4

Sorting Atomic Items

This lecture will focus on the very-well known 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 a constant-fixed number of memory cells, typically they are integers or reals represented with a fixed number of bytes, say 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 pair 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 then 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. Surprisingly we will show that this *complexity gap* does not exist in the disk model, in that these two problems 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 *bottleneck*, rather than *finding* the sorted permutation. 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 on D-disks 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 allows blocked-access to disk 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. If items are more than two, it splits the input array S into two halves, and then recurses on each part. As recursion ends, the two halves $S[i, m-1]$ and $S[m, j]$ are ordered so that Step 5 fuses them in $S[i, j]$ by invoking procedure Merge. This merging step 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 advances 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 recurrence 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 become the most important resource to be analyzed. In practice every I/O takes 5ms on average, so one could think

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

that every item comparison takes one I/O and thus one could estimate the running time of Mergesort on a massive S 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 memorys, 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 4.1 can keep in internal memory the 2 pages that contain the two pointers scanning $S[i, j]$ where $z = j - i + 1$. 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, $S[i, j]$ 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 $S[i, j]$ by using an auxiliary array. 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 z fits in internal memory, namely $z = O(M)$, then it will be entirely cached by the underlying operating system using $O(z/B)$ 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 $z = O(M)$ items is no longer $\Theta(\frac{z}{B} \log z)$, as accounted for in the previous recurrence relation, but it is $O(z/B)$ I/Os which accounts only the cost of loading the subsequence in internal memory. 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 at the sorter. Moreover this bounds suggests three immediate optimizations to the classic pseudocode of Algorithm 4.1 that we discuss below.

4.1.1 Stopping recursion

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.

4.1.2 Snow Plow

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 passes 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 **??**).

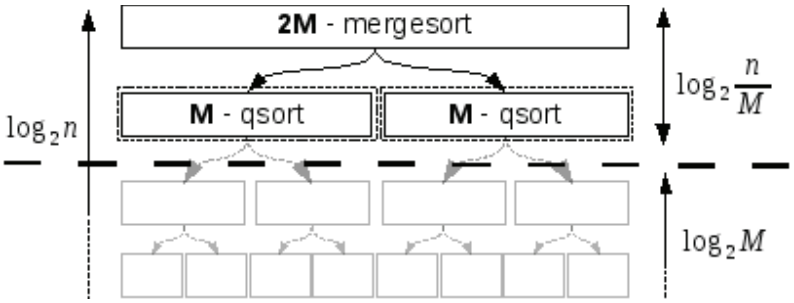

FIGURE 4.1: When a run fits in the internal memory of size M, we apply `qsort` over its items. In gray we depict the recursive calls that are executed in internal memory, and thus do not elicit I/Os. Above there are the calls based on classic Mergesort, only the call on $2M$ items is shown.

The second technique is based on an elegant idea, called the `Snow Plow` and due to D. Knuth [3], 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 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

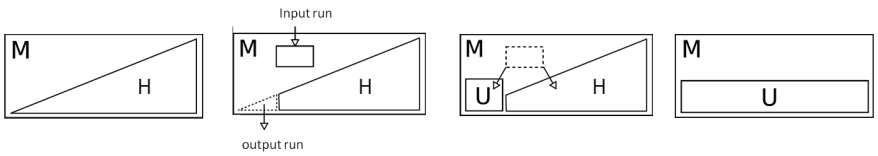

FIGURE 4.2: An illustration of four steps of a phase in Snow Plow. The leftmost picture shows the starting step in which $\mathcal{U}$ is heapified, then a picture shows the output of the minimum element in $\mathcal{H}$, hence the two possible cases for the insertion of the new item, and finally the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

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. 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 it will be written out at the next step thus destroying the property of ordered 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 non-decreasing also the output run, (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 $M + (\tau - M) = \tau$, 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 know that the items inserted in $\mathcal{U}$ are M, so we can set $M = \tau/2$ and thus we get $\tau = 2M$.

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, this achieves an I/O-complexity of $O(\frac{n}{B}\log_2 \frac{n}{2M})$ on average.*

4.1.3 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, therefore 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 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 $O(\frac{n}{B}\log_{M/B} \frac{n}{B})$, as it typically occurs in the literature, because $\log_{M/B} M$ can be written as $\log_{M/B}(B\times(M/B)) = (\log_{M/B} B)+1 = \Theta(\log_{M/B} B)$. 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 $O(n \log n)$ comparisons/-time to sort n atomic items in a two-level memory 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 Lower bounds

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 Sorting is *equivalent* to Permuting in terms of I/O-volume. 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 $S[\pi[i]]$, for $i = 1, 2, \ldots, n$, thus 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 item $S[\pi[i]]$, 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 $O(\min\{n, \frac{n}{B} \log_{M/B} \frac{n}{M}\})$ I/Os in a two-level memory 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!

	time complexity (RAM model)	I/O complexity (two-level memory model)
Permuting	$O(n)$	$O(\min\{n, \frac{n}{B} \log_{M/B} \frac{n}{M}\})$
Sorting	$O(n \log_2 n)$	$O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{M})$

TABLE 4.1 Time and I/O complexities of the Permuting and Sorting problems in a two-level memory model in which M is the internal-memory size, B is the disk-page size, and $D = 1$ is the number of available disks. The case of multi-disks presents the multiplicative term n/D in place of n.

4.2.1 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 and Permuting.[2] We start by resorting the comparison-tree technique for proving comparison-based lower bounds in the RAM model. An algorithm corresponds to a family of such trees, one per input size (so infinite in number). Every node is a comparison between two items. The comparison has two possible results, so the fan-out of each internal node is two and the tree is binary. Each leaf of the tree corresponds to a solution of the underlying problem to be solved: so in the case of sorting, we have one leaf per permutation of the input. Every root-to-leaf path in the comparison-tree corresponds to a computation, so the longest path corresponds to the worst-case number of comparisons executed by the algorithm. In order to derive a lower bound, it is therefore enough to compute the depth of the shallowest binary tree having that number of leaves. The shallowest binary tree with ℓ leaves is the (quasi-)perfectly balanced tree, for which the height h is such that $2^h \geq \ell$. Hence $h \geq \log_2 \ell$. In the case of sorting $\ell = n!$ so the classic lower bound $h = \Omega(n \log_2 n)$ is easily derived by applying logarithms at both sides of the equation and using the Stirling's approximation for the factorial.

In the two-level memory model the use of comparison-trees is more sophisticated. Here we wish to account for I/Os, and exploit the fact that the information available in the internal memory can be used for free. As a result every node corresponds to one I/O, the number of leaves equals still to $n!$, but the fan-out of each internal node equals to the *number of comparison-results* that this single I/O can generate among the items it reads from disk (i.e. B) and the items available in internal memory (i.e. $M - B$). These B items can be distributed in at most $\binom{M}{B}$ ways among the other $M - B$ items present in internal memory, so one I/O can generate no more than $\binom{M}{B}$ different results for those comparisons. But this is an incomplete answer because we are not considering the permutations among those items! However, some of these permutations have been already counted by some previous I/O, and thus we have not to recount them. These permutations are the ones concerning with items that have already passed through internal memory, and thus have been fetched by some previous I/O. So we have to count only the permutations among the *new* items, namely the ones

[2] There are two assumptions that are typically introduced in those arguments. One concerns with *item indivisibility*, so items cannot be broken up into pieces (hence hashing is not allowed!), and the other concerns with the possibility to *only move items* and not create/destroy/copy them, which actually implies that exactly one copy of each item does exist during their sorting or permuting.

that have never been considered by a previous I/O. We have n/B input pages, and thus n/B I/Os accessing new items. So these I/Os generate $\binom{M}{B}(B!)$ results by comparing those new B items with the $M - B$ ones in internal memory.

Let us now consider a computation with t I/Os, and thus a path in the comparison-tree with t nodes. n/B of those nodes must access the input items, which must be surely read to generate the final permutation. The other $t - \frac{n}{B}$ nodes read pages containing already processed items. Any root-to-leaf path has this form, so we can look at the comparison tree as having the new-I/Os at the top and the other nodes at its bottom. Hence if the tree has depth t, its number of leaves is at least $\binom{M}{B}^t \times (B!)^{n/B}$. By imposing that this number is $\geq n!$, and applying logarithms to both members, we derive that $t = \Omega(\frac{n}{B} \log_{M/B} \frac{n}{M})$. It is not difficult to extend this argument to the case of D disks thus obtaining the following.

THEOREM 4.3 *In a two-level memory model with internal memory of size M, disk-page size B and D disks, a comparison-based sorting algorithm must execute $\Omega(\frac{n}{DB} \log_{M/B} \frac{n}{DB})$ I/Os.*

It is interesting to observe that the number of available disks D does not appear in the denominator of the base of the logarithm, although it appears in the denominator of all other terms. If this would be the case, instead, D would somewhat penalize the sorting algorithms because it would reduce the logarithm's base. In the light of Theorem 4.1, multi-way Mergesort is I/O and time optimal on one disk, so D linearly boosts its performance thus having more disks is *linearly* advantageous (at least from a theoretical point of view). But Mergesort is no longer optimal on multi-disks because the simultaneous merging of $k > 2$ runs, should take $O(n/DB)$ I/Os in order to be optimal. This means that the algorithm should be able to fetch D pages per I/O, hence one per disk. This cannot be guaranteed, at every step, by the current merging-scheme because whichever is the distribution of the k runs among the D disks, and even if we know which are the next DB items to be loaded in the heap $\mathcal{H}$, it could be the case that more than B of these items reside on the same disk thus requiring more than one I/O from that disk, hence preventing the parallelism in the read operation.

In the following Section 4.4 we will address this issue by proposing the *disk striping* technique, that comes close to the I/O-optimal bound via a simple data layout on disks, and the *Greedsort* algorithm that achieves full optimality by devising an elegant and sophisticated merging scheme.

4.2.2 A lower-bound for Permuting

Let us assume that at any time the memory of our model, hence the internal memory of size M and the unbounded disk, contains a permutation of the input items possibly interspersed by empty cells. No more than n blocks will be non empty during the execution of the algorithm, because n steps (and thus I/Os) is an obvious upper bound to the I/O-complexity of Permuting (obtained by mimicking on disk the Permuting algorithm for the RAM model). We denote by P_t the number of permutations generated by an algorithm with t I/Os, where $t \leq n$ and $P_0 = 1$ since at the beginning we have the input order as initial permutation. In what follows we estimate P_t and then set $P_t \geq n!$ in order to derive the minimum number of steps t needed to realize any possible permutation given in input. Permuting is different from Sorting because the permutation to be realized is provided in input, and thus we do not need any computation. So in this case we distinguish three types of I/Os, which contribute differently to the number of generated permutations:

Write I/O: This may increase P_t by a factor $O(n)$ because we have at most $n+1$ possible ways to write the output page among the at most n not-empty pages available on disk. Any written page is "touched", and they are no more than n at any instant of the permuting process.

Read I/O on an untouched page: If the page was an input page never read before, the read operation imposes to account for the permutations among the read items, hence $B!$ in number, and to account also for the permutations that these B items can realize by distributing them among the $M - B$ items present in internal memory (similarly as done for Sorting). So this read I/O can increase P_t by a factor $O(\binom{M-B}{B}(B!)) = O(\binom{M}{B}(B!))$. The number of input (hence "untouched") pages is n/B. After a read I/O, they become "touched".

Read I/O on a touched page: If the page was already read or written, we already accounted in P_t for the permutations among its items, so this read I/O can increase P_t only by a factor $O(\binom{M}{B})$ due to the shuffling of the B read items with the $M - B$ ones present in internal memory. The number of touched pages is at most n.

If t_r is the number of reads and t_w is the number of writes executed by a Permuting algorithm, where $t = t_r + t_w$, then we can bound P_t as follows (here big-Oh have been dropped to ease the reading of the formulas):

$$P_t \leq (\frac{n}{B}\binom{M}{B}(B!))^{n/B} \times (n\binom{M}{B})^{t_r - n/B} \times n^{t_w} \leq (n\binom{M}{B})^t (B!)^{\frac{n}{B}}$$

In order to generate every possible permutation of the n input items, we need that $P_t \geq n!$. We can thus derive a lower bound on t by imposing that $n! \leq (n\binom{M}{B})^t \ (B!)^{\frac{n}{B}}$ and resolving with respect to t:

$$t = \Omega(\frac{n \log \frac{n}{B}}{B \log \frac{M}{B} + \log n})$$

We distinguish two cases. If $B \log \frac{M}{B} \leq \log n$, then the above equation becomes $t = \Omega(\frac{n \log \frac{n}{B}}{\log n}) = \Omega(n)$; otherwise it is $t = \Omega(\frac{n \log \frac{n}{B}}{B \log \frac{M}{B}}) = \Omega(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{M})$. As for sorting, it is not difficult to extend this proof to the case of D disks.

THEOREM 4.4 *In a two-level memory model with internal memory of size M, disk-page size B and D disks, permuting n items needs $\Omega(\min\{\frac{n}{D}, \frac{n}{DB} \log_{M/B} \frac{n}{DB}\})$ I/Os.*

Theorems 4.2–4.4 prove that the I/O-bounds provided in Table 4.1 for the Sorting and Permuting problems are optimal. Comparing these bounds we notice that they are asymptotically different whenever $B \log \frac{M}{B} < \log n$. Given the current values for B and M, respectively few KBs and few GBs, this inequality holds if $n = \Omega(2^B)$ and hence when n is more than Yottabytes ($= 2^{80}$). This is indeed an unreasonable situation to deal with one CPU and few disks. Probably in this context it would be more reasonable to use a *cloud* of PCs, and thus analyze the proposed algorithms via a *distributed* model of computation which takes into account many CPUs and more-than-2 memory levels. It is therefore not surprising that researchers typically assume `Sorting = Permuting` in the I/O-setting.

4.3 The distribution-based sorting paradigm

Like Mergesort, Quicksort is based on the divide&conquer paradigm, so it proceeds by dividing the array to be sorted into two pieces which are then sorted recursively. But unlike Mergesort, Quicksort does not explicitly allocate *extra*-working space, its *combine*-step is absent and its *divide*-step is sophisticated and impacts onto the overall efficiency of this sorting algorithm. Algorithm 4.3

reports the pseudocode of Quicksort, this will be used to comment on its complexity and argue for some optimizations or tricky issues which arise when implementing it over hierarchical memories.

Algorithm 4.3 The binary quick-sort: QuickSort(S, i, j)

```
1: if (i < j) then
2:     r = pick the position of a "good pivot";
3:     swap S[r] with S[i];
4:     p = Partition(S, i, j);
5:     QuickSort(S, i, p − 1);
6:     QuickSort(S, p + 1, j);
7: end if
```

The key idea is to partition the input array $S[i, j]$ in two pieces such that one contains items which are *smaller (or equal)* than the items contained in the latter piece. This partition is order preserving because no subsequent steps are necessary to recombine the ordered pieces after the two recursive calls. Partitioning is typically obtained by selecting one input item as a *pivot*, and by distributing all the other input items into two sub-arrays according to whether they are smaller/greater than the pivot. Items equal to the pivot can be stored anywhere. In the pseudocode the pivot is forced to occur in the first position $S[i]$ of the array to be sorted (steps 2–3): this is obtained by swapping the real pivot $S[r]$ with $S[i]$ before that procedure Partition(S, i, j) is invoked. We notice that step 2 does not detail the selection of the pivot, because this will be the topic of a subsequent section.

There are two issues for achieving efficiency in the execution of Quicksort: one concerns with the implementation of Partition(S, i, j), and the other one with the ratio between the size of the two formed pieces because the more *balanced* they are, the more Quicksort comes closer to Mergesort and thus to the optimal time complexity of $O(n \log n)$. In the case of a totally unbalanced partition, in which one piece is possibly empty (i.e. $p = i$ or $p = j$), the time complexity of Quicksort is $O(n^2)$, thus recalling in its cost the Insertion sort. Let us comment these two issues in detail in the following subsections.

4.3.1 From two- to three-way partitioning

The goal of Partition(S, i, j) is to divide the input array into two pieces, one contains items which are smaller than the pivot, and the other contains items which are larger than the pivot. Items equal to the pivot can be arbitrarily distributed among the two pieces. The input array is therefore permuted so that the smaller items are located before the pivot, which in turn precedes the larger items. At the end of Partition(S, i, j), the pivot is located at $S[p]$, the smaller items are stored in $S[i, p - 1]$, the larger items are stored in $S[p + 1, j]$. This partition can be implemented in many ways, taking $O(n)$ optimal time, but each of them offers a different cache usage and thus different performance in practice. We present below a tricky algorithm which actually implements a *three-way* distribution and takes into account the presence of items equal to the pivot. They are detected and stored aside in a "special" sub-array which is located between the two smaller/larger pieces. The following Figure **??** provides a graphical description of the three-way distribution.

It is clear that the central sub-array, which contains items equal to the pivot, can be discarded from the subsequent recursive calls, similarly as we discard the pivot. This reduces the number of items to be sorted recursively, but needs a change in the (classic) pseudo-code of Algorithm 4.3, because Partition must now return the pair of indices which delimit the central sub-array instead of just the position p of the pivot. The following Algorithm 4.4 details an implementation for the three-way

partitioning of $S[i, j]$ which uses three pointers that move rightward over this array and maintain the following invariant: P is the pivot driving the three-way distribution, $S[c]$ is the item currently compared against P, and $S[i, c-1]$ is the part of the input array already scanned and three-way partitioned in its elements. In particular $S[i, c-1]$ consists of three parts: $S[i, l-1]$ contains items smaller than P, $S[l, r-1]$ contains items equal to P, and $S[r, c-1]$ contains items larger than P. It may be the case that anyone of these sub-arrays is empty.

Algorithm 4.4 The three-way partitioning: PARTITION(S, i, j)

```
1:  P = S[i]; l = i; r = i − 1;
2:  for (c = r; c ≤ j; c++) do
3:      if (S[c] == P) then
4:          swap S[c] with S[r];
5:          r++;
6:      else if (S[c] < P) then
7:          swap S[c] with S[l];
8:          swap S[l] with S[r];
9:          r++; l++;
10:     end if
11: end for
12: return ⟨l, r − 1⟩;
```

Step 1 initializes P to the first item of the array to be partitioned (which is the pivot), l and r are set to guarantee that the smaller/greater pieces are empty, whereas the piece containing items equal to the pivot consists of the only item P. Next the algorithm scans $S[i+1, j]$ trying to maintain the invariant above. This is easy if $S[c] > P$, because it suffices to extend the part of the larger items by advancing r. In the other two cases (i.e. $S[c] \leq P$) we have to insert $S[c]$ in its correct position among the items of $S[i, r-1]$, in order to preserve the invariant on the three-way partition of $S[i, c]$. The cute idea is that this can be implemented in $O(1)$ time by means of at most two swaps, as described graphically in Figure 4.3.

The three-way partitioning algorithm takes $O(n)$ time and offers two positive properties: (i) stream-like access to the array S which allows the pre-fetching of the items to be read; (ii) the items equal to the pivot can then be eliminated from the following recursive calls.

4.3.2 Pivot selection

The selection of the pivot is crucial to get balanced partitions, reduce the number of recursive calls, and achieve optimal $O(n \log n)$ time complexity. The pseudo-code of Algorithm 4.3 does not detail the way the pivot is selected because this may occur in many different ways, each offering pros/cons. As an example, if we choose the pivot as the first item of the input array (namely $r = i$), the selection is fast but it is easy to instantiate the input array in order to induce un-balanced partitions: just take S to be an increasing or decreasing ordered sequence of items. Worse than this, it is the observation that any deterministic choice incurs in this drawback.

One way to circumvent bad inputs is to select the pivot *randomly* among the items in $S[i, j]$. This prevents the case that a given input is bad for Quicksort, but makes the behavior of the algorithm un-predictable in advance and dependant on the random selection of the pivot. We can show that the *average* time complexity is the optimal $O(n \log n)$, with an hidden constant small and equal to 1.45. This fact, together with the in-place nature of Quicksort, makes this approach much appealing

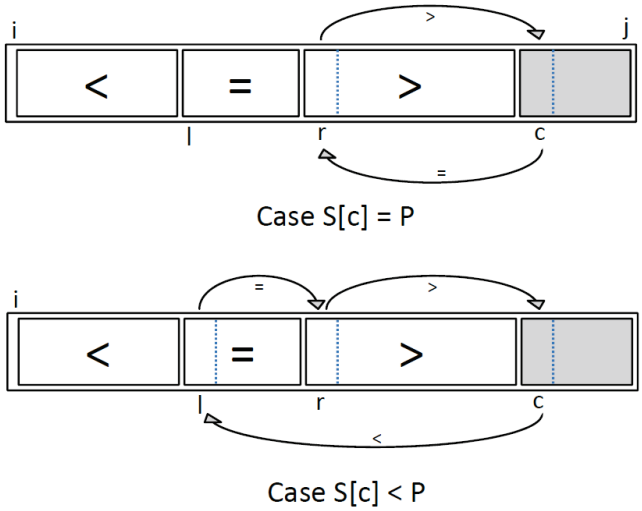

FIGURE 4.3: The two cases and the corresponding swapping. On the arrow we specify the value of the moved item with respect to the pivot.

in practice (cfr `qsort` below).

THEOREM 4.5 *The random selection of the pivot drives Quicksort to compare no more than* $1.45n \log n$ *items, on average.*

Proof The proof is deceptively simple if attacked from the correct angle. Let us denote by $X_{u,v}$ the random variable indicating whether items $S[u]$ and $S[v]$ are compared by Partition during a recursive call of Quicksort. Say $u < v$ and denote by $p_{u,v}$ the probability that this event occurs. The average number of comparisons executed by Quicksort can then be computed as $E[\sum_{u,v} X_{u,v}] = \sum_u \sum_{v>u} 1 \times p_{u,v} + 0 \times (1 - p_{u,v}) = \sum_{u=1}^{n} \sum_{v=u+1}^{n} p_{u,v}$ by linearity of expectation. To estimate $p_{u,v}$ we concentrate on the random choice of the pivot $S[r]$ and distinguish three cases. If $S[r] < S[u]$ or $S[r] > S[v]$, then the two items $S[u]$ and $S[v]$ are not compared to each other and they are passed to the same recursive call of Quicksort. So the problem presents itself again on a smaller subset of items containing both $S[u]$ and $S[v]$. In the case that $r = u$ or $r = v$, the two items are compared by Partition. In all other cases the pivot has a value between $S[u]$ and $S[v]$, so these two items go to two different partitions (hence two different recursive calls of Quicksort) and will never be compared. As a result, to compute $p_{u,v}$ we have to consider as interesting pivot-selections the ones for which $S[u] \leq S[r] \leq S[v]$. They are $v - u + 1$ in number, and among them only two choices induce a comparison between $S[u]$ and $S[v]$, namely the ones for which $r = u$ or $r = v$. So $p_{u,v} = 2/(v - u + 1)$. Substituting in the formula above we get:

$$E[\sum_{u,v} X_{u,v}] = \sum_{u=1}^{n} \sum_{v>u}^{n} \frac{2}{v-u+1} \leq 2 \sum_{u=1}^{n} \sum_{k=1}^{n} \frac{1}{k} \leq 2n \ln n$$

where the last inequality comes from the properties of the n-th harmonic number. The statement of the theorem then follows by observing that $\ln n \leq 1.45 \log_2 n$. ∎

The next question is how we can enforce the average behavior. The natural answer is to sample more than one pivot. Typically 3 pivots are randomly sampled from S and the central one (i.e. the median) is taken, thus requiring just two comparisons in $O(1)$ time. Taking more than 3 pivots makes the selection of a "good one" more robust, as proved in the following theorem [2].

THEOREM 4.6 *If Quicksort partitions around the median of* $2s+1$ *randomly selected elements, it sorts n distinct elements in* $\frac{2nH_n}{H_{2s+2}-H_{s+1}} + O(n)$ *expected comparisons, where* H_z *is the z-th harmonic number* $\sum_{i=1}^{z} \frac{1}{i}$.

By increasing s, we can push the expected number of comparisons close to $n \log n + O(n)$, however the selection of the median incurs a higher cost. In fact this can be implemented either by sorting the s samples in $O(s \log s)$ time and taking the one in the middle position $s+1$ of the ordered sequence; or in $O(s)$ worst-case time via a sophisticated algorithm (not detailed here). Randomization helps in simplifying the selection still guaranteeing $O(s)$ time on average. We detail this approach here because its analysis is elegant and its structure general enough to be applied not only for the selection of the median of an unordered sequence, but also for selecting the item of any rank k.

Algorithm 4.5 Selecting the k-th ranked item: RandSelect(S, k)

```
1:  r = pick a random item from S;
2:  S_< = items of S which are smaller than S[r];
3:  S_> = items of S which are larger than S[r];
4:  n_< = |S_<|;
5:  n_= = |S| − (|S_<| + |S_>|);
6:  if (k ≤ n_<) then
7:      return RandSelect(S_<, k);
8:  else if (k ≤ (n_< + n_=)) then
9:      return S[r];
10: else
11:     return RandSelect(S_>, k − n_< − n_=);
12: end if
```

Algorithm 4.5 is randomized and selects the item of the unordered S having rank k. It is interesting to see that the algorithmic scheme mimics the one used in the Partitioning phase of Quicksort: here the selected item $S[r]$ plays the same role of the pivot in Quicksort, because it is used to partition the input sequence S in three parts consisting of items smaller/equal/larger than $S[r]$. But unlike Quicksort, RandSelect recurses only in one of these three parts, namely the one containing the k-th ranked item. This part can be determined by just looking at the sizes of those parts, as done in Steps 6 and 8. There are two specific issues that deserve a comment. We do not need to recurse on $S_=$ because it consists of items equal to $S[r]$. If recursion occurs on $S_>$, we need to update the rank k because we are dropping from the original sequence the items belonging to the set $S_< \cup S_=$. Correctness is therefore immediate, so we are left with computing the average time complexity of this algorithm which turns to be the optimal $O(n)$, given that S is unsorted and thus all of its n items have to be examined to find the one having rank k among them.

THEOREM 4.7 *Selecting the k-th ranked item in an unordered sequence of size n takes* $O(n)$ *average time in the RAM model, and* $O(n/B)$ *I/Os in the two-level memory model.*

Proof Let us call "good selection" the one that induces a partition in which $n_<$ and $n_>$ are not larger than $2n/3$. We do not care of the size of $S_=$ since, if it contains the searched item, that item is returned immediately as $S[r]$. It is not difficult to observe that $S[r]$ must have rank in the range $[n/3, 2n/3]$ in order to ensure that $n_< \leq 2n/3$ and $n_> \leq 2n/3$. This occurs with probability $1/3$, given that $S[r]$ is drawn uniformly at random from S. So let us denote by $\hat{T}(n)$ the average time complexity of RandSelect when run on an array $S[1, n]$. We can write

$$\hat{T}(n) \leq O(n) + \frac{1}{3} \times \hat{T}(2n/3) + \frac{2}{3} \times \hat{T}(n),$$

where the first term accounts for the time complexity of Steps 2-5, the second term accounts for the average time complexity of a recursive call on a "good selection", and the third term is a crude upper bound to the average time complexity of a recursive call on a "bad selection" (that is actually assumed to recurse on the entire S again). This is not a classic recurrent relation because the term $\hat{T}(n)$ occurs on both sides; nevertheless, we observe that this term occurs with different constants in the front. Thus we can simplify the relation by subtracting those terms, so getting $\frac{1}{3}\hat{T}(n) \leq O(n) + \frac{1}{3}\hat{T}(2n/3)$, which gives $\hat{T}(n) = O(n) + \hat{T}(2n/3) = O(n)$. If this algorithm is executed in the two-level memory model, the equation becomes $\hat{T}(n) = O(n/B) + \hat{T}(2n/3) = O(n/B)$ given that the construction of the three subsets can be done via a single pass over the input items. ∎

We can use RandSelect in many different ways within Quicksort. For example, we can select the pivot as the median of the entire array S (setting $k = n/2$) or the median among an over-sampled set of $2s + 1$ pivots (setting $k = s + 1$, where $s \ll n/2$), or finally, it could be subtly used to select a pivot that generates a balanced partition in which the two parts have different sizes both being a fraction of n, say αn and $(1 - \alpha)n$ with $\alpha < 0.5$. This last choice $k = \lfloor \alpha n \rfloor$ seems meaningless because the three-way partitioning still takes $O(n)$ time but increases the number of recursive calls from $\log_2 n$ to $\log_{1-\alpha} n$. But this observation neglects the sophistication of modern CPUs which are parallel, pipelined and superscalar. These CPUs execute instructions in parallel, but if there is an event that impacts on the instruction flow, their parallelism is *broken* and the computation slows down significantly. Particularly slow are *branch mispredictions*, which occur in the execution of Partition(S, i, j) whenever an item smaller than or equal to the pivot is encountered. If we reduce these cases, then we reduce the number of branch-mispredictions, and thus deploy the full parallelism of modern CPUs. Thus the goal is to properly set α in a way that the reduced number of mispredictions balances the increased number of recursive calls. The right value for α is clearly architecture dependent, recent results have shown that a reasonable value is 0.1.

4.3.3 Bounding the extra-working space

QuickSort is frequently named as an *in-place* sorter because it does not use extra-space for ordering the array S. This is true if we limit ourself to the pseudocode of Algorithm 4.3, but it is no longer true if we consider the cost of managing the recursive calls. In fact, at each recursive call, the OS must allocate space to save the local variables of the caller, in order to retrieve them whenever the recursive call ends. Each recursive call has a space cost of $\Theta(1)$ which has to be multiplied by the number of nested calls Quicksort can issue on an array $S[1, n]$. This number can be $\Omega(n)$ in the worst case, thus making the extra-working space $\Theta(n)$ on some bad inputs (such as the already sorted ones, pointed out above).

We can circumvent this behavior by restructuring the pseudocode of Algorithm 4.3 as specified in Algorithm 4.6. This algorithm is cryptic at a first glance, but the underlying design principle is pretty smart and elegant. First of all we notice that the while-body is executed only if the input array is longer than n_0, otherwise Insertion-sort is called in Step 13, thus deploying the well-known efficiency of this sorter over very small sequences. The value of n_0 is typically chosen of few tens

Algorithm 4.6 The binary quick-sort with bounded recursive-depth: BoundedQS(S, i, j)

```
1:  while (j − i > n_0) do
2:      r = pick the position of a "good pivot";
3:      swap S[r] with S[i];
4:      p = Partition(S, i, j);
5:      if (p ≤ (i+j)/2) then
6:          BoundedQS(S, i, p − 1);
7:          i = p + 1;
8:      else
9:          BoundedQS(S, p + 1, j);
10:         j = p − 1;
11:     end if
12: end while
13: InsertionSort(S, i, j);
```

of items. If the input array is longer than n_0, a modified version of the classic binary Quicksort is executed that mixes one single recursive call with an iterative while-loop. The ratio underlying this code re-factoring is that the correctness of classic Quicksort does not depend on the order of the two recursive calls, so we can reshuffle them in such a way that the first call is always executed on the smaller part of the two/three-way partition. This is exactly what the IF-statement in step 5 guarantees. In addition to that, the pseudo-code above drops the recursive call onto the larger part of the partition in favor of another execution of the body of the while loop in which we properly changed the parameters i and j to reflect the new extremes of that larger part. This "change" is well-known in the literature of compilers with the name of *elimination of tail recursion*. The net result is that the recursive call is executed on a sub-array whose size is no more than the half of the input array. This guarantees an upper bound of $O(\log_2 n)$ on the number of recursive calls, and thus on the size of the extra-space needed to manage them.

THEOREM 4.8 BoundedQS *sorts n atomic items in the RAM model taking $O(n \log n)$ average time, and using $O(\log n)$ additional working space.*

We conclude this section by observing that the C89 and C99 ANSI standards define a sorting algorithm, called `qsort`, whose implementation encapsulates most of the algorithmic tricks detailed above.[3] This witnesses further the efficiency of the distribution-based sorting scheme over the 2-levels: cache and DRAM.

4.3.4 From binary to multi-way Quicksort

Distribution-based sorting is the *dual* of merge-based sorting in that the first proceeds by splitting sequences according to pivots and then ordering them recursively, while the latter merges sequences which have been ordered recursively. Disk-efficiency was obtained in Multi-way Mergesort by managing (fusing) multiple sequences together. The same idea is applied to design the Multi-way

[3] Actually `qsort` is based on a different two-way partitioning scheme that uses two iterators, one moves forward and the other one moves backward over S; a swap occurs whenever two un-sorted items are encountered. The asymptotic time complexity does not change, but practical efficiency can spur from the fact that the number of swaps is reduced since equal items are not moved.

Quicksort which splits the input sequence into $k = \Theta(M/B)$ sub-sequences by using $k-1$ pivots. Given that $k \gg 1$ the selection of those pivots is not a trivial task because it must ensure that the k partitions they form, are *balanced* and thus contain $\Theta(n/k)$ items each. Section 4.3.2 discussed the difficulties underlying the selection of one pivot, so the case of selecting many pivots is even more involved and needs a sophisticated analysis.

We start with denoting by $s_1, \ldots, s_{k-1}$ the pivots used by the algorithm to split the input sequence $S[1,n]$ in k parts, also called *buckets*. For the sake of clarity we introduce two dummy pivots $s_0 = -\infty$ and $s_k = +\infty$, and denote the i-th bucket by $B_i = \{S[j] : \ s_{i-1} < S[j] \leq s_i\}$. We wish to guarantee that $|B_i| = \Theta(n/k)$ for all the k buckets. This would ensure that $\log_k \frac{n}{M}$ partitioning phases are enough to get sub-sequences shorter than M, which can thus be sorted in internal-memory without any further I/Os. Each partitioning phase can be implemented in $O(n/B)$ I/Os by using a memory organization which is the dual of the one employed for Mergesort: namely, 1 input block (used to read from the input sequence to be partitioned) and k output blocks (used to write into the k partitions under formation). By imposing $k = \Theta(M/B)$, we derive that the number of partitioning phases is $\log_k \frac{n}{M} = \Theta(\log_{M/B} \frac{n}{M})$ so that the Multi-way Quicksort takes the optimal I/O-bound of $\Theta(\frac{n}{B} \log_{M/B} \frac{n}{M})$, provided that each partitioning step *distributes evenly* the input items among the k buckets.

To find efficiently k good pivots, we deploy a fast and simple randomized strategy based on *oversampling*, whose pseudocode is given in Algorithm 4.7 below. Parameter $a \geq 0$ controls the amount of oversampling and thus impacts onto the robustness of the selection process as well as on the cost of Step 2. The latter cost is $O((ak)\log(ak))$ if we adopt an optimal in-memory sorter, such as Heapsort or Mergesort, to sort the $\Theta(ak)$ sampled items.

Algorithm 4.7 Selection of $k-1$ good pivots via oversampling

1: Take $(a+1)k-1$ samples at random from the input sequence;
2: Sort them into an ordered sequence A;
3: For $i = 1, \ldots, k-1$, pick the pivot $s_i = A[(a+1)i]$;
4: **return** the pivots s_i;

The main idea is to select $\Theta(ak)$ candidate pivots from the input sequence and then pick k among them, namely the ones which are evenly spaced and thus $(a+1)$ far apart from each other. We are arguing that those $\Theta(ak)$ samples provide a faithful picture of the distribution of the items in the entire input sequence, so that the balanced selection $s_i = A[(a+1)i]$ should provide us with "good pivots". The larger is a the closer to $\Theta(n/k)$ should be the size of all buckets, but the higher would be the cost of sorting the samples. At the extreme case of $a = n/k$, the samples could not be sorted in internal memory! On the other hand, the closer a is to zero the faster would be the pivot selection but more probable is to get unbalanced partitions. As we will see in the following Lemma 4.1, choosing $a = \Theta(\log k)$ is enough to obtain balanced partitions with a pivot-selection cost of $O(k \log^2 k)$ time. We notice that the buckets will be not perfectly balanced but quasi-balanced, since they include no more than $\frac{4n}{k} = O(n/k)$ items; the factor 4 will nonetheless leave unchanged the aimed asymptotic time complexity.

LEMMA 4.1 Let $k \geq 2$ and $a + 1 = 12 \ln k$. A sample of size $(a+1)k-1$ suffices to ensure that all buckets receives less than $4n/k$ elements, with probability at least $1/2$.

Proof We provide an upper bound of $1/2$ to the probability of the complement event stated in

the Lemma, namely that there exists one bucket whose size is larger than $4n/k$. This corresponds to a *failure* sampling, which induces an un-balanced partition. To get this probability estimate we will introduce a cascade of events that are implied by this one and thus have larger and larger probabilities to occur. For the last one in the sequence we will be able to fix an explicit upper-bound of $1/2$. Given the implications, this upper bound will also hold for the original event. And so we will be done.

Let us start by considering the sorted version of the input sequence S, which hereafter we denote by S'. We logically split S' in $k/2$ segments of length $2n/k$ each. The event we are interested in is that there exists a bucket B_i with at least $4n/k$ items assigned to it. As illustrated in Figure 4.4 this large bucket completely spans at least one segment, say t_2 in the Figure below, because the former contains $\geq 4n/k$ items whereas the latter contains $2n/k$ items.

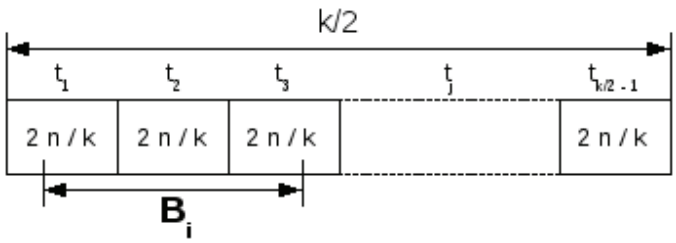

FIGURE 4.4: Splitting of the sorted sequence S' into segments.

By definition of the buckets, the pivots s_{i-1} and s_i which delimit B_i fall outside t_2. Hence, by Algorithm 4.7, less that $(a+1)$ samples fall in the segment overlapped by B_i. In the figure it is t_2, but it might be any segment of S'. So we have that:

$$\begin{aligned} \mathcal{P}(\exists B_i : |B_i| \geq 4n/k) &\leq \mathcal{P}(\exists t_j : t_j \text{ contains } < (a+1) \text{ samples}) \\ &\leq \frac{k}{2} \times \mathcal{P}(\text{a specific segment contains } < (a+1) \text{ samples}) \end{aligned} \quad (4.1)$$

where the last inequality comes from the *union bound*, given that $k/2$ is the number of segments constituting S'. So we will hereafter concentrate on providing an upper bound to the last term. The probability that one sample ends in a given segment is equal to $\frac{(2n/k)}{n} = \frac{2}{k}$ because they are assumed to be drawn uniformly at random from S (and thus from S'). So let us call X the number of those samples, we are interested in computing $\mathcal{P}(X < a+1)$. We start by observing that $E[X] = ((a+1)k-1) \times \frac{2}{k} = 2(a+1) - \frac{2}{k}$. The Lemma assumes that $k \geq 2$, so $E[X] \geq 2(a+1) - 1$ which is $\geq \frac{3}{2}(a+1)$ for all $a \geq 1$.

Since we are interested in $\mathcal{P}(X < a+1)$, we resort the Chernoff bound:

$$\mathcal{P}(X < (1-\delta)E[X]) \leq e^{-\frac{\delta^2}{2} E[X]}$$

From above we know that $E[X] \geq \frac{3}{2}(a+1)$, and thus $a+1 \leq (2/3)E[X] = (1-\frac{1}{3})E[X]$. As a result, we can write

$$\begin{aligned} \mathcal{P}(X < a+1) &\leq \mathcal{P}(X < (1-\frac{1}{3})E[X]) \\ &\leq e^{-E[X]/18} \leq e^{-(a+1)/12} = e^{-\ln k} = \frac{1}{k} \end{aligned} \quad (4.2)$$

where we used the lemma's assumption that $a + 1 = 12 \ln k$. By plugging this value in Eqn 4.1, we get the statement of the Lemma. ∎

4.4 Sorting with multi-disks$^{\infty}$

The bottleneck in disk-based sorting is obviously the time needed to perform an I/O operation. In order to mitigate this problem, we can use D disks working in parallel so to transfer DB items per I/O. On the one hand this increases the bandwidth of the I/O subsystem, but on the other hand, it makes the design of I/O-efficient algorithms particularly difficult. Let's see why.

The simplest approach to manage parallel disks is called *disk striping* and consists of looking at the D disks as *one single* disk whose page size is $B' = DB$. This way we gain simplicity in algorithm design by just using *as-is* any algorithm designed for one disk, now with a disk-page of size B'. Unfortunately, this simple approach pays an un-negligible price in terms of I/O-complexity:

$$O(\frac{n}{B'} \log_{M/B'} \frac{n}{M}) = O(\frac{n}{DB} \log_{M/DB} \frac{n}{M})$$

This bound is not optimal because the base of the logarithm is D times smaller than what indicated by the lower bound proved in Theorem 4.3. The ratio between the bound achieved via disk-striping and the optimal bound is $1 - \log_{M/B} D$, which shows disk striping to be less and less efficient as the number of disks increases $D \longrightarrow M/B$. The problem resides in the fact that we are not deploying the *independency* among disks by using them as a monolithic sub-system.

On the other hand, deploying this independency is tricky and it took several years before designing fully-optimal algorithms running over multi-disks and achieving the bounds stated in Theorem 4.3. The key problem with the management of multi-disks is to guarantee that every time we access the disk sub-system, we are able to read or write D pages each one coming from or going to a different disk. This is to guarantee a throughput of DB items per I/O. In the case of sorting, such a difficulty arises both in the case of distributed-based and merge-based sorters, each with its specialties given the duality of those approaches.

Let us consider the multi-way Quicksort. In order to guarantee a D-way throughput in reading the input items, these must be distributed evenly among the D disks. For example they could be striped circularly as indicated in Figure 4.5. This would ensure that a scan of the input items takes $O(n/DB)$ optimal I/Os.

	Block 1	Block 2	Block 3	Block 4	Block 5	
Disk 1	1 2	9 10	17 18	25 26	33 34	...
Disk 2	3 4	11 12	19 20	27 28	35 36	...
Disk 3	5 6	13 14	21 22	29 30	37 38	...
Disk 4	7 8	15 16	23 24	31 32	39 40	...

FIGURE 4.5: An example of striping a sequence of items among $D = 4$ disks, with $B = 2$.

This way the subsequent distribution phase can read the input sequence at that I/O-speed. Nonetheless problems occur when writing the output sub-sequences produced by the partitioning process. In fact that writing should guarantee that each of these sub-sequences is circularly striped among the disks in order to maintain the invariant for the next distribution phase (to be executed independently over those sub-sequences). In the case of D disks, we have D output blocks that are filled by the partitioning phase. So when they are full these D blocks must be written to D distinct disks to ensure full I/O-parallelism, and thus one I/O. Given the striping of the runs, if all these output blocks belong to the same run, then they can be written in one I/O. But, in general, they belong to different runs so that conflicts may arise in the writing process because blocks of different runs could have to be written onto the same disks. An example is given in Figure 4.6 where we have illustrated a situation in which we have three runs under formation by the partitioning phase of Quicksort, and three disks. Runs are striped circularly among the 3 disks and shadowed blocks correspond to the prefixes of the runs that have been already written on those disks. Arrows point to the next free-blocks of each run where the partitioning phase of Quicksort can append the next distributed items. The figure depicts an extremely bad situation in which all these blocks are located on the same disk D_2, so that an I/O-conflict may arise if the next items to be output by the partitioning phase go to these runs. This practically means that the I/O-subsystem must *serialize* the write operation in $D = 3$ distinct I/Os, hence loosing all the I/O-parallelism of the D-disks. In order to avoid these difficulties, there are known *randomized* solutions that ensure optimal I/Os in the average case [6].

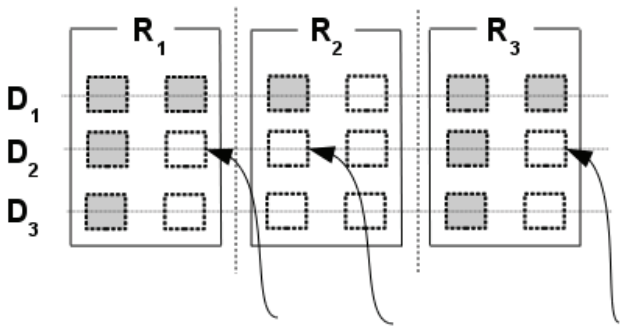

FIGURE 4.6: An example of an I/O-conflict in writing $D = 3$ blocks belonging to 3 distinct runs.

In what follows we sketch a deterministic multi-disk sorter, known as Greed Sort [5], which solves the difficulties above via an elegant merge-based approach which consists of two stages: first, items are *approximately* sorted via an I/O-efficient Multi-way Merger that deals with $R = \Theta(\sqrt{M/B})$ sorted runs in an independent way (thus deploying disks in parallel), and then it *completes* the sorting of the input sequence by using an algorithm (aka ColumnSort, due to T. Leighton in 1985) that takes a linear number of I/Os when executed over *short* sequences of length $O(M^{\frac{3}{2}})$. Correctness comes from the fact that the distance of the un-sorted items from their correct sorted position, after the first stage, is smaller than the size of the sequences manageable by Columsort. Hence the second stage can correctly turn the approximately-sorted sequence into a totally-sorted sequence by a single pass.

How to get the approximately sorted runs in I/O-efficient way is the elegant algorithmic contribution of GreedSort. We sketch its main ideas here, and refer the interested reader to the corresponding paper [5] for further details. We assume that sorted runs are stored in a striped way among the D disks, so that reading D consecutive blocks from each of them takes one I/O. As we discussed for

Quicksort, also in this Merge-based approach we could incur in I/O-conflicts when reading these runs. GreedSort avoids this problem by operating independently on each disk: in a parallel read operation, GreedSort fetches the two *best* available blocks from each disk. These two blocks are called "best" because they contain the *smallest minimum item*, say m_1, and the *smallest maximum item*, say m_2, currently present in blocks stored on that disk (possibly these two blocks are the same). It is evident that this selection can proceed independently over the D disks, and it needs a proper data structure that keeps track of minimum/maximum items in disk-blocks. Actually [5] shows that this data structure can fit in internal memory, thus not incurring any further I/Os for this selection operations.

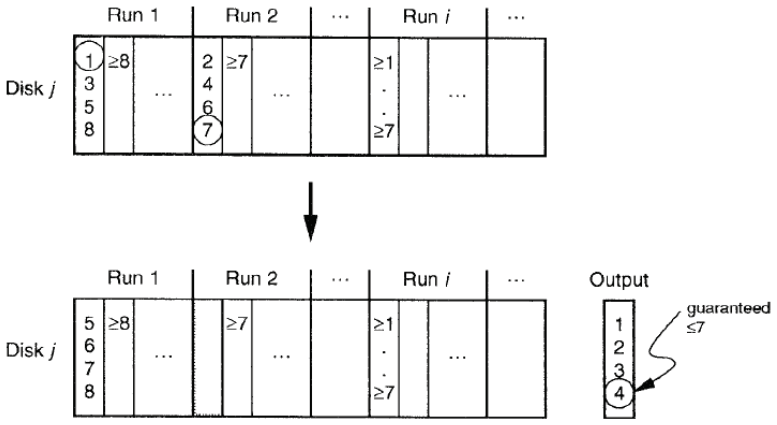

FIGURE 4.7: Example taken from the GreedSort's paper [5].

Figure 4.7 shows an example on disk j, which contains the blocks of several runs because of the striping-based storage. The figure assumes that run 1 contains the block with the smallest minimum item (i.e. 1) and run 2 contains the block with the smallest maximum item (i.e. 7). All the other blocks which come from run 1 contain items larger than 8 (i.e. the maximum of the first block), and all the other blocks which come from run 2 contain items larger than 7. All blocks coming form other runs have minimum larger than 1 and maximum larger than 7. Greedsort then merges these blocks creating two new sorted blocks: the first one is written to output (it contains the items $\{1, 2, 3, 4\}$), the second one is written back to the run of the smallest minimum m_1, namely run 1 (it contains the items $\{5, 6, 7, 8\}$). This last write back into run 1 does not disrupt that ordered subsequence, because the second block contains surely items smaller than the maximum of the block of m_1.

We notice that the items written in output are not necessarily the four smallest items of all runs. In fact it could exist a block in another run (different from runs 1 and 2) which contains a value within $[1, 4]$, say 2.5, and whose minimum is larger than 1 and whose maximum is larger than 7. So this block is compatible with the selection we did above from run 1 and 2, but it contains items that should be stored in the first block of the sorted sequence. So the selection of the "two-best blocks" proceeds independently over all disks until all runs have been examined and written in output. The final sequence produced by this merging process is *not* sorted, but if we read it in a striped-way along all D disks, then it results *approximately* sorted as stated in the following lemma (proved in [5]).

LEMMA 4.2 A sequence is called *L-regressive* if any pair of un-sorted records, say $\ldots y \ldots x \ldots$ with $y > x$, has distance less than L in the sequence. The previous sorting algorithm creates an output that is L-regressive, with $L = RDB = D\sqrt{MB}$.

The application of ColumnSort over the L-regressive sequence, by sliding a window of $2L$ items which moves L steps forward at each phase, allows to produce a merged sequence which is totally sorted. In fact $L = D\sqrt{MB} \leq DB\sqrt{M} \leq M^{3/2}$ and thus ColumnSort is effective in producing the entirely sorted sequence. We notice that at this point this sorted sequence is striped along all D disks, thus the invariant for the next merging phase is preserved and the merge can thus start over a number of runs that has been reduced by a factor R. The net result is that each merging takes $O(n/DB)$ I/Os, the total number of merging stages is $\log_R \frac{n}{M} = O(\log_{M/B} \frac{n}{M})$, and thus the optimal I/O-bound follows.

References

[1] Alok Aggarwal and Jeffrey S. Vitter. The Input/Output complexity of Sorting and Related Problems. *Communication of the ACM*, 31(9): 1116-1127, 1988.

[2] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Procs of the 8th ACM-SIAM Symposium on Discrete Algorithms*, 360–369, 1997.

[3] Donald E. Knuth. *The Art of Computer Programming: volume 3*. Addison-Wesley, 2nd Edition, 1998.

[4] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The basic toolbox.* Springer, 2009.

[5] Mark H. Nodine and Jeffrey S. Vitter. Greed Sort: Optimal Deterministic Sorting on Parallel Disks. *Journal of the ACM*, 42(4): 919-933, 1995.

[6] Jeffrey S. Vitter. External memory algorithms and data structures. *ACM Computing Surveys*, 33(2):209–271, 2001.