 bytes), we can use *integer compression* techniques that squeeze those items in fewer bits thus allowing us to pack more of them in internal memory. A following lecture will describe in detail several approaches to this problem (see Chapter **??**), here we content ourselves mentioning the names of some of these approaches: γ-code, δ-code, Rice/Golomb-coding, etc. etc.. In addition, since the smaller is an integer the fewer bits are used for its encoding, we can enforce the presence of small integers in the sorted runs by encoding not just their absolute value but the *difference* between one integer and the previous one in the sorted run (the so called *delta*-coding). This difference is surely non negative (equals zero if the run contains equal items), and smaller than the item to be encoded. This is the typical approach to the encoding of integer sequences used in modern search engines, that we will discuss in a following lecture (see Chapter **??**).

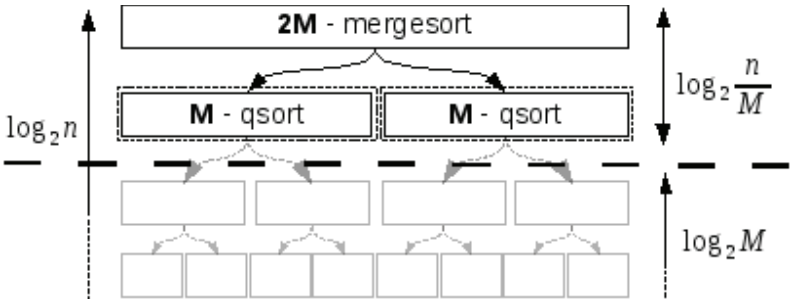

FIGURE 4.1: When a run fits in the internal memory of size M, we apply `qsort` over its items. In gray we depict the recursive calls that are executed in internal memory, and thus do not elicit I/Os. Above there are the calls based on classic Mergesort, only the call on $2M$ items is shown.

The second technique is based on an elegant idea, called the `Snow Plow` and due to D. Knuth [2], that allows to *virtually* increase the memory size of a factor 2 *on average*. This technique scans the input sequence S and generates sorted runs of whose length has variable size longer than M and $2M$ on average. Its use needs to change the sorting scheme because it first creates these sorted runs, of variable length, and then applies repeatedly over the sorted runs the MERGE-procedure. Although runs will have different lengths, the MERGE will operate as usual requiring an optimal number of I/Os for their merging. Hence $O(n/B)$ I/Os will suffice to halve the number of runs, and thus a total of $O(\frac{n}{B}\log\frac{n}{2M})$ I/Os will be used on average to produce the totally ordered sequence. This corresponds to a saving of 1 pass over the data, which is non negligible if the sequence S is very long.

For ease of description, let us assume that items are transferred one at a time from disk to memory, instead that block-wise. Eventually, since the algorithm scans the input items it will be apparent that the number of I/Os required by this process is linear in their number (and thus optimal). The algorithm proceeds in phases, each phase generates a sorted run (see Figure 4.2 for an illustrative example). A phase starts with the internal-memory filled of M (unsorted) items, stored in a heap data structure called $\mathcal{H}$. Since the array-based implementation of heaps requires no additional space, in addition to the indexed items, we can fit in $\mathcal{H}$ as many items as we have memory cells available.

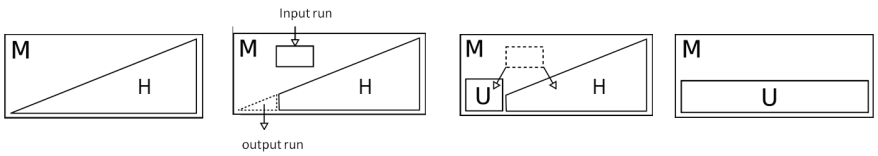

FIGURE 4.2: An illustration of four steps of a phase in Snow Plow. The leftmost picture shows the starting step in which $\mathcal{U}$ is heapified, then a picture shows the output of the minimum element in $\mathcal{H}$, hence the two possible cases for the insertion of the new item, and finally the stopping condition in which $\mathcal{H}$ is empty and $\mathcal{U}$ fills entirely the internal memory.

The phase scans the input sequence S (which is unsorted) and at each step, it writes to the output the minimum item within $\mathcal{H}$, say `min`, and loads in memory the next item from S, say `next`. Since we want to generate a sorted output, we cannot store `next` in $\mathcal{H}$ if `next` < `min`, because it will be the new heap-minimum and thus will be written out at the next step thus destroying the property of increasing run. So in that case `next` must be stored in an auxiliary array, called $\mathcal{U}$, which stays unsorted. Of course the total size of $\mathcal{H}$ and $\mathcal{U}$ is M over the whole execution of a phase. A phase stops whenever $\mathcal{H}$ is empty and thus $\mathcal{U}$ consists of M unsorted items, and the next phase can thus start (storing those items in a new heap $\mathcal{H}$ and emptying $\mathcal{U}$). Two observations are in order: (i) during the phase execution, the minimum of $\mathcal{H}$ is non decreasing and so it is the output, (ii) the items in $\mathcal{H}$ at the beginning of the phase will be eventually written to output which thus is longer than M. Observation (i) implies the correctness, observation (ii) implies that this approach is not less efficient than the classic Mergesort.

Algorithm 4.2 A phase of the Snow-Plow technique

Require: $\mathcal{U}$ is an unsorted array of M items
1: $\mathcal{H}$ = build a min-heap over $\mathcal{U}$'s items;
2: Set $\mathcal{U} = \emptyset$;
3: **while** ($\mathcal{H} \neq \emptyset$) **do**
4: `min` = Extract minimum from $\mathcal{H}$;
5: Write `min` to the output run;
6: `next` = Read the next item from the input sequence;
7: **if** (`next` < `min`) **then**
8: write `next` in $\mathcal{U}$;
9: **else**
10: insert `next` in $\mathcal{H}$;
11: **end if**
12: **end while**

Actually it is more efficient than that on average. Suppose that a phase reads τ items in total from S. By the while-guard in Step 3 and our comments above, we can derive that a phase ends when $\mathcal{H}$ is empty and $|\mathcal{U}| = M$. We know that the read items go in part in $\mathcal{H}$ and in part in $\mathcal{U}$. But since items are added to $\mathcal{U}$ and never removed during a phase, M of the τ items end-up in $\mathcal{U}$. Consequently $(\tau - M)$ items are inserted in $\mathcal{H}$ and eventually written to the output (sorted) run. So the length of the sorted run at the end of the phase is $M + (\tau - M)$, where the first addendum accounts for the items in $\mathcal{H}$ at the beginning of a phase, whereas the second addendum accounts for the items read

from S and inserted in $\mathcal{H}$ during the phase. The key issue now is to compute the average of τ. This is easy if we assume a random distribution of the input items. In this case we have probability $1/2$ that `next` is smaller than `min`, and thus we have equal probability that a read item is inserted either in $\mathcal{H}$ or in $\mathcal{U}$. Overall it follows that $\tau/2$ items go to $\mathcal{H}$ and $\tau/2$ items go to $\mathcal{U}$. But we already knew that the items inserted in $\mathcal{U}$ were M, so we can set $M = \tau/2$ and thus $\tau = 2M$. Substituting in the formula $M + (\tau - M)$ we get that the average number of items written in the sorted run is $2M$.

FACT 4.1 *Snow-Plow builds $O(n/M)$ sorted runs, each longer than M and actually of length $2M$ on average. Using Snow-Plow for the formation of sorted runs in a Merge-based sorting scheme achieves an I/O-complexity of $O(\frac{n}{B}\log_2\frac{n}{2M})$.*

Optimization #3: From binary to multi-way Mergesort. Previous optimizations deployed the internal-memory size M to reduce the number of recursion levels by increasing the size of the initial (sorted) runs. But then the merging was *binary* in that it fused two input runs at a time. This binary-merge impacted onto the base 2 of the logarithm of the I/O-complexity of Mergesort. Here we wish to increase that base to a much larger value, and in order to get this goal we need to deploy the memory M also in the merging phase by enlarging the number of runs that are fused at a time. In fact the merge of 2 runs uses only 3 blocks of the internal memory: 2 blocks are used to cache the current disk pages that contain the compared items, namely $S[x]$ and $S[y]$ from the notation above, and 1 block is used to cache the output items which are flushed when the block is full (so to allow a block-wise writing to disk of the merged run). But the internal memory contains a much larger number of blocks, i.e. $M/B \gg 3$, which remain unused over the whole merging process. The third optimization we propose consists of deploying all those blocks by designing a k-way merging scheme that fuses k runs at a time, with $k \gg 2$. Let us set $k = (M/B) - 1$, so that k blocks are available to read block-wise k input runs, and 1 block is reserved again for a block-wise writing of the merged run to disk. This scheme poses a challenging merging problem because at each step we have to select the minimum among k candidates items and this cannot be obviously done brute-forcedly by iterating among them. We need a smarter solution that again hinges onto the use of a min-heap data structure, which contains k pairs (one per input run) each consisting of two components: one denoting an item and the other denoting the origin run. Initially the items are the minimum items of the k runs, and so the pairs have the form $\langle R_i[1], i\rangle$, where R_i denotes the ith input run and $i = 1, 2, \ldots, k$. At each step, we extract the pair containing the current smallest item in $\mathcal{H}$ (given by the first component of its pairs), write that item to output and insert in the heap the next item in its origin run. As an example, if the minimum pair is $\langle R_m[x], m\rangle$ then we write in output $R_m[x]$ and insert in $\mathcal{H}$ the new pair $\langle R_m[x+1], m\rangle$, provided that the mth run is not exhausted, in which case no pair replaces the extracted one. In the case that the disk page containing $R_m[x+1]$ is not cached in internal memory, an I/O-fault occurs and that page is fetched, thus guaranteeing that the next B reads from run R_m will not elicit any further I/O. It should be clear that this merging process takes $O(\log_2 k)$ time per item, and again $O(z/B)$ I/Os to merge k runs of total length z. As a result the merging-scheme recalls a k-way tree with $O(n/M)$ leaves (runs) which can have been formed using any of the optimizations above (possibly via Snow Plow). Hence the total number of merging levels is now $O(\log_{M/B}\frac{n}{M})$ for a total volume of I/Os equal to $O(\frac{n}{B}\log_{M/B}\frac{n}{M})$. We observe that sometime we will also write the formula as $O(\frac{n}{B}\log_{M/B}\frac{n}{M})$, as it typically occurs in the literature, by exchanging the denominator within the logarithm's argument. This makes no difference asymptotically given that $\log_{M/B}\frac{n}{M} = \Theta(\log_{M/B}\frac{n}{B})$.

THEOREM 4.1 *Multi-way Mergesort takes $O(\frac{n}{B}\log_{M/B}\frac{n}{M})$ I/Os and $O(n\log n)$ time to sort n atomic items in a two-level model in which the internal memory has size M and the disk page has*

size B. The use of Snow-Plow or integer compressors would virtually increase the value of M with a twofold advantage in the final I/O-complexity, because M occurs twice in the I/O-bound.

In practice the number of merging levels will be very small: assuming a block size $B = 4$KB and a memory size $M = 4$GB, we get $M/B = 2^{32}/2^{12} = 2^{20}$ so that the number of passes is $1/20$th smaller than the ones needed by binary Mergesort. Probably more interesting is to observe that one pass is able to sort $n = M$ items, but two passes are able to sort M^2/B items, since we can merge M/B-runs each of size M. It goes without saying that in practice the internal-memory space which can be dedicated to sorting is smaller than the *physical* memory available (typically MBs versus GBs). Nevertheless it is evident that M^2/B is of the order of Terabytes already for $M = 128$MB and $B = 4$KB.

4.2 A lower bound on I/Os

At the beginning of this lecture we commented on the relation existing between the Sorting and the Permuting problems, concluding that the former one is more difficult than the latter in the RAM model. The gap in time complexity is given by a logarithmic factor. The question we address in this section is whether this gap does exist also when measuring I/Os. Surprisingly enough we will show that in terms of I/O-volume Sorting is *equivalent* to Permuting. This result is amazing because it can be read as saying that the I/O-cost for sorting is not in the *computation* of the sorted permutation but rather the *movement* of the data on the disk to realize it. This is the so called *I/O-bottleneck* that has in this result the mathematical proof and quantification.

Before digging into the proof of this lower bound, let us briefly show how a sorter can be used to permute a sequence of items $S[1,n]$ in accordance to a given permutation $\pi[1,n]$. This will allow us to derive an upper bound to the number of I/Os which suffice to solve the Permuting problem on any $\langle S, \pi\rangle$. Recall that this means to generate the sequence $S[\pi[1]], S[\pi[2]], \ldots, S[\pi[n]]$. In the RAM model we can jump among S's items according to permutation π and create the new sequence $S[\pi[i]]$ taking $O(n)$ optimal time. On disk we have actually two different algorithms which induce two incomparable I/O-bounds. The first algorithm consists of mimicking what is done in RAM, paying one I/O per moved item and thus taking $O(n)$ I/Os. The second algorithm consists of generating a proper set of tuples and then sort them. Precisely, the algorithm creates the sequence $\mathcal{P}$ of pairs $\langle i, \pi[i]\rangle$ where the first component indicates the position i where the item $S[\pi[i]]$ must be stored. Then it sorts these pairs according to the π-component, and via a parallel scan of S and $\mathcal{P}$ substitutes $\pi[i]$ with the character $S[\pi[i]]$, thus creating the new pairs $\langle i, S[\pi[i]]\rangle$. Finally another sort is executed according to the first component of these pairs, thus obtaining a sequence of items correctly permuted. The algorithm uses two scan of the data and two sorts, so it needs $O(\frac{n}{B}\log_{M/B}\frac{n}{M})$ I/Os.

THEOREM 4.2 *Permuting n items takes $O(\min\{n, \frac{n}{B}\log_{M/B} n/M\})$ I/Os in a two-level model in which the internal memory has size M and the disk page has size B.*

In what follows we will show that this algorithm, in its simplicity, is I/O-optimal. The two upper-bounds for Sorting and Permuting equal each other whenever $n = \Omega(\frac{n}{B}\log_{M/B}\frac{n}{M})$. This occurs when $B > \log_{M/B}\frac{n}{M}$ that holds always in practice because that logarithm term is about 2 or 3 for values of n up to many Terabytes. So programmers should not be afraid to find sophisticated strategies for moving their data in the presence of a permutation, just sort them, you cannot do better!

A lower-bound for Sorting. There are some subtle issues here that we wish to do not investigate too much, so we hereafter give only the intuition which underlies the lower-bounds for both Sorting