ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo
Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Notes 8.10.
Assistances and comments will be acknowledged.

8.10 External Sorting

Suppose we are sorting n entries that are stored in secondary memory and we are unable to load all of the entries into main memory. We will look at a strategy that can be used to sort those entries efficiently.

Recall that secondary memory such as hard drives and tape drives are block-addressable. If necessary, we will assume that a block size is 4 KiB. Let us assume that we can store b entries per block.

8.10.1 Naming Conventions

To begin, we will use lower-case letters to denote variables that store a number of entries and will use upper-case letters to denote variables that store the size of larger structures. For your reference, we include Table 1.

Table 1. Variables used in this topic.

Variable	Description
n	The total number of entries
b	The number of entries per block
M	The number of blocks that can be loaded into main memory at one time
m	The number of entries that can be loaded into main memory at one time: $m = Mb$
N	The total number of sections of M blocks: $n = Nm = NMb$

8.10.2 Strategy

The first step is to load as many blocks into main memory at once. Suppose we can load a section of M blocks into main memory, as is shown in Figure 1. This means we are loading $m = Mb$ entries into main memory at once. For example, if allocated 2 GiB of memory for loading blocks into main memory, we could load $M = 512$ blocks. Let us assume that there are a total of N sections of M blocks.

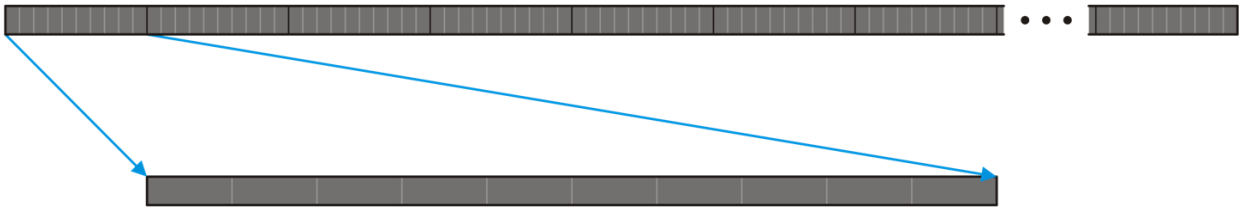

Figure 1. Loading M blocks into main memory with N sections of M blocks in secondary memory.

We can use an in-place sorting algorithm such as quick-sort to sort these $m = Mb$ entries in $\Theta(m \ln(m))$ time and then write the result back into secondary memory, as is shown in Figure 2.

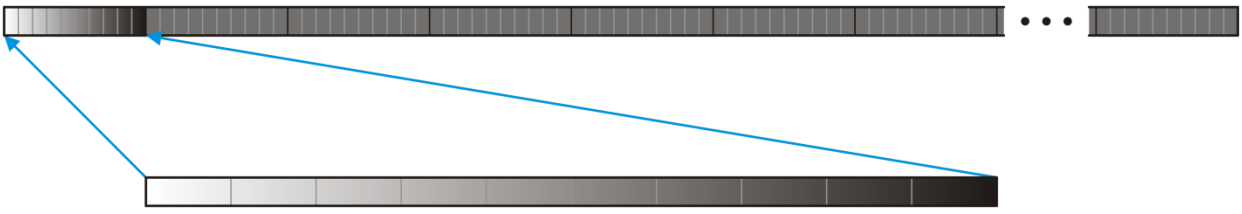

Figure 2. Copying the sorted M blocks back into secondary memory.

If we repeat this process for all N blocks, as is shown in Figure 3, the run time will be

$$\Theta(Nm \ln(m)) = \Theta(n \ln(m)).$$

Figure 3. The result after all N sections are individually sorted.

Next, we will merge these N blocks into a single contiguous section. Because we must access all of the N sections while merging (as the next largest entry could be in any of them), it is necessary to have at least some component of each section in main memory. To do this, we will allocate memory for one block-sized input buffer for every section. We will then load the first block of each of the N sections into the corresponding input buffer, as is shown in Figure 4.

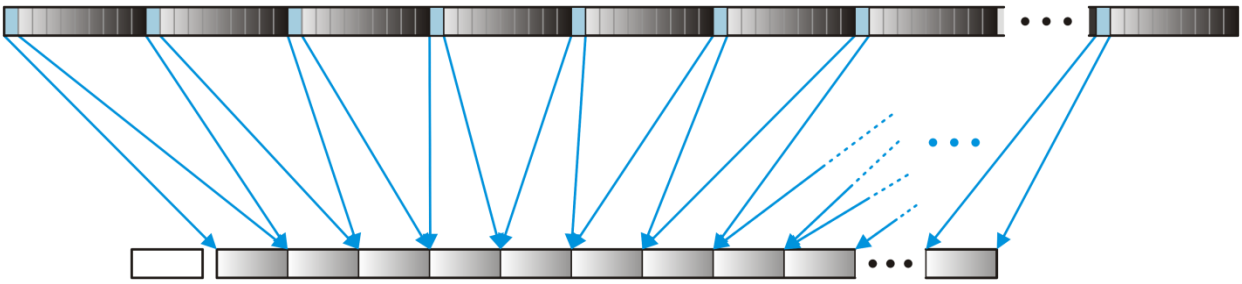

Figure 4. Loading the first block from each section into the N input buffers.

Now, suppose we have allocated 2 GiB of main memory for the process of sorting arrays. This means that each section could contain 512 blocks or 2 GiB of entries to be sorted. At this point, if we load the first block of each section into main memory, we can load a block from up to 512 × 1024 sections. Thus, this is a reasonable scheme so long as we are sorting 1 EiB of data or less. For larger data sets, we could either access more main memory (probably requiring a 64-bit computer) or load only one part of each block (loading only the first 256 bytes would allow us to sort up to 16 TiB of data).

Now we perform an N-way merge of these N blocks of sorted entries. For this, we would have to allocate an additional block-sized output buffer (*e.g.*, 4 KiB) of main memory . This merging process is shown in Figure 5.

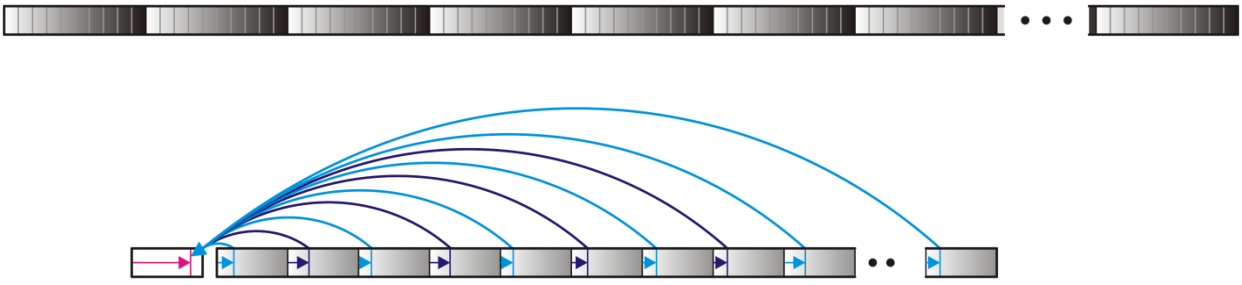

Figure 5. Performing an N-way merge of the N blocks in main memory.

Once the output buffer is full, we can write it to secondary memory and empty it, as is shown in Figure 6.

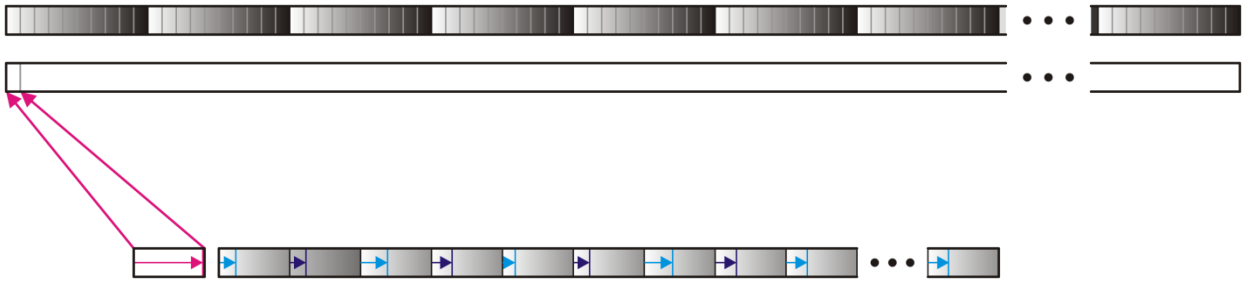

Figure 6. Writing the first merged block in the output buffer to secondary memory. After this, we reuse the buffer and continue writing the output buffer to secondary memory every time it is filled.

The merging process requires that we allocate new memory for the merged entries and therefore this form of external sorting will require $\Theta(n)$ additional memory.

Similarly, when one of the N input buffers has been emptied by having copied all of its entries in the merge process, we would copy the next block in the corresponding section from secondary memory into main memory. Figure 7 shows the second block in the forth section being copied into main memory.

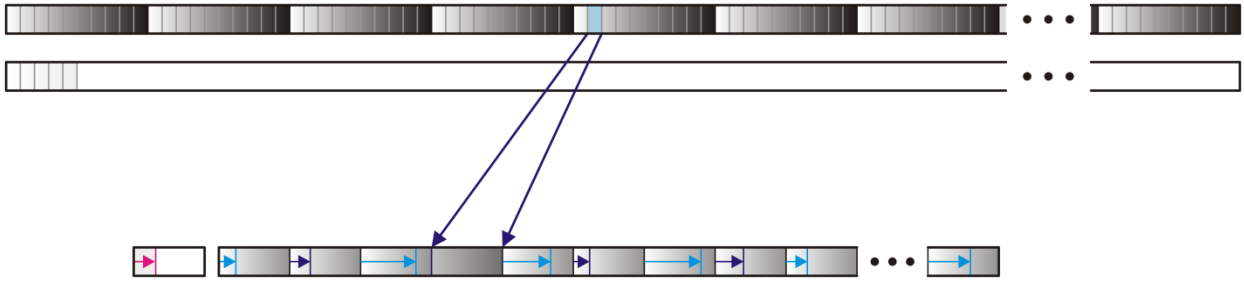

Figure 7. Copying the 2nd block from the 4th section into its corresponding input buffer in main memory.

We continue the merging process until all N blocks have been merged and copied into secondary memory, as is shown in Figure 8.

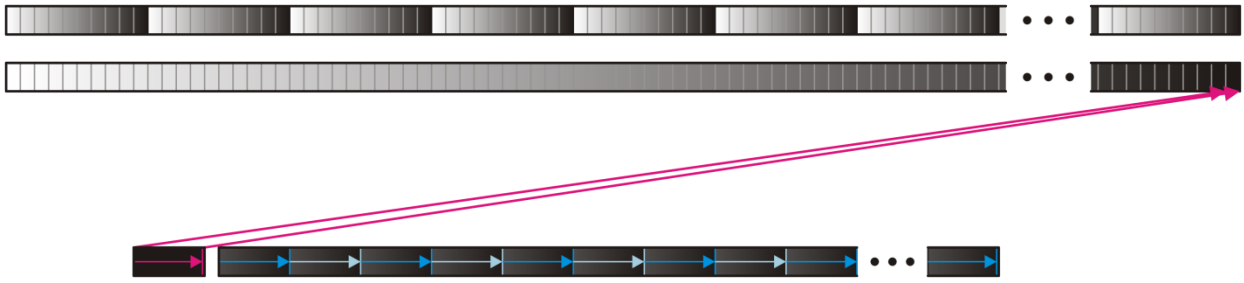

Figure 8. Copying the last merged block into secondary memory.

At this, the MN blocks copied into secondary memory are now a sorted variation of the first block.

8.10.3 Run-time Analysis

In the topic on merge sort, we saw that we could merge two lists of size n_1 and n_2 in $\Theta(n_1 + n_2)$ time and thus this suggests that merging can be performed in linear time. Now, the time required to perform all of the applications of quick sort was (from above) $n \ln(m)$ where m is the number of items in each of the sections. If m is sufficiently small and fixed, this suggests that this process of applying quick sort to N sections can run in essentially $\Theta(n)$ time. If merging is also linear, the entire process is consequently linear.

This, however, cannot be true: any sorting algorithm that involves comparing different entries must run in $\Omega(n \ln(n))$ time. The fallacy lies in the merging process. In order to efficiently merge N different lists, the best we can do is to use a binary min-heap to determine which comes next. Each time we remove the minimum element, we insert a new entry from the same block from which that minimum element originated. Thus, the run time to insert and remove n objects into a binary min-heap containing at N entries is $\Theta(n \ln(N))$.

We may therefore conclude that this process has a run time of

$$\Theta(n \ln(m) + n \ln(N)) = \Theta(n (\ln(m) + \ln(N))) = \Theta(n \ln(mN)) = \Theta(n \ln(n))$$

as $n = mN$. Thus, the run time of this external sorting routine must still be $\Theta(n \ln(n))$, as expected.

8.10.4 Additional Remarks

$$1\,\text{MiB} \cdot \left(\frac{1\,\text{MiB}}{4\,\text{KiB}} \right)^K = 1\,\text{MiB} \cdot \left(\frac{2^{20}}{2^{12}} \right)^K = 2^{8K}\,\text{MiB}$$

With 4 KiB blocks and 2 GiB of main memory, we determined that we could sort up to 1 EiB of data. Suppose we had to sort more data than this, or (more likely), we have less available memory, say 1 MiB. Now, we can only sort up to

$$1\,\text{MiB} \cdot \frac{1\,\text{MiB}}{4\,\text{KiB}} = 1\,\text{MiB} \cdot \frac{2^{20}}{2^{12}} = 1\,\text{MiB} \cdot 2^8 = 256\,\text{MiB}$$

of data. If we had more data than this, you could divide it into 256 MiB chunks, sort each of these using external sorting, and then repeat the merging algorithm described above. We could repeat this process K times, and at each step, we are limited by the number 4 KiB blocks we can load into main memory, allowing us to sort

$$1\,\text{MiB} \cdot \left(\frac{1\,\text{MiB}}{4\,\text{KiB}} \right)^K = 2^{8K}\,\text{MiB}$$

of data.

There are other techniques that can be used to speed up the process of external sorting by, for example, performing the application of quick sort on the sections in parallel, storing at least a portion of the data in solid-state drives, and using larger input and output buffers. Using larger buffers will be significantly beneficial only if the data is being stored contiguously in secondary memory.