Technische Universiteit
Eindhoven
University of Technology

Course Notes on

I/O-Efficient Algorithms

Mark de Berg

Contents

Chapter 1

Introduction to I/O-Efficient Algorithms

Using data from satellites or techniques such as LIDAR (light detection and ranging) it is now possible to generate highly accurate digital elevation models of the earth's surface. The simplest and most popular digital elevation model (DEM) is a grid of square cells, where we store for each grid cell the elevation of the center of the cell. In other words, the digital elevation model is simply a 2-dimensional array A, where each entry $A[i, j]$ stores the elevation of the corresponding grid cell. As mentioned, DEMs are highly accurate nowadays, and resolutions of 1 m or less are not uncommon. This gives massive data sets. As an example, consider a DEM representation an area of 100 km $\times$ 100 km at 1 m resolution. This gives an array $A[0..m-1, 0..m-1]$ where $m = 100,000$. If we use 8 bytes per grid cell to store its elevation, the array needs about 80GB.

Now suppose we wish to perform a simple task, such as computing the average elevation in the terrain. This is of course trivial to do: go over the array A row by row to compute the sum[1] of all entries, and divide the result by m^2 (the total number of entries).

ComputeAverage-RowByRow(A)
$\triangleright$ A is an $m \times m$ array
1: $s \leftarrow 0$
2: **for** $i \leftarrow 0$ **to** $m-1$ **do**
3: **for** $j \leftarrow 0$ **to** $m-1$ **do**
4: $s \leftarrow s + A[i, j]$
5: **end for**
6: **end for**
7: **return** s/m^2

Alternatively we could go over the array column by column, by exchanging the two **for**-loops. Doesn't matter, right? Wrong. You may well find out that one of the two algorithms is much slower than the other. How is this possible? After all, both algorithms run in $O(n)$ time, where $n := m^2$ denotes the total size of the array, and seem equivalent. The problem is that the array A we are working on does not fit into the internal memory. As a result, the actual running time is mainly determined by the time needed to fetch the data from the hard disk,

[1] Actually, things may not be as easy as they seem because adding up a very large number of values may lead to precision problems. We will ignore this, as it is not an I/O-issue.

not by the time needed for CPU computations. In other words, the time is determined by the number of I/O-operations the algorithm performs. And as we shall see, a small difference such as reading a 2-dimensional array row by row or column by column can have a big impact on the number of I/O-operations. Thus it is important to make algorithms I/O-*efficient* when the data on which they operate does not fit into internal memory. In this part of the course we will study the theoretical basics of I/O-efficient algorithms.

1.1 The I/O-model

To be able to analyze the I/O-behavior of our algorithms we need an abstract model of the memory of our computer. In the model we assume our computer is equipped with two types of memory: an *internal memory* (the main memory) of size M and a *external memory* (the disk) of unlimited size. Unless stated otherwise, the memory size M refers to the number of basic elements—numbers, pointers, etcetera—that can be stored in internal memory. We are interested in scenarios where the input does not fit into internal memory, that is, where the input size is greater than M. We assume that initially the input resides entirely in external memory.

An algorithm can only perform an operation on a data element—reading a value stored in a variable, changing the value of a variable, following a pointer, etc.—when the data element is available in internal memory. If this is not the case, the data should first be *fetched* from external memory. For example, in line 4 of *ComputeAverage-RowByRow* the array element $A[i, j]$ may have to be fetched from external memory before s can be updated. (In fact, in theory the variables s, i, j may have to be fetched from external memory as well, although any reasonable operating system would ensure that these are kept in internal memory.)

I/O-operations (fetching data from disk, or writing data to disk) are slow compared to CPU-operations (additions, comparisons, assignments, and so on). More precisely, they are very, very slow: a single I/O-operation can be 100,000 times or more slower than a single CPU-operation. This is what makes I/O-behavior often the bottleneck for algorithms working on data stored on disk. The main reason that reading to (or writing from) disk is so slow, is that we first have to wait until the read/write head is positioned at the location on the disk where the data element is stored. This involves waiting for the head to move to the correct track on the disk (*seek time*), and waiting until the disk has rotated such that the data to be read is directly under the head (*rotational delay*). Once the head and disk are positioned correctly, we can start reading or writing. Note that if we want to read a large chunk of data stored consecutively on disk, we have to pay the seek time and rotational delay only once, leading to a much better average I/O-time per data element. Therefore data transfer between disk and main memory is not performed on individual elements but on *blocks* of consecutive elements: When you read a single data element from disk, you actually get an entire block of data—whether you like it or not. To make an algorithm I/O-efficient, you want to make sure that the extra data that you get are actually useful. In other words: you want your algorithm to exhibit *spatial locality*: when an element is needed by the algorithm, elements from the same block are also useful because they are needed soon as well.

When we need to fetch a block from external memory, chances are that the internal memory is full. In this case we have to *evict* another block—that is, write it back to external memory—before we can bring in the block we need. If we need the evicted block again at some later point in time, we may have to fetch it again. Thus we would like our algorithms to

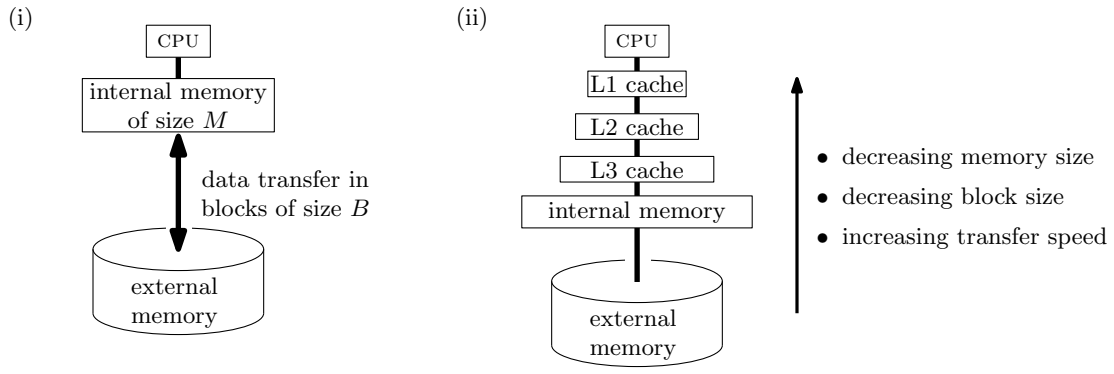

Fig. 1.1: (i) The I/O-model: a two-level memory hierarchy. (ii) A more realistic (but still simplified) multi-level memory hierarchy.

exhibit *temporal locality*: we would like the accesses to any given data element to be clustered in time, so we can keep (the block containing) the element in internal memory for some time and avoid spending an I/O every time we need it. Note that this requires a good *replacement policy* (or, *caching policy*): if we need a block that is currently in internal memory again in the near future, we should not evict it to make room for another block. Instead, we should evict a block that is not needed for a long time.

To summarize, we have the following model; see also Fig. 1.1. We have an internal memory of size M and a disk of unlimited storage capacity. Data is stored in external memory, and transferred between internal and external memory, in blocks of size B. Obviously we must have $M \geqslant B$. Sometimes we need to assume that $M = \Omega(B^2)$; this is called the *tall-cache assumption*.

Analyzing an algorithm in the I/O-model means expressing the number of I/O-operations (block transfers) as a function of M, B, and the input size n. Note that this analysis ignores the number of CPU-operations, which is the traditional measure of efficiency of an algorithm. Of course this is still a relevant measure: for a massive data set, a running time (that is, number of CPU-operations) of $\Theta(n^2)$, say, is problematic even if the number of I/O-operations is small. Hence, an external-memory algorithm should not only have good I/O-efficiency but also a good running time in the traditional sense. Ideally, the running time is the same as the best running time that can be achieved with an internal-memory algorithm.

Controlling the block formation and replacement policy? The I/O-performance of an algorithm is not only determined by the algorithm itself, but also by two other issues: (i) the way in which data elements are grouped into blocks, and (ii) the policy used to decide which block is evicted from internal memory when room has to be made for a new block. In our discussions we will explicitly take the first issue into account, and describe how the data is grouped into blocks in external memory. As for the second issue, we usually make the following assumption: the operating system uses an optimal caching strategy, that is, a strategy that leads to the minimum number of I/Os (for the given algorithm and block formation). This is not very realistic, but as we shall see later, the popular LRU strategy actually comes close to this optimal strategy.

Is the model realistic? The memory organization of a computer is actually much more involved than the abstract two-level model we described above. For instance, besides disk and main memory, there are various levels of cache. Interestingly, the same issues that play a role between main memory and disk also play a role between the cache and main memory: data is transferred in blocks (which are smaller than disk blocks) and one would like to minimize the number of cache misses. The same issues even arise between different cache levels. Thus, even when the data fits entirely in the computer's main memory, the model is relevant: the "internal memory" can then represent the cache, for instance, while the "external memory" represents the main memory. Another simplification we made is in the concept of blocks, which is more complicated than it seems. There is a physical block size (which is the smallest amount of data that can actually be read) but one can artificially increase the block size. Moreover, a disk typically has a disk cache which influences its efficiency. The minimum block size imposed by the hardware is typically around 512 bytes in which case we would have $B = 64$ when the individual elements need 8 bytes. In practice it is better to work with larger block sizes, so most operating systems work with block sizes of 4KB or more.

Despite the above, the abstract two-level model gives a useful prediction of the I/O-efficiency of an algorithm, and algorithms that perform well in this model typically also perform well in practice on massive data sets.

Cache-aware versus cache-oblivious algorithms. To control block formation it is convenient to know the block size B, so that one can explicitly write things like "put these B elements together in one block". Similarly, for some algorithms it may be necessary to know the internal-memory size M. Algorithms that make use of this knowledge are called *cache-aware*. When running a cache-aware algorithm one first has to figure out the values of B and M for the platform the algorithm is running on; these values are then passed on to the algorithm as parameters.

Algorithms that do not need to know B and M are called *cache-oblivious*. For cache-oblivious algorithms, one does not need to figure out the values of B and M—the algorithm can run on any platform without knowledge of these parameters and will be I/O-efficient no matter what their values happen to be for the given platform. A major advantage of this is that cache-oblivious algorithms are automatically efficient across all levels of the memory hierarchy: it is I/O-efficient with respect to data transfer between main memory and disk, it is I/O-efficient with respect to data transfer between L3 cache and main memory, and so on. For a cache-aware algorithm to achieve this, one would have to know the sizes and block sizes of each level in the memory hierarchy, and then set up the algorithm in such a way that it takes all these parameters explicitly into account. Another advantage of cache-oblivious algorithms is that they keep being efficient when the amount of available memory changes during the execution—in practice this can easily happen due to other processes running on the same machine and claiming parts of the memory.

Note that the parameters B and M are not only used in the analysis of cache-aware algorithms but also in the analysis of cache-oblivious algorithms. The difference lies in how the algorithm works—cache-aware algorithms need to know the actual values of B and M, cache-oblivious algorithms do not—and not in the analysis. When analyzing a cache-oblivious algorithm, one assumption is made on the block-formation process: blocks are formed according to the order in which elements are created. For instance, if we create an array A of n elements, then we assume that the first B elements—whatever the value of B may be—are

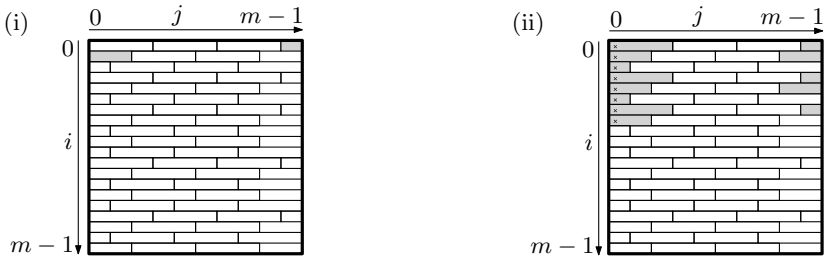

Fig. 1.2: (i) An array stored in row-major order. Blocks may "wrap around" at the end of a row; the block indicated in grey is an example of this. (ii) The blocks accessed to read the first eight elements (the crosses) from the first column.

put into one block, the second B elements are put into the second block, and so on. Another assumption, which was already mentioned earlier, is that the operating system uses an optimal replacement strategy.

1.2 A simple example: Computing the average in a 2-dimensional array

As an easy example of an I/O-analysis, let's consider the problem described earlier: compute the average of the elevation values stored in an array $A[0..m-1, 0..m-1]$. The I/O-efficiency of the two algorithms we discussed—scanning the array row by row or column by column—depends on the way in which A is stored. More precisely, it depends on how the elements are grouped into blocks. Let's assumed A is stored in *row-major order*, as in Fig. 1.2(i).

If we go through the elements of the array row by row, then the access pattern corresponds nicely to the the way in which the elements are blocked: the first B elements that we need, $A[0,0]$ up to $A[0,B-1]$, are stored together in one block, the second B elements are stored together in one block, and so on. As a result, each block is fetched exactly once, and the number of I/Os that *ComputeAverage-RowByRow* uses is $\lceil n/B \rceil$, where $n := m^2$ is the size of the input array.

If we go through the array column by column by column, however, then the first B elements that we need, $A[0,0]$ up to $A[B-1,0]$, are all be stored in different blocks (assuming $B \leqslant m$). The next B elements are again in different blocks, and so on; see Fig. 1.2(ii) for an illustration. Now if the array size is sufficiently large—more precisely, when $m > M/B$—then the internal memory becomes full at some point as we go over the first column. Hence, by the time we go the the second column we have already evicted the block containing $A[0,1]$. Thus we have to fetch this block again. Continuing this argument we see that in every step of the algorithm we will need to fetch a block from external memory. The total number of I/Os is therefore n.

We conclude that the number of I/Os of the row-by-row algorithm and the column-by-column algorithm differs by a factor B. This can make a huge difference in performance: the row-by-row algorithm could take a few minutes, say, while the column-by-column algorithm takes days. Note that both algorithms are cache-oblivious: we only used M and B in the analysis, not in the algorithm.

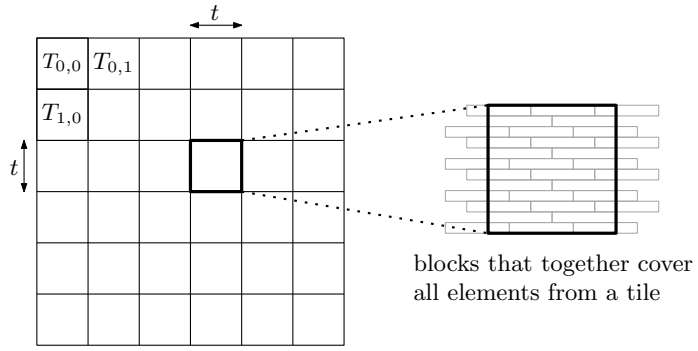

Fig. 1.3: Partitioning of a matrix into tiles, and the blocks needed to cover a given tile.

1.3 Matrix transposition

Many applications in scientific computing involve very large matrices. One of the standard operations one often has to perform on a matrix A is to compute its *transpose* A^{T}. To simplify the notation, let's consider a square matrix $A[0..m-1, 0..m-1]$. Its transpose is the matrix $A^{\mathrm{T}}[0..m-1, 0..m-1]$ defined by

$$A^{\mathrm{T}}[i,j] := A[j,i]. \tag{1.1}$$

Let's assume that A is stored in row-major order, and that we also want to store A^{T} in row-major order. Assume moreover that we do not need to keep A, and that our goal is to transform A into A^{T}. Here's a simple algorithm for this task.

Naive-Transpose(A)

```
1: for i ← 1 to m − 1 do                ▷ A[0..m − 1, 0..m − 1] is an m × m matrix
2:     for j ← 0 to i − 1 do
3:         swap A[i, j] and A[j, i]
4:     end for
5: end for
```

After the algorithm finishes, the array A stores the transpose of the original matrix in row-major order. (Another way to interpret the algorithm is that it takes a 2-dimensional array and changes the way in which the elements are stored from row-major order to column-major order.) It's not hard to see that this algorithm will incur $\Theta(n)$ I/Os if $m > M/B$. Indeed, it effectively traverses A in row-order (to obtain the values $A[i,j]$ in line 3) and in column-order (to obtain the values $A[j,i]$) at the same time. As we already saw, traversing an array in column order when it is stored in row-major order takes $\Theta(n)$ I/Os.

An I/O-efficient cache-aware algorithm. If we know the value of M, it is easy to obtain a more I/O-efficient algorithm. To this end imagine grouping the elements from the matrix into square sub-matrices of size $t \times t$, as in Fig. 1.3. (We will determine a suitable value for t later.) We call these sub-matrices *tiles*. Let's assume for simplicity that m, the size of one row or column, is a multiple of the tile size t; the algorithm is easily adapted when this is not the case. Thus we can number of tiles as $T_{i,j}$, where $0 \leqslant i, j \leqslant m/t - 1$, in such a way that $T_{i,j}$ is the sub-matrix $A[it..(i+1)t-1, jt..(j+1)t-1]$.

Observe that for any given tile $T_{i,j}$, there is exactly one tile (namely $T_{j,i}$) that contains the elements with which the elements from $T_{i,j}$ should be swapped. The idea is to choose the tile size t such that we can store two complete tiles in internal memory. Then we need to read and write each tile only once, which leads to a good I/O-efficiency. It may seem sufficient to take $t := \sqrt{M/2}$, since then a tile contains at most $M/2$ elements. The matrix A is stored in blocks, however, and the total size of the blocks containing the elements from a tile can be bigger than the tile size itself—see Fig. 1.3. So how should we choose t such that the total size of the blocks covering any given tile $T_{i,j}$ does not exceed $M/2$? Note that each row of $T_{i,j}$ has at most two blocks sticking out: one on the left and one on the right. Hence, the blocks that cover a $t \times t$ tile have total size at most $t^2 + 2t(B-1)$. We thus want

$$t^2 + 2t(B-1) \leqslant M/2,$$

which is satisfied when we take $t := \sqrt{M/2 + B^2} - B$. (Of course t must be an integer, so we should round this value down. Moreover, we can actually take $t := \left\lfloor \sqrt{M/2 + (B-1)^2} - (B-1) \right\rfloor$. In the following we will use the value $t = \sqrt{M/2 + B^2} - B$ to avoid cluttering the formulas.) This gives us the following algorithm.

Algorithm 1.1 Cache-aware algorithm for matrix transposition.

CacheAware-Transpose(A, M, B)
1: $t \leftarrow \sqrt{M/2 + B^2} - B$ ▷ M is size of internal memory, B is block size
2: **for** $i \leftarrow 0$ **to** m/t **do** ▷ $A[0..m-1, 0..m-1]$ is an $m \times m$ matrix
3: **for** $j \leftarrow 0$ **to** i **do**
4: Read the tiles $T_{i,j}$ and $T_{j,i}$ from external memory.
5: Swap the elements of these tiles according to Equation (1.1).
6: Write the tiles back to external memory.
7: **end for**
8: **end for**

Theorem 1.1 *Let $A[0..m-1, 0..m-1]$ be a matrix of size $n := m^2$. Under the tall-cache assumption, we can transpose A with a cache-aware algorithm that performs $O(n/B)$ I/Os.*

Proof. Consider algorithm *CacheAware-Transpose*. It handles $O(n/t^2)$ pairs of tiles. To read a given tile from external memory, we read at most $t/B + 2$ blocks per row of the tile. Hence, in total we need at most $2t(t/B + 2)$ I/Os to read two tiles. Similarly, the number of I/Os needed to write a tile back to external memory is $2t(t/B + 2)$. The total number of I/Os over all pairs of tiles is therefore bounded by

$$O\left(\frac{n}{t^2} \cdot 4t(t/B + 2)\right) = O\left(\frac{n}{B} + \frac{n}{t}\right),$$

where $t = \sqrt{M/2 + B^2} - B$. Under the tall-cache assumption we have

$$t = \sqrt{M/2 + B^2} - B \geqslant \sqrt{B^2/2 + B^2} - B = (\sqrt{3/2} - 1)B$$

which implies $n/t = O(n/B)$, thus proving the theorem. □

It is instructive to think about why the tall-cache assumption is actually needed. Intuitively, the reason is as follows. To obtain $O(n/B)$ I/Os, we can use only $O(t^2/B)$ I/Os per tile. Stated differently, we want to use only $O(t/B)$ I/Os per row of the tile. Because there can be two blocks that are sticking out, the actual number of blocks per row is $\lfloor t/B \rfloor + 2$. The "+2" in this formula disappears in the O-notation, that is, $\lfloor t/B \rfloor + 2 = O(t/B)$, *but only when* $t/B = \Omega(1)$. In other words, we need $t = \Omega(B)$ to be able to "pay" for fetching the two blocks sticking out. Hence, a tile should have size $\Omega(B^2)$. As we want to be able to store two tiles completely in internal memory, we thus need $M = \Omega(B^2)$.

A cache-oblivious algorithm. The matrix-transposition algorithm described above needs to know the memory size M and block size B. We now give a cache-oblivious algorithm for matrix transposition. The new algorithm uses an idea that is useful in many other contexts: If we apply divide-and-conquer, then the problem size in the recursive calls decreases gradually from n (the initial problem size) to $O(1)$ (the base case). Hence, there will always be a moment when the subproblem to be solved has roughly size M and can be solved entirely in internal memory. Hence, if we can perform the conquer-step in the algorithm in an I/O-efficient and cache-oblivious manner, then the whole algorithm is I/O-efficient and cache-oblivious.

Our recursive algorithm will have four parameters (besides the array A): indices i_1, i_2, j_1, j_2 with $i_1 \leqslant i_2$ and $j_1 \leqslant j_2$. The task of the algorithm is to swap the elements in the sub-matrix $A[i_1..i_2, j_1..j_2]$ with the elements in the sub-matrix $A[j_1..j_2, i_1..i_2]$, according to Equation (1.1). We will make sure that $A[i_1..i_2, j_1..j_2]$ either lies entirely above the main diagonal of the matrix A, or the diagonal of the sub-matrix is a part of the diagonal of A. This ensures that each element is swapped exactly once, as required. Initially we have $i_1 = j_1 = 0$ and $i_2 = j_2 = m - 1$.

In a generic step, the algorithm splits the sub-matrix $A[i_1..i_2, j_1..j_2]$ into four sub-matrices on which it recurses. If diagonal of A crosses the sub-matrix—when this happens we must have $i_1 = j_1$ and $i_2 = j_2$—then one of the four smaller sub-matrices lies below the diagonal of A. Hence, we should not recurse on this sub-matrix. The test in line 8 of the algorithm below takes care of this. The recursion stops when $i_1 = i_2$ or $j_1 = j_2$. It is not hard to verify that in this case we either have a 1×1 sub-matrix, or a 2×1 sub-matrix, or a 1×2 sub-matrix. This base case ensures that we never make a call on an empty sub-matrix.

Algorithm 1.2 Cache-oblivious algorithm for matrix transposition.

CacheOblivious-Transpose(A, i_1, i_2, j_1, j_2)
1: **if** $i_1 = i_2$ or $j_1 = j_2$ **then**
2: swap $A[i_1..i_2, j_1..j_2]$ and $A[j_1..j_2, i_1..i_2]$ according to Equation (1.1)
3: **else**
4: $i_{\text{mid}} \leftarrow \lfloor (i_1 + i_2)/2 \rfloor$; $j_{\text{mid}} \leftarrow \lfloor (j_1 + j_2)/2 \rfloor$
5: *CacheOblivious-Transpose*$(A, i_1, i_{\text{mid}}, j_1, j_{\text{mid}})$
6: *CacheOblivious-Transpose*$(A, i_{\text{mid}} + 1, i_2, j_1, j_{\text{mid}})$
7: *CacheOblivious-Transpose*$(A, i_{\text{mid}} + 1, i_2, j_{\text{mid}} + 1, j_2)$
8: **if** $i_1 \geqslant j_{\text{mid}} + 1$ **then**
9: *CacheOblivious-Transpose*$(A, i_1, i_{\text{mid}}, j_{\text{mid}} + 1, j_2)$
10: **end if**
11: **end if**

Theorem 1.2 *Let $A[0..m-1, 0..m-1]$ be a matrix of size $n := m^2$. Under the tall-cache assumption, we can transpose A with a cache-oblivious algorithm that performs $O(n/B)$ I/Os.*

Proof. Consider algorithm *CacheOblivious-Transpose*. One way to think about the algorithm is that it recursively partitions A into sub-matrices until the sub-matrices are such that two of them (including the blocks that are "sticking out") fit in internal memory. At this point the sub-matrices have the same size as the tiles in the cache-aware algorithm. A recursive call on such a sub-matrix will now partition the sub-matrix into even smaller pieces, but from the I/O point of view this is irrelevant: all data needed for the call and sub-calls from now on fits in internal memory, and the assumption that the operating system uses an optimal replacement policy guarantees that all relevant blocks are read only once. Hence, the same computation as in the analysis of the cache-aware algorithm shows that the cache-oblivious algorithm performs $O(n/B)$ I/Os.

A more precise proof can be given by writing a recurrence for $T(t)$, the number of I/Os the algorithm performs when called on a sub-matrix of total size $t \times t$. (For simplicity we ignore the fact that the dimensions of the array may differ by one in a recursive call, that is, we ignore that a call can also be on a $t \times (t+1)$ or $(t+1) \times t$ sub-matrix.) Following the arguments from the proof of Theorem 1.1, we see that when $2t^2 + 4t(B-1) < M$, the entire computation fits in internal memory. Hence, we have

$$T(t) \leqslant \begin{cases} 4t(t/B+2) & \text{if } 2t^2 + 4t(B-1) < M \\ 4T(t/2) & \text{otherwise} \end{cases}$$

Using the tall-cache assumption one can now show that $T(t) = O(t^2/B)$. Since in the first call we have $t = m = \sqrt{n}$ this proves the theorem. □

1.4 Replacement policies

When describing our I/O-efficient algorithms we do not always explicitly describe how block replacement is being done—in particular we do not describe this for cache-oblivious algorithms. Instead we make the assumption that the operating system employs an optimal block-replacement policy. In this section we show that this is not as unrealistic as it may seem. In particular, we show that the popular LRU policy is close to being optimal. First, let's define LRU and another replacement policy called MIN. Both policies only evict a block when necessary, that is, when the internal memory is full and we need to make room to be able to bring in a block from external memory. The difference in the two policies is which block they evict.

LRU **(Least Recently Used).** The LRU replacement policy always evicts the block that has not been used for the longest time. In other words, if τ_i denotes the last time any data element in the i-th block was accessed then LRU will evict the block for which τ_i is smallest. The idea is that if a block has been used recently, then chances are it will be needed again soon because the algorithm (hopefully) has good temporal locality. Thus it is better to evict a block that has not been used for a long time.

MIN **(Longest Forward Distance).** The MIN replacement policy always evicts the block whose next usage is furthest in the future, that is, the block that is not needed for the

longest period of time. (If there are blocks that are not needed at all anymore, then such a block is chosen.) One can show that MIN is optimal: for any algorithm (and memory size and block size), the MIN policy performs the minimum possible number of I/Os.

There is one important difference between these two policies: LRU is an *on-line* policy—a policy that can be implemented without knowledge of the future—while MIN is not. In fact, MIN cannot be implemented by an operating system, because the operating system does not know how long it will take before a block is needed again. However, we can still use it as a "golden standard" to assess the effectiveness of LRU.

Suppose we run an algorithm ALG on a given input I, and with an initially empty internal memory. Assume the block size B is fixed. We now want to compare the number of I/Os that LRU needs to the number of I/Os that MIN would need. It can be shown that if LRU and MIN have the same amount of internal memory available, then LRU can be much worse than MIN; see Exercise 1.7. However, when we give LRU slightly more internal memory to work with, then the performance comes close to the performance of MIN. To make this precise, we define

LRU (ALG,M) := number of I/O-operations performed when algorithm ALG is run with the LRU replacement policy and internal-memory size M.

We define MIN (ALG,M) similarly for the MIN replacement policy. We now have the following theorem.

Theorem 1.3 *For any algorithm* ALG*, and any M and M' with $M \geqslant M'$, we have*

$$\text{LRU}(\text{ALG}, M) \leqslant \frac{M}{M - M' + B} \cdot \text{MIN}(\text{ALG}, M').$$

In particular, $\text{LRU}(\text{ALG}, M) < 2 \cdot \text{MIN}(\text{ALG}, M/2)$.

Proof. Consider the algorithm ALG when run using LRU and with internal-memory size M. Let $t_1, t_2, \ldots, t_s$, with $s = \text{LRU}(\text{ALG}, M)$, be the moments in time where LRU fetches a block from external memory. Also consider the algorithm when run using MIN and internal-memory size M'. Note that up to $t_{M'/B}$, LRU and MIN behave the same: they simply fetch a block when they need it, but since the internal memory is not yet full, they do not need to evict anything.

Now partition time into intervals $T_0, T_1, \ldots$ in such a way that LRU fetches exactly M/B blocks during T_j for $j \geqslant 1$, and at most M/B blocks during T_0. Here we count each time any block is fetched, that is, if the same block is fetched multiple times it is counted multiple times. We analyze the behavior of MIN on these time intervals, where we treat T_0 separately.

- During T_0, LRU fetches at most M/B blocks from external memory. Since we assumed that we start with an empty internal memory, the blocks fetched in T_0 are all different. Since MIN also starts with an empty internal memory, it must also fetch each of these blocks during T_0.

- Now consider any of the remaining time intervals T_i. Let b be the last block accessed by the algorithm before T_i. Thus, at the start of T_i, both LRU and MIN have b in internal memory. We first prove the following claim:

  Claim. Let $\mathcal{B}_i$ be the set of blocks accessed during T_i. (Note that $\mathcal{B}_i$ contains all blocks

that are accessed, not just the ones that need to be fetched from external memory.) Then $\mathcal{B}_i$ contains at least M/B blocks that are different from b.

To prove the claim, we distinguish three cases. First, suppose that b is evicted by LRU during T_i. Because b is the most recently used block in LRU's memory at the start of T_i, at least $M/B - 1$ other blocks must be accessed before b can become the least recently used block. We then need to access at least one more block (from external memory) before b needs to be evicted, thus proving the claim. Second, suppose that some other block b' is fetched twice by LRU during T_i. Then a similar argument as in the first case shows that we need to access at least M/B blocks in total during T_i. If neither of these two cases occurs, then all blocks fetched by LRU during T_i are distinct (because the second case does not apply) and different from b (because the first case does not apply), which also implies the claim.

At the start of T_i, MIN has only M'/B blocks in its internal memory, one of which is b. Hence, at least $M/B - M'/B + 1$ blocks from $\mathcal{B}_i$ are not in MIN's internal memory at the start of T_i. These blocks must be fetched by MIN during T_i. This implies that

$$\frac{\text{number of I/Os performed by LRU during } T_i}{\text{number of I/Os performed by MIN during } T_i} \leqslant \frac{M/B}{M/B - M'/B + 1} = \frac{M}{M - M' + B}.$$

We conclude that during T_0 LRU uses at most the same number of I/Os as MIN, and for each subsequent time interval T_i the ratio of the number of I/Os is $M/(M - M' + B)$. The theorem follows. □

1.5 Exercises

Exercise 1.1 Consider a machine with 1 GB of internal memory and a disk with the following properties:

- the average time until the read/write head is positioned correctly to start reading or writing data (seek time plus rotational delay) is 12 ms,
- once the head is positioned correctly, we can read or write at a speed of 60 MB/s.

The standard block size used by the operating system to transfer data between the internal memory and the disk is 4 KB. We use this machine to sort a file containing 10^8 elements, where the size of each element is 500 bytes; thus the size of the file is 50 GB. Our sorting algorithm performs roughly $2\frac{n}{B}\lceil \log_{M/B} \frac{n}{B} \rceil$ I/Os for a set of n elements, where M is the number of elements that fit into the internal memory and B is the number of elements that fit into a block. Here we assume that 1 KB = 10^3 bytes, 1 MB = 10^6 bytes, and 1 GB = 10^9 bytes.

(i) Compute the total time in hours spent on I/Os by the algorithm when we work with the standard block size of 4 KB.

(ii) Now suppose we force the algorithm to work with blocks of size 1 MB. What is the time spent on I/Os in this case?

(iii) Same question as (ii) for a block size of 250 MB.

Exercise 1.2 Consider the algorithm that computes the average value in an $m \times m$ array A column by column, for the case where A is stored in row-major order. Above (on page 8) it was stated that the algorithm performs n I/Os, where $n := m^2$ is the total size of the array, because whenever we need a new entry from the array we have already evicted the block containing that entry. This is true when LRU is used and when $m > M/B$, but it is not immediately clear what happens when some other replacement policy is used.

(i) Suppose that $m = M/B + 1$, where you may assume that M/B is integral. Show that in this case there is a replacement policy that would perform only $O(n/B + \sqrt{n})$ I/Os.

(ii) Prove that when $m > 2M/B$, then any replacement policy will perform $\Omega(n)$ I/Os.

(iii) In the example in the introduction to this chapter, the array storing the elevation values had size $100,000 \times 100,000$ and each elevation value was an 8-byte number. Suppose that the block size B is 1024 bytes. Is it realistic to assume that we cannot store one block for each row in internal memory in this case? Explain your answer.

(iv) Suppose m, M, and B are such that $m = M/(2B)$. Thus we can easily store one block for each row in internal memory. Would you expect that in this case the actual running times of the row-by-row algorithm and the column-by-column algorithm are basically the same? Explain your answer.

Exercise 1.3 A *stack* is a data structure that supports two operations: we can *push* a new element onto the stack, and we can *pop* the topmost element from the stack. We wish to implement a stack in external memory. To this end we maintain an array $A[0..m-1]$ on disk. The value m, which is the maximum stack size, is kept in internal memory. We also maintain the current number of elements on the stack, s, in internal memory. The *push*- and *pop*-operations can now be implemented as follows:

Push(A, x)
1: **if** $s = m$ **then**
2: **return** "stack overflow"
3: **else**
4: $A[s] \leftarrow x$; $s \leftarrow s + 1$
5: **end if**

Pop(A)
1: **if** $s = 0$ **then**
2: **return** "stack is empty"
3: **else**
4: **return** $A[s-1]$; $s \leftarrow s - 1$
5: **end if**

Assume A is blocked in the standard manner: $A[0 \ldots B-1]$ is the first block, $A[B \ldots 2B-1]$ is the second block, and so on.

(i) Suppose we allow the stack to use only a single block from A in internal memory. Show that there is a sequence of n operations that requires $\Theta(n)$ I/Os.

(ii) Now suppose we allow the stack to keep two blocks from A in internal memory. Prove that any sequence of n *push*- and *pop*-operations requires only $O(n/B)$ I/Os.

Exercise 1.4 Let $A[0..n-1]$ be an array with n elements that is stored on disk, where n is even. We want to generate a random subset of $n/2$ elements from A, and store the subset in an array $B[0..(n/2)-1]$.

The following algorithm computes such a random subset. It simply picks $n/2$ times a random element from A. (Note that the same element may be picked multiple times). To this

end, the algorithm uses a random-number generator $Random(a, b)$ that, given two integers a and b with $a \leqslant b$, generates an integer $r \in \{a, a+1, \ldots, b\}$ uniformly at random.

RandomSubset-I(A)
1: $\triangleright$ $A[0..n-1]$ is an array with n elements, for n even.
2: **for** $i \leftarrow 0$ **to** $(n/2)-1$ **do**
3: $\quad r \leftarrow Random(0, n-1)$
4: $\quad B[i] \leftarrow A[r]$
5: **end for**

We can also generate a random subset of $n/2$ distinct elements, by going over all the n elements in A and deciding for each element whether to select it into the subset. More precisely, when we arrive at element $A[i]$ (and we still have $n-i$ elements left to choose from, including $A[i]$) and we already selected m elements (and thus we still need to select $n/2-m$ elements) then we select $A[i]$ with probability $(n/2-m)/(n-i)$.

RandomSubset-II(A)
1: $\triangleright$ $A[0..n-1]$ is an array with n elements, for n even.
2: $m \leftarrow 0$ $\triangleright$ m is the number of elements already selected.
3: **for** $i \leftarrow 0$ **to** $n-1$ **do**
4: $\quad r \leftarrow Random(1, n-i)$
5: $\quad$ **if** $r \leqslant n/2-m$ **then**
6: $\quad\quad B[m] \leftarrow A[i]$; $m \leftarrow m+1$
7: $\quad$ **end if**
8: **end for**

Assume the memory size M is much smaller than the array size n. Analyze the expected number of I/Os performed by both algorithms. Take into account the I/Os needed to read elements from A and the I/Os needed to write elements to B.

Exercise 1.5 Suppose we are given two $m \times m$ matrices X and Y, which are stored in row-major order in 2-dimensional arrays $X[0..m-1, 0..m-1]$ and $Y[0..m-1, 0..m-1]$. We wish to compute the product $Z = XY$, which is the $m \times m$ matrix defined by

$$Z[i,j] := \sum_{k=0}^{m-1} X[i,k] \cdot Y[k,j],$$

for all $0 \leqslant i, j < m$. The matrix Z should also be stored in row-major order. Let $n := m^2$. Assume that m is much larger than M and that $M \geqslant B^2$.

(i) Analyze the I/O-complexity of the matrix-multiplication algorithm that simply computes the elements of Z one by one, row by row. You may assume LRU is used as replacement policy, and that one third of the internal memory is reserved for each of the matrices X, Y and Z.

(ii) Analyze the I/O-complexity of the same algorithm if X and Z are stored in row-major order while Y is stored in column-major order. You may assume LRU is used as replacement policy, and that one third of the internal memory is reserved for each of the matrices X, Y and Z.

(iii) Consider the following alternative algorithm. For a square matrix Q with more than one row and column, let Q_{TL}, Q_{TR}, Q_{BL}, Q_{BR} be the top left, top right, bottom left, and bottom right quadrant, respectively, that result from cutting Q between rows $\lfloor m/2 \rfloor$ and $\lfloor m/2 \rfloor + 1$, and between columns $\lfloor m/2 \rfloor$ and $\lfloor m/2 \rfloor + 1$. We can compute $Z = XY$ recursively by observing that

$$\begin{aligned} Z_{TL} &= X_{TL}Y_{TL} + X_{TR}Y_{BL}, \\ Z_{TR} &= X_{TL}Y_{TR} + X_{TR}Y_{BR}, \\ Z_{BL} &= X_{BL}Y_{TL} + X_{BR}Y_{BL}, \\ Z_{BR} &= X_{BL}Y_{TR} + X_{BR}Y_{BR}. \end{aligned}$$

Analyze the I/O-complexity of this recursive computation. You may assume that m is a power of two, and that an optimal replacement policy is used.
Hint: Express the I/O-complexity as a recurrence with a suitable base case, and solve the recurrence.

(iv) If you solved (ii) and (iii) correctly, you have seen that the number of I/Os of the recursive algorithm is smaller than the number of I/Os in part (ii). Does the algorithm from (iii) have better spatial locality than the algorithm from (ii)? And does it have better temporal locality? Explain your answers.

Exercise 1.6 Let $A[0..n-1]$ be a sorted array of n numbers, which is stored in blocks in the standard way: $A[0..B-1]$ is the first block, $A[B..2B-1]$ is the second block, and so on.

(i) Analyze the I/O-complexity of binary search on the array A. Make a distinction between the case where the internal memory can store only one block and the case where it can store more than one block.

(ii) Describe a different way to group the elements into blocks such that the I/O-complexity of binary search is improved significantly, and analyze the new I/O-complexity. Does it make a difference if the internal memory can store one block or more than one block?

(iii) Does your solution improve spatial and/or temporal locality? Explain your answer.

Exercise 1.7 This exercise shows that the bound from Theorem 1.3 is essentially tight.

(i) Suppose that an algorithm operates on a data set of size $M + B$, which is stored in blocks $b_1, \ldots, b_{M/B+1}$ in external memory. Consider LRU and MIN, where the replacement policies have the same internal memory size, M. Show that there is an infinitely long sequence of block accesses such that, after the first M/B block accesses (on which both LRU and MIN perform an I/O) LRU will perform an I/O for every block access while MIN only performs an I/O once every M/B access.
NB: On an algorithm with this access sequence we have $\text{LRU}(\text{ALG}, M) \to \frac{M}{B} \cdot \text{MIN}(\text{ALG}, M)$ as the length of the sequence goes to infinity, which shows that the bound from Theorem 1.3 is essentially tight for $M = M'$.

(ii) Now suppose LRU has an internal memory of size M while MIN has an internal memory of size M'. Generalize the example from (i) to show that there are infinitely long access sequences such that $\text{LRU}(\text{ALG}, M) \to \frac{M}{M-M'+B} \cdot \text{MIN}(\text{ALG}, M')$ as the length of

the sequence goes to infinity. (This shows that the bound from Theorem 1.3 is also essentially tight for $M > M'$.)

Exercise 1.8 Consider an image-processing application that repeatedly scans an image, over and over again, row by row from the top down. The image is also stored row by row in memory. The image contains n pixels, while the internal memory can hold M pixels and pixels are moved into and out of cache in blocks of size B, where $M \geqslant 3B$.

(i) Construct an example—that is, pick values for B, M and n—such that the LRU caching policy results in M/B times more cache misses than an optimal caching policy for this application (excluding the first M/B cache misses of both strategies).

(ii) If we make the image only half the size, then what is the performance ratio of LRU versus optimal caching?

(iii) If we make the image double the size, then what is the performance ratio of LRU versus optimal caching?

Exercise 1.9 Consider an algorithm that needs at most $cn\sqrt{n}/(MB)$ I/Os if run with optimal caching, for some constant c. Prove that the algorithm needs at most $c'n\sqrt{n}/(MB)$ I/Os when run with LRU caching, for a suitable constant c'.

Exercise 1.10 Let $X[0..n-1]$ and $Y[0..n-1]$ be two arrays of n numbers, where $n \gg M$. Suppose we have a function $f : \mathbb{R}^2 \to \mathbb{R}$ and we want to compute $\min\{f(X[i], Y[j]) : 0 \leqslant i < n \text{ and } 0 \leqslant j < n\}$. If we have no other information about f, then the only thing we can do is compute $f(X[i], Y[j])$ for all pairs i, j with $0 \leqslant i < n$ and $0 \leqslant j < n\}$. A simple way to do this is using the following algorithm.

```
FindMin(X, Y)
1: z ← +∞
2: for i ← 0 to n − 1 do
3:     for j ← 0 to n − 1 do
4:         z ← min(z, f(X[i], Y[j]))
5:     end for
6: end for
7: return z
```

(i) Analyze the number of I/Os performed by *FindMin*.

(ii) Give a cache-aware algorithm that solves the problem using $O(n^2/(MB))$ I/Os.

(iii) Give a cache-oblivious algorithm that solves the problem using $O(n^2/(MB))$ I/Os.

Chapter 2

Sorting and Permuting

In this chapter we study I/O-efficient algorithms for sorting. We will present a sorting algorithm that performs $O(\frac{n}{B}\log_{M/B}\frac{n}{B})$ I/Os in the worst case to sort n elements, and we will prove that this is optimal (under some mild assumptions). For simplicity we assume the input to our sorting algorithm is an array of n numbers. (In an actual application the array would store a collection of records, each storing a numerical *key* and various other information, and we want to sort the records by their key.) We also assume that the numbers we wish to sort are distinct; the algorithms can easily be adapted to deal with the case where numbers can be equal.

2.1 An I/O-efficient sorting algorithm

Let $A[0..n-1]$ be an array of n distinct numbers. We want to sort A into increasing order. There are many algorithms that can sort A in $O(n\log n)$ time when A fits entirely in internal memory. One such algorithm is *MergeSort*. We will first show that *MergeSort* already has fairly good I/O-behavior. After that we will modify *MergeSort* to improve its I/O-efficiency even more. As usual, we assume that initially the array A is stored consecutively on disk (or rather, we assume that $A[0..B-1]$ is one block, $A[B..2B-1]$ is one block, and so on).

MergeSort is a divide-and-conquer algorithm: it partitions the input array A into two smaller arrays A_1 and A_2 of roughly equal size, recursively sorts A_1 and A_2, and then merges the two results to obtain the sorted array A. In the following pseudocode, $\mathit{length}(A)$ denotes the length (that is, number of elements) of an array A.

MergeSort(A)
1: $n \leftarrow \mathit{length}(A)$
2: **if** $n > 1$ **then** ▷ else A is sorted by definition
3: $n_{\text{left}} \leftarrow \lfloor n/2 \rfloor$; $n_{\text{right}} \leftarrow \lceil n/2 \rceil$
4: $A_1[0..n_{\text{left}}-1] \leftarrow A[0..n_{\text{left}}-1]$; $A_2[0..n_{\text{right}}-1] \leftarrow A[n_{\text{left}}..n-1]$
5: *MergeSort*(A_1); *MergeSort*(A_2)
6: Merge A_1 and A_2 to obtain the sorted array A
7: **end if**

In line 6 we have to merge the sorted arrays A_1 and A_2 to get the sorted array A. This can be done by simultaneously scanning A_1 and A_2 and writing the numbers we encounter to their correct position in A. More precisely, when the scan of A_1 is at position $A_1[i]$ and the

scan of A_2 is at position $A_2[j]$, we write the smaller of the two numbers to $A[i+j]$—in other words, we set $A[i+j] \leftarrow \min(A_1[i], A_2[j])$—and we increment the corresponding index (i or j). When the scan reaches the end of one of the two arrays, we simply write the remaining numbers in the other array to the remaining positions in A. This way the merge step takes $O(n)$ time. Hence, running time $T(n)$ of the algorithm satisfies[1] $T(n) = 2T(n/2) + O(n)$ with $T(1) = O(1)$, and so $T(n) = O(n \log n)$.

Let's now analyze the I/O-behavior of *MergeSort*. Note that the merge step only needs $O(n/B)$ I/Os, because it just performs three scans: one of A_1, one of A_2, and one of A. Hence, if $T_{\text{IO}}(t)$ denotes the (worst-case) number of I/Os performed when *MergeSort* is run on an array of length t, then

$$T_{\text{IO}}(t) = 2T_{\text{IO}}(t/2) + O(t/B). \tag{2.1}$$

What is the base case for the recurrence? When writing a recurrence for the running time of an algorithm, we usually take $T(1) = O(1)$ as base case. When analyzing the number of I/Os, however, we typically have a different base case: as soon as we have a recursive call on a subproblem that can be solved completely in internal memory, we only need to bring the subproblem into internal memory once (and write the solution back to disk once).

The analysis of *MergeSort* becomes a bit easier if we slightly change the algorithm. Instead of copying the first half of A to A_1 and the second half to A_2, we make the algorithm work *in place*: we give the algorithm two extra parameters, i and j, such that a call of the form *MergeSort*(A, i, j) will sort the subarray $A[i..j]$. The two recursive calls are then *MergeSort*(A, i, i_{mid}) and *MergeSort*$(A, i_{\text{mid}} + 1, j)$, where $i_{\text{mid}} = \lfloor (i+j)/2 \rfloor$. This way we do not need extra storage for the arrays A_1 and A_2. Thus a recursive call on a subarray of size t fits in the internal memory when $t = M$. This is not quite precise, as there may be blocks sticking out of the subarray. If we take this into account, the base case becomes:

$$T_{\text{IO}}(t) = O(M/B) \text{ when } t + 2(B-1) \leqslant M.$$

Together with (2.1) this implies that $T_{\text{IO}}(n) = O((n/B)\log_2(n/M))$.

This I/O bound is already quite good. It is not optimal, however: the base of the logarithm can be improved. Note that the factor $O(\log_2(n/M))$ in the I/O-bound is equal to the number of levels of recursion before the size of the subproblem drops below (roughly) M. The logarithm in this number has base 2 because we partition the input array into two (roughly equal sized) subarrays in each recursive step; if we would partition into k subarrays, for some $k > 2$, then the base of the logarithm would be k. Thus we change the algorithm as follows. We would like to choose k as large as possible. However, we still have to be able to do the merge step with only $O(n/B)$ I/Os. This means that we should be able to keep at least one block from each of the subarrays $A_1, \ldots, A_k$ (as well as from A) in main memory, so that we can simultaneously scan all these arrays without running into memory problems. Hence, we set $k := M/B - 1$ and we get the recurrence

$$T_{\text{IO}}(t) = \sum_{i=1}^{k} T_{\text{IO}}(t/k) + O(t/B). \tag{2.2}$$

[1] More precisely, we have $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$, but (as usual) we omit the floor and ceiling for simplicity.

Algorithm 2.1 A cache-aware I/O-efficient version of *MergeSort*.

```
EM-MergeSort(A)
1: n ← length(A)
2: if n > 1 then                                  ▷ else A is sorted by definition
3:     Pick a suitable value of k (see below).
4:     Partition A into k roughly equal-sized subarrays A_1, ..., A_k.
5:     for i ← 1 to k do
6:         EM-MergeSort(A_i)
7:     end for
8:     Merge A_1, ..., A_k to obtain the sorted array A
9: end if
```

As above, we actually do not explicitly create the subarrays $A_1, \ldots, A_k$ but implement the algorithm in place, which means we have the same base case, namely $T_{\text{IO}}(t) = O(M/B)$ when $t + 2(B_1) \leqslant M$. The solution is $T_{\text{IO}}(n) = O((n/B)\log_{M/B}(n/M))$, which is equivalent[2] to

$$T_{\text{IO}}(n) = O((n/B)\log_{M/B}(n/B)).$$

(See also Exercise 2.2.) Thus we managed to increase the base of the logarithm from 2 to M/B. This increase can be significant. For instance, for $n = 1,000,000,000$ and $M = 1,000,000$ and $B = 100$ we have $\log_2 n \approx 29.9$ and $\log_{M/B} n = 2.25$, so changing the algorithm as just described may give a 10-fold reduction in the number of I/Os. (This computation is perhaps not very meaningful, because we did not analyze the constant factor in the number of I/Os. These are similar, however, and in practice the modified algorithm indeed performs much better for very large data sets.)

The above leads to the following theorem.

Theorem 2.1 *An array $A[0..n-1]$ with n numbers can be sorted using $O((n/B)\log_{M/B}(n/B))$ I/Os using a k-way variant of* MergeSort*, where $k := M/B - 1$.*

Notice that in order to obtain the best performance, we take $k := M/B - 1$. Thus the choice of k depends on M and B, which means that the modified version of the algorithm is no longer cache-oblivious. There are also cache-oblivious sorting algorithms that perform only $O((n/B)\log_{M/B}(n/B))$ I/Os.

Many external-memory algorithms use sorting as a subroutine, and often the sorting steps performed by the algorithm determine its running time. Hence, it is convenient to introduce a shorthand for the number of I/Os needed to sort n elements on a machine with a memory size M and block size B. Thus we define $\text{Sort}(n) := O((n/B)\log_{M/B}(n/B))$.

2.2 The permutation lower bound

In the previous section we saw a sorting algorithm that performs $O((n/B)\log_{M/B}(n/B))$ I/Os. In this section we show that (under certain mild conditions) this is optimal. In fact, we will prove a lower bound on the worst-case number of I/Os needed to sort an array of n numbers,

[2]The second expression is usually preferred because n/B is the size of the input in terms of the number of blocks; hence, this quantity is a natural one to use in bounds on the number of I/Os.

even if we already know the rank of each number, that is, even if we already know where each number needs to go. To state the result more formally, we first define the *permutation problem* as follows.

Let $A[0..n-1]$ be an array where each $A[i]$ stores a pair (pos_i, x_i) such that the sequence $pos_0, pos_1, \ldots, pos_{n-1}$ of ranks is a permutation of $0, \ldots, n-1$. The permutation problem is to rearrange the array such that (pos_i, x_i) is stored in $A[pos_i]$. Note that the sorting problem is at least as hard as the permutation problem, since we not only need to move each number in the array to its correct position in the sorted order, but we also need to determine the correct position. Below we will prove a lower bound on the number of I/Os for the permutation problem, which thus implies the same lower bound for the sorting problem.

To prove the lower bound we need to make certain assumptions on what the permutation algorithm is allowed to do, and what it is not allowed to do. These assumptions are as follows.

- The algorithm can only *move* elements from external memory to internal memory and vice versa; in particular, the algorithm is not allowed to copy or modify elements.
- All read and write operations are performed on *full blocks*.
- When writing a block from internal to external memory, the algorithm can choose any B items from internal memory to form the block. (A different way of looking at this is that the algorithm can move around the elements in internal memory for free.)

We will call this the *movement-only model*. Observe that the second assumption implies that n is a multiple of B. In the movement-only model a permutation algorithm can be viewed as consisting of a sequence of read and write operations, where a read operations selects a block from external memory and brings it into internal memory—of course this is only possible if there is still space in the internal memory—and a write operation chooses any B elements from the internal memory and writes them as one block to a certain position in the external memory. The task of the permutation algorithm is to perform a number of read and write operations so that at end of the algorithm all elements in the array A are stored in the correct order, that is, element (pos_i, x_i) is stored in $A[pos_i]$ for all $0 \leqslant i < n$.

Theorem 2.2 *Any algorithm that solves the permutation problem in the movement-only model needs $\Omega((n/B)\log_{M/B}(n/B))$ I/Os in the worst case, assuming that $n < B\sqrt{\binom{M}{B}}$.*

Proof. The global idea of our proof is as follows. Let X be the total number of I/Os performed by the permutation algorithm, in the worst case. Then within X I/Os the algorithm has to be able to rearrange the input into blocks in many different ways, depending on the particular permutation to be performed. This means that algorithm should be able to reach many different "states" within X I/Os. However, one read or write can increase the number of different states only by a certain factor. Thus, if we can determine upper bounds on the increase in the number of states by doing a read or write, and we can determine the total number of different final states that the algorithm must be able to reach, then we can derive a lower bound on the total number of I/Os. Next we make this idea precise.

We assume for simplicity that the permutation algorithm only uses the part of the external memory occupied by the array A. Thus, whenever it writes a block to memory, it writes to $A[jB..(j+1)B-1]$ for some $0 \leqslant j < n/B$. (There must always be an empty block when a write operation is performed, because elements are not copied.) This assumption

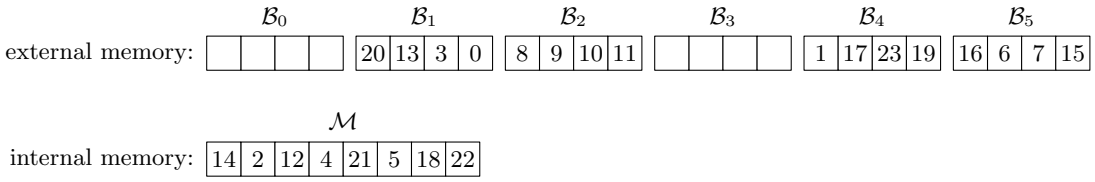

Fig. 2.1: A possible state of the permutation algorithm after several read and write operations. The numbers in the figure indicate the desired positions pos_i of the input elements. Note that $\mathcal{B}_2$ already contains the correct elements.

is not necessary—see Exercise 2.7—but it simplifies the presentation. Now we can define the *state* of (the memory of) the algorithm as the tuple $(\mathcal{M}, \mathcal{B}_0, \ldots, \mathcal{B}_{n/B-1})$, where $\mathcal{M}$ is the set of elements stored in the internal memory and the $\mathcal{B}_j$ is the set of elements in the block $A[jB..(j+1)B-1]$; see Fig. 2.1. Note that the order of the elements within a block is unimportant for the state. Because we only read and write full blocks, each subset $\mathcal{B}_j$ contains exactly B elements. Moreover, because elements cannot be copied or modified, any element is either in $\mathcal{M}$ or it is in (exactly) one of the subsets $\mathcal{B}_j$. Initially the algorithm is in a state where $\mathcal{M} = \emptyset$ and the subsets $\mathcal{B}_j$ form the blocks of the given input array. Next we derive bounds on the number of different states the algorithm must be able to reach, and on the increase in the number of reachable states when we perform a read or a write operation.

- We first derive a bound on the number of different states the algorithm must be able to reach to be able to produce all possible permutations. For any given input—that is, any given set of positions $pos_0, \ldots, pos_{n-1}$—there is one output state, namely where $\mathcal{M} = \emptyset$ and each $\mathcal{B}_j$ contains the elements with $jB \leqslant pos_i \leqslant (j+1)B - 1$. However, not every different input leads to a different output state, because we do not distinguish between different orderings of the elements within the same block. Observe that the number of different permutations we can make when we fix the sets $\mathcal{B}_0, \ldots, \mathcal{B}_{n/B-1}$ is $(B!)^{n/B}$, since for each set $\mathcal{B}_j$ we can pick from $B!$ orderings. Thus the $n!$ different inputs actually correspond to only $n!/(B!)^{n/B}$ different output states.

- Now consider a read operation. It transfers one block from external memory, thus making one of the $\mathcal{B}_j$'s empty and adding its elements to $\mathcal{M}$. The algorithm can choose at most n/B different blocks for this, so the total number of different states the algorithm can be in after a read increase by a factor of at most n/B.

- In a write operation the algorithm chooses B elements from the internal memory, forms a block out of these elements, and writes the block to one of the at most n/B available (currently empty) blocks $\mathcal{B}_j$. The number of different subsets of B elements that the algorithm can choose is $\binom{M}{B}$ and the number of possibilities to choose $\mathcal{B}_j$ is at most n/B. Hence, a write operation increases the number of different states by a factor $\binom{M}{B} \cdot (n/B)$.

Now let X_r denote the number of read operations the algorithm performs in the worst case, and let X_w denote the number of write operations the algorithm performs in the worst case. By the above, the algorithm can then reach at most

$$\left(\frac{n}{B}\right)^{X_\mathrm{r}} \cdot \left(\binom{M}{B} \cdot \frac{n}{B}\right)^{X_\mathrm{w}}$$

different states, and this number must be at least $n!/(B!)^{n/B}$ for the algorithm to function correctly on all possible inputs. Since both initially and at the end all elements are in external memory, we must have $X_\mathrm{r} = X_\mathrm{w} = X/2$. Hence,

$$\begin{aligned}\tfrac{n!}{(B!)^{n/B}} &\leqslant \left(\tfrac{n}{B}\right)^{X_\mathrm{r}} \cdot \left(\tbinom{M}{B} \cdot \tfrac{n}{B}\right)^{X_\mathrm{w}} \\ &= \left(\tfrac{n}{B}\right)^{X/2} \cdot \left(\tbinom{M}{B} \cdot \tfrac{n}{B}\right)^{X/2} \\ &= \left(\tfrac{n}{B}\right)^{X} \cdot \tbinom{M}{B}^{X/2}.\end{aligned}$$

It remains to show that this inequality leads to the claimed lower bound on X. The above implies that

$$X \cdot \log\left(\frac{n}{B} \cdot \binom{M}{B}^{1/2}\right) \geqslant \log\left(\frac{n!}{(B!)^{n/B}}\right). \tag{2.3}$$

We first bound the right-hand side of Inequality (2.3). From Stirling's approximation, which states that $n! \sim \sqrt{2\pi n}(n/e)^n$, it follows that $\log(n!) = n\log(n/e) + O(\log n)$. Hence, the right-hand side in (2.3) can be bounded as

$$\begin{aligned}\log\left(\tfrac{n!}{(B!)^{n/B}}\right) &= \log(n!) - \log\left((B!)^{n/B}\right) \\ &= n\log(n/e) + O(\log n) - (n/B)\log(B!) \\ &= n\log(n/e) + O(\log n) - (n/B)\left(B\log(B/e) + O(\log B)\right) \\ &= n\log(n/B) + O(\log n) - O((n/B)\log B) \\ &= \Omega(n\log(n/B)).\end{aligned}$$

Now consider the factor $\log\left(\frac{n}{B} \cdot \binom{M}{B}^{1/2}\right)$ in the left-hand side of (2.3). Using the condition in the theorem that $n < B\sqrt{\binom{M}{B}}$ and using the inequality $\binom{M}{B} \leqslant \left(\frac{eM}{B}\right)^B$, we obtain

$$\begin{aligned}\log\left(\tfrac{n}{B} \cdot \tbinom{M}{B}^{1/2}\right) &< \log\tbinom{M}{B} \\ &\leqslant B\log\left(\tfrac{eM}{B}\right) \\ &< 2B\log\left(\tfrac{M}{B}\right).\end{aligned}$$

Hence,

$$\begin{aligned}X &\geqslant \frac{\log\left(\frac{n!}{(B!)^{n/B}}\right)}{\log\left(\frac{n}{B}\cdot\binom{M}{B}^{1/2}\right)} \\ &> \frac{\Omega(n\log(n/B))}{2B\log(M/B)} \\ &= \Omega((n/B)\log_{M/B}(n/B)).\end{aligned}$$

□

Theorem 2.2 has the condition that $n < B\sqrt{\binom{M}{B}}$. This condition is satisfied for all reasonable values of n, M, and B since then $\binom{M}{B}$ is extremely large.

2.3 Exercises

Exercise 2.1 Consider the recurrence for the number of I/Os performed by (the unmodified version of) *MergeSort*:

$$T_{\text{IO}}(n) \leqslant \begin{cases} T_{\text{IO}}(\lfloor n/2 \rfloor) + T_{\text{IO}}(\lceil n/2 \rceil) + O(n/B) & \text{if } n > M/2 \\ O(M/B) & \text{otherwise} \end{cases}$$

Prove that $T_{\text{IO}}(n) = O((n/B)\log_2(n/M))$. You may ignore rounding issues, and replace $T_{\text{IO}}(\lfloor n/2 \rfloor) + T_{\text{IO}}(\lceil n/2 \rceil)$ by $2T_{\text{IO}}(n/2)$ and use the recurrence-tree method.

Exercise 2.2 Show that $\log_{M/B}(n/M) = \Theta(\log_{M/B}(n/B))$.

Exercise 2.3 Analyze the running time (not the number of I/Os) of algorithm *EM-MergeSort*. Does it still run in $O(n \log n)$ time? If not, explain how to implement the merge step to make it run in $O(n \log n)$ time.

Exercise 2.4 In this chapter we saw an I/O-efficient version of *MergeSort*, a standard algorithm for sorting a set of n numbers in $O(n \log n)$ time. Another standard sorting algorithm is *QuickSort*.

QuickSort(A) ▷ $A[0..n-1]$ is an array of distinct numbers
1: **if** $n > 1$ **then** ▷ else A is sorted by definition
2: Compute an element $A[i^*]$ such that the number of elements in A smaller than or equal to $A[i^*]$ is $\lceil n/2 \rceil$ and the number of elements in A larger than $A[i^*]$ is $\lfloor n/2 \rfloor$. (Thus $A[i^*]$ is a median of the elements in A.)
3: Scan A and write all elements smaller than or equal to $A[i^*]$ to an array B_1 and write all elements larger than $A[i^*]$ to an array B_2.
4: *QuickSort*(B_1); *QuickSort*(B_2)
5: Scan B_1 and write its contents to the first $\lceil n/2 \rceil$ positions in A, and then scan B_2 and write its contents to the last $\lfloor n/2 \rfloor$ positions in A.
6: **end if**
7: **return** A.

In this exercise we consider the I/O-performance of *QuickSort*.

(i) Assume that Step 2 can be performed in $O(n/B)$ I/Os. Give a recurrence for $T_{IO}(n)$, the number of I/Os that *QuickSort* performs when run on an array of size n. Make sure you use a suitable base case in your recurrence.

(ii) Prove that the recurrence you gave in part (i) solves to $T_{IO} = O((n/B)\log_2(n/M))$.

(iii) The number of I/Os performed by *QuickSort* is $O((n/B)\log_2(n/M))$ under the assumption that we can perform Step 2 with $O(n/B)$ I/Os, which is not so easy. Therefore one often uses a randomized version of *QuickSort*, where the pivot element $A[i^*]$ is chosen randomly. (More precisely, the index i^* is chosen from the set $\{0, \ldots, n-1\}$ uniformly at random.) Intuitively, one would expect the pivot to be close enough to the median, and indeed one can show that the expected number of I/Os is $O((n/B)\log_2(n/M))$. This randomized version of *QuickSort* is nice and simple, but its I/O-performance is

sub-optimal because the logarithm has base 2. Describe how to modify the randomized version of *QuickSort* such that its I/O-performance is improved. The goal would be to obtain a randomized version of *QuickSort* for which the expected number of I/Os is $O(\text{SORT}(n))$.

NB: Keep your explanation brief. A short description of your modification and a few lines of explanation why the expected number of I/Os would hopefully be $O(\text{SORT}(n))$ suffices.

Exercise 2.5 Let $A[0 \ldots n-1]$ be an array of n distinct numbers. The *rank* of an element in $A[i]$ is defined as follows:

$$\text{rank}(A[i]) := (\text{number of elements in } A \text{ that are smaller than } A[i]) + 1.$$

Define the *displacement* of $A[i]$ to be $|i - \text{rank}(A[i]) + 1|$. Thus the displacement of $A[i]$ is equal to the distance between its current position in the array (namely i) and its position when A is sorted.

(i) Suppose we know that A is already "almost" sorted, in the sense that the displacement of any element in A is less than $M - 2B$. Give an algorithm that sorts A using $O(n/B)$ I/Os, and argue that it indeed performs only that many I/Os.

(ii) The $O(n/B)$ bound on the number of I/Os is smaller than the $\Omega((n/B)\log_{M/B}(n/B))$ lower bound from Theorem 2.2. Apparently the lower bound does not hold when the displacement of any element in A is less than $M - 2B$. Explain why the proof of Theorem 6.2 does not work in this case.
NB: You can keep your answer short—a few lines is sufficient—but you should point to a specific place in the proof where it no longer works.

Exercise 2.6 Let $X[0..n-1]$ and $Y[0..n-1]$ be two arrays, each storing a set of n numbers. Let $Z[0..n-1]$ be another array, in which each entry $Z[i]$ has three fields: $Z[i].x$, $Z[i].y$ and $Z[i].sum$. The fields $Z[i].x$ and $Z[i].y$ contain integers in the range $\{0, \ldots, n-1\}$; the fields $Z[i].sum$ are initially empty. We wish to store in each field $Z[i].sum$ the value $X[Z[i].x] + Y[Z[i].y]$. A simple algorithm for this is as follows.

ComputeSums(X, Y, Z)
1: **for** $i \leftarrow 0$ **to** $n-1$ **do**
2: $\quad Z[i].sum \leftarrow X[Z[i].x] + Y[Z[i].y]$
3: **end for**

(i) Analyze the number of I/Os performed by *ComputeSums*.

(ii) Give an algorithm to compute the values $Z[i].sum$ that performs only $O(\text{SORT}(n))$ I/Os. At the end of your algorithm the elements in arrays X, Y, and Z should still be in the original order.

Exercise 2.7 In the proof of Theorem 2.2 we assumed that the algorithm only uses the blocks of the input array A, it does not use any additional storage in the external memory. Prove that without this assumption the same asymptotic lower bound holds.

Exercise 2.8 Consider the permutation lower bound of Theorem 2.2.

(i) The lower bound holds when $n \leqslant B(eM/B)^{B/2}$. Show that this condition is satisfied for any practical values of n, M, and B. To this end, pick reasonable values of M and B, and then compute how large n should be to violate the condition.

(ii) Prove that $\Omega(n)$ is a lower bound on the number of I/Os when $n > B(eM/B)^{B/2}$.

Exercise 2.9 The *MergeSort* algorithm can sort any set of n numbers in $O(n \log n)$ time. As we have seen, *MergeSort* can be adapted such that it can sort any set of n numbers in $O((n/B) \log_{M/B}(n/B))$ I/Os. For the special case where the input consists of integers in the range $1, \ldots, n^c$, for some fixed constant c, there exist sorting algorithms that run in $O(n)$ time. Would it be possible to adapt such an algorithm so that it can sort any set of n integers in the range $1, \ldots, n^c$ in $O(n/B)$ I/Os? Explain your answer.

Chapter 3

Buffer trees and time-forward processing

In this chapter we study time-forward processing, a simple but powerful technique to design I/O-efficient algorithms. Time-forward processing needs an I/O-efficient priority queue. Hence, we first present such a priority queue, which is based on a so-called buffer tree.

3.1 Buffer trees and I/O-efficient priority queues

A *dictionary* is a data structure for storing a set S of elements, each with a *key*, in which one can quickly *search* for an element with a given key. In addition, dictionaries support the *insertion* and *deletion* of elements. One of the standard ways to implement a dictionary in internal memory is by a balanced binary search tree—a red-black tree, for instance—which supports these three operations in $O(\log n)$ time. In external memory a binary search tree is not very efficient, because the worst-case number of I/Os per operation is $O(\log n)$ if the nodes are grouped into blocks in the wrong way. A more efficient alternative is a so-called *B-tree*. B-trees are also search trees, but they are not binary: internal nodes have $\Theta(B)$ children (instead of just two), where the maximum degree is chosen such that a node fits into one memory block. More precisely, the (internal and leaf) nodes in a B-tree store between $d_{\min}-1$ and $2d_{\min}-1$ keys. Hence, the degree of the internal nodes is between $d_{\min}$ and $2d_{\min}$; the only exception is the root, which is allowed to store between 1 and $2d_{\min}-1$ keys. All leaves are at the same depth in the tree, as illustrated in Fig. 3.1. The value $d_{\min}$ should be

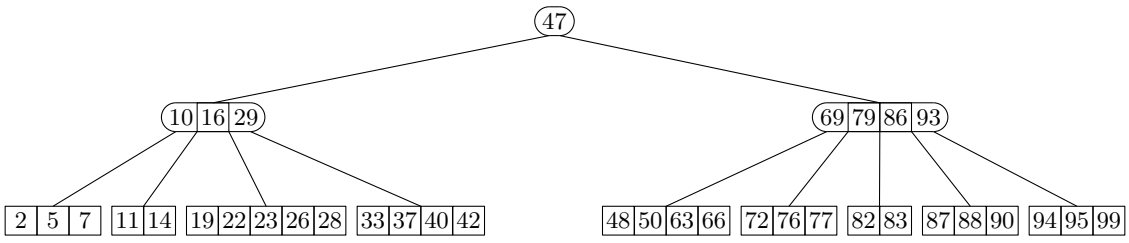

Fig. 3.1: Example of a B-tree. For simplicity the figure shows a B-tree with $d_{\min} = 3$, but in practice $d_{\min}$ is typically (at least) about one hundred.

chosen as large as possible but such that any node still fits into one block in external memory. This means that $d_{\min} = \Theta(B)$.

Because B-trees are balanced—all leaves are at the same level—and the internal nodes have degree $\Theta(B)$, B-trees have depth $O(\log_B n)$. Hence, a search operation can be done in $O(\log_B n)$ I/Os. Also insertions and deletions can be performed with only $O(\log_B n)$ I/Os. The change in the base of the logarithm as compared to binary search trees makes B-trees much more efficient in external memory.

The $O(\log_B n)$ worst-case number of I/Os per operation (search, insert, or delete) is essentially optimal. However, in many applications we have to perform many operations, and we care more about the total number of I/Os than about the worst-case number of I/Os for an individual operation. In such cases the *buffer tree* can be a more efficient solution. The buffer-tree technique can be used for several different purposes. Here we describe how to apply it to obtain an I/O-efficient priority queue.

The basic buffer tree. Let S be a set of elements, where each element $x \in S$ has a key $key(x)$. Suppose we are given a sequence of n operations $op_0, \ldots, op_{n-1}$ on S. Typically the sequence contains a mixture of insert-, delete- and query-operations; the type of query operations being supported depends on the particular structure being implemented by the buffer tree. The goal is to process each of the operations and, in particular, to answer all the queries. However, we do not have to process the operations one by one: we are allowed to first collect some operations and then execute them in a *batched* manner. Next we describe how a buffer tree exploits this. First, we ignore the queries and focus on the insert- and delete-operations. After that we will show how to deal with queries for the case where the buffer tree implements a priority queue.

The basic buffer tree—see also Fig. 3.2—is a tree structure $\mathcal{T}_{\text{buf}}$ with the following properties:

(i) $\mathcal{T}_{\text{buf}}$ is a search tree with respect to the keys in S, with all leaves at the same level.

(ii) Each leaf of $\mathcal{T}_{\text{buf}}$ contain $\Theta(B)$ elements and fits into one block.

(iii) Each internal node of $\mathcal{T}_{\text{buf}}$, except for the root, contains between $d_{\min} - 1$ and $4d_{\min} - 1$ elements, where $d_{\min} = \Theta(M/B)$; the root contains between 1 and $4d_{\min} - 1$ elements.

(iv) Each internal node ν has a *buffer* $\mathcal{B}_\nu$ of size $\Theta(M)$ associated to it, which stores operations that have not yet been fully processed. Each operation has a *time-stamp*, which indicates its position in the sequence of operations: the first operation gets time stamp 0, the next operation gets time stamp 1, and so on.

If we consider only properties (i)–(iii) then a buffer tree is very similar to a B-tree, except that the degrees of the internal nodes are much larger: instead of degree $\Theta(B)$ they have degree $\Theta(M/B)$. Thus a single node no longer fits into one block. This means that executing a single operation is not very efficient, as even accessing one node is very costly. The buffers solve this problem: they collect operations until the number of still-to-be-processed operations is large enough that we can afford to access the node. Besides the buffer tree $\mathcal{T}_{\text{buf}}$ itself, which is stored in external memory, there is one block b_{buf} in internal memory that is used to collect unprocessed operations. Once b_{buf} is full, the operations in it are inserted into $\mathcal{T}_{\text{buf}}$. After the operations from b_{buf} have been inserted into $\mathcal{T}_{\text{buf}}$ we start filling up b_{buf} again, and so on. Inserting a batch of $\Theta(B)$ operations from b_{buf} into $\mathcal{T}_{\text{buf}}$ is done as follows.

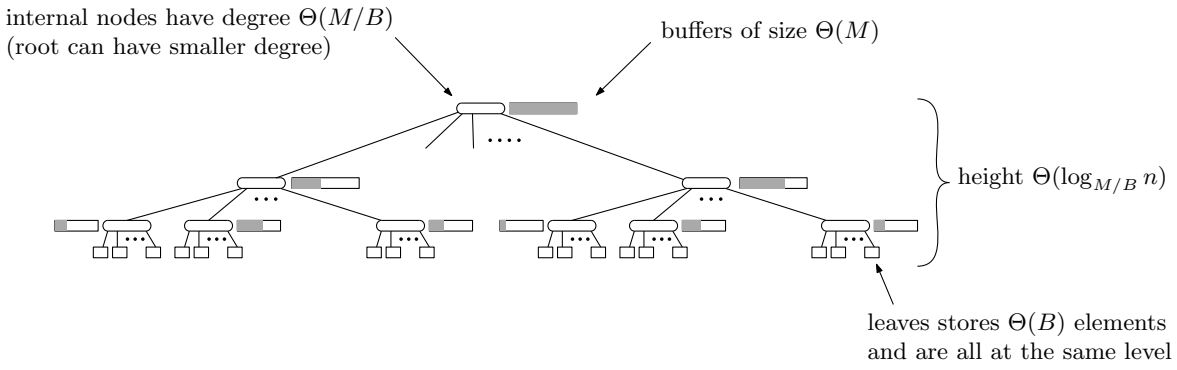

Fig. 3.2: A buffer tree. The grey bars inside the buffers indicate how full they are. Note that the buffer of the root is completely full. Thus the next insertion into the buffer will cause it to be flushed.

- If the buffer $\mathcal{B}_{root}$ of the root still has space left to store the operations in b_{buf}, these operations are written to $\mathcal{B}_{root}$.
- If the buffer of the root overflows, it is *flushed*. This means that it contents are pushed down into the buffers of the appropriate children of the root. These buffers could in turn overflow, which causes them to be flushed as well, and so on. More precisely, flushing an internal node ν is done recursively as follows.
  - Load the buffer $\mathcal{B}_\nu$ into internal memory and sort the operations by their key.
  - Simultaneously scan the sorted list of operations and the keys stored in ν, and push the operations down into the appropriate children. (The child to which an operation involving key x should be pushed is the next node on the search path to x.) Pushing an operation down into a child means that we add the operation to the buffer of that child. This is done in a batched manner: for each child μ we collect the operations that should be pushed down into μ in groups of $\Theta(B)$ operations, which we write as one block to $\mathcal{B}_\mu$. (The last block of operations to be pushed down into μ could contains fewer than $\Theta(B)$ operations.)
  - For each child of ν whose buffer has overflown due to the operations being pushed down from ν, flush its buffer using the same procedure.

  For internal nodes that are just above the leaf level, the flushing procedure is slightly different. For such a node ν, we load its buffer into internal memory, together with all the leaves below ν. (Recall that ν has degree $\Theta(M/B)$ and leaves have size B, so all of this fits in internal memory.) Let S' be the set of elements stored in these leaves. We then perform the operations from $\mathcal{B}_\nu$ on S'. Finally, we construct a subtree for the new set S', which replaces the old subtree (consisting of ν and the leaves below it).

This finishes the sketch of the buffer tree. We have swept several important details under the rug. In particular, when flushing a node just above leaf level and constructing a subtree for the new set S', we may find that S' contains too many or too few elements to keep properties (ii) and (iii). Thus we have to re-balance the tree. This can be done within the same I/O-bounds; we omit the details.

How many I/Os does a buffer tree perform to execute all operations from the given sequence? Clearly the process of flushing a buffer is quite expensive, since it requires loading the entire buffer into internal memory. However, flushing also involves many operations, and the number of I/Os needed to load (and write back) a full buffer $\mathcal{B}_\nu$ is $O(|\mathcal{B}_\nu|/B)$. In other words, each operation in $\mathcal{B}_\nu$ incurs a cost of $O(1/B)$ I/Os when it is in a buffer being flushed. When this happens the operation is moved to a buffer at a lower level in the tree, and since the depth of the tree is $O(\log_{M/B}(n/B))$, the total cost incurred by an operation is $O((1/B)\log_{M/B}(n/B))$. The total number of I/Os over all n operations is therefore $O((n/B)\log_{M/B}(n/B)) = O(\text{SORT}(n))$. (Again, we sweep several details under the rug, about how nodes just above leaf level are handled and about the re-balancing that is needed.)

An I/O-efficient priority queue based on buffer trees. Now let's see how we can use the buffer-tree technique to implement an I/O-efficient priority queue. Let $S = \{x_1, \ldots, x_n\}$ be a set of elements, where each element x_i has a priority $prio(x_i)$. A *min-priority queue* on S is an abstract data structure that supports the following operations:[1]

- *Extract-Min*, which reports an element $x_i \in S$ with the smallest priority and removes the element from S.
- *Insert*(x), which inserts a new element x with given priority $prio(x)$ into S.

(A max-priority queue is similar, except that it supports an operation that extracts the element of maximum priority.) One way to implement a priority queue in internal memory is to use a heap. Another way is to use a binary search tree, where the priorities play the role of the keys, that is, the priorities determine the left-to-right order in the tree. An *Extract-Min* operation can then be performed by deleting the leftmost leaf of the tree. Since a buffer tree provides us with an I/O-efficient version of a search tree, it seems we can directly use the buffer tree as an I/O-efficient priority queue. Things are not that simple, however, because the smallest element is not necessarily stored in the leftmost leaf in a buffer tree—there could be an insert-operation of a smaller element stored in one of the buffers. Moreover, in many applications it is not possible to batch the *Extract-Min* operations: priority queues are often used an event-driven algorithms that need an immediate answer to the *Extract-Min* operation, otherwise the algorithm cannot proceed. These problems can be overcome as follows.

Whenever we receive an *Extract-Min* we flush all the nodes on the path to the leftmost leaf. We then load the $M/4$ smallest elements from $\mathcal{T}_{\text{buf}}$ into internal memory—all these elements are stored in the leftmost internal node and its leaf children—and we delete them from $\mathcal{T}_{\text{buf}}$. Let S^* be this set of elements. The next $M/4$ operations can now be performed on S^*, without having to do a single I/O: when we get an *Extract-Min*, we simply remove (and report) the smallest element from S^* and when we get an *Insert* we just insert the element into S^*. Because we perform only $M/4$ operations on S^* in this manner, and S^* initially has size $M/4$, the answers to the *Extract-Min* operations are correct: an element that is still in $\mathcal{T}_{\text{buf}}$ cannot become the smallest element before $M/4$ *Extract-Min* operations have taken place. Moreover, the size of S^* does not grow beyond $M/2$, so S^* can indeed be kept in internal memory.

After processing $M/4$ operations in this manner, we empty S^* by inserting all its elements in the buffer tree. When the next *Extract-Min* arrives, we repeat the process: we flush all

[1] Some priority queues also support a *Decrease-Key*(x, Δ) operation, which decreases the priority of the element $x \in S$ by Δ. We will not consider this operation.

nodes on the path to the leftmost leaf, we delete the $M/4$ smallest elements and load them into internal memory, thus obtaining a set S^* on which the next $M/4$ operations are performed.

We need $O((M/B)\log_{M/B}(n/B))$ I/Os to flush all buffers on the path to the leftmost leaf. These I/Os can be charged to $M/4$ operations preceding the flushing, so each operation incurs a cost of $O((1/B)\log_{M/B}(n/B))$ I/Os. This means that the total number of I/Os over n *Insert* and *Extract-Min* operations is still $((n/B)\log_{M/B}(n/B))$. We get the following theorem.

Theorem 3.1 *There is an I/O-efficient priority queue that can process any sequence of n* Insert *and* Extract-Min *operations using $((n/B)\log_{M/B}(n/B))$ I/Os in total.*

3.2 Time-forward processing

Consider an *expression tree* $\mathcal{T}$ with n leaves, where each leaf corresponds to a number and each internal node corresponds to one of the four standard arithmetic operations $+$, $-$, $*$, $/$. Fig. 3.3(i) shows an example of an expression tree. Evaluating an expression tree in linear time is easy in internal memory: a simple recursive algorithm does the job. Evaluating it in an I/O-efficient manner is much more difficult, however, in particular when we have no control over how the tree is stored in external memory. The problem is that each time we follow a pointer from a node to a child, we might have to do an I/O. In this section we describe a simple and elegant technique to overcome this problem; it can evaluate an expression tree in $O(\textsc{Sort}(n))$ I/Os. The technique is called *time-forward processing* and it applies in a much more general setting, as described next.

Let $\mathcal{G} = (V, E)$ be a directed, acyclic graph (DAG) with n nodes, where each node $v_i \in V$ has a label $\lambda(v_i)$ associated to it. The goal is to compute a recursively defined function f on the nodes that satisfies the following conditions. Let $N_{\text{in}}(v_i)$ be the set of in-neighbors of v_i, that is, $N_{\text{in}}(v_i) := \{v_j : (v_j, v_i) \in E\}$.

- When $|N_{\text{in}}(v_i)| = 0$ then $f(v_i)$ depends only on $\lambda(v_i)$. Thus when v_i is a source in $\mathcal{G}$ then $f(v_i)$ can be computed from the label of v_i itself only.
- When $|N_{\text{in}}(v_i)| > 0$ then $f(v_i)$ can be computed from $\lambda(v_i)$ and the f-values of the in-neighbors of v_i.

We will call a function f satisfying these conditions a *local function* on the DAG $\mathcal{G}$. Note that if we direct all edges of an expression tree towards the root then we obtain a DAG; evaluating the expression tree then corresponds to computing a local function on the DAG. Local functions on DAGs can easily be computed in linear time in internal memory. Next we show to do this I/O-efficiently if the vertices of the DAG are stored in topological order in external memory. More precisely, we assume that $\mathcal{G}$ is stored as follows.

Let $v_0, \ldots, v_{n-1}$ be a topological order on $\mathcal{G}$. Thus, if $(v_i, v_j) \in E$ then $i < j$. We assume $\mathcal{G}$ is stored in an array $A[0..n-1]$, where each entry $A[i]$ stores the label $\lambda(v_i)$ as well a list containing the indices of the out-neighbors of v_i; see Fig. 3.3(ii). We denote this value and list by $A[i].\lambda$ and $A[i].\text{OutNeighbors}$, respectively. Blocks are formed as illustrated in the figure.

We will use an array $F[0..n-1]$ to store the computed f-values, so that at the end of the computation we will have $F[i] = f(v_i)$ for all i. Now suppose we go over the nodes $v_0, \ldots, v_{n-1}$ in order. (Recall that this is a topological order.) In internal memory we could keep a list for each node v_j that stores the already computed f-values of its in-neighbors.

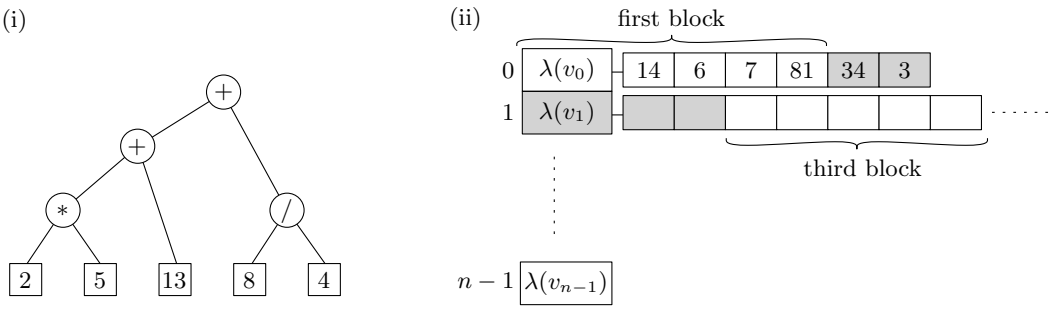

Fig. 3.3: (i) An expression tree. The value of the tree is 25. (ii) Representation of a vertex-labeled DAG in external memory. Vertex v_1 has edges to v_{14}, v_6, v_7, v_{81}, v_{34}, and v_3. The way in which the representation would be blocked for $B = 5$ is also indicated. (The grey elements form the second block.)

When we handle a node v_i, its list will contain the f-values of all its in-neighbors since these have been handled before v_i. Hence, we can compute $f(v_i)$, store the value in $F[i]$ and insert it into the lists of all out-neighbors of v_i. The latter step is expensive in external memory, because we may have to spend an I/O to access each of these lists. Thus we may spend one I/O per edge in $\mathcal{G}$, which means the algorithm would not be I/O-efficient.

In time-forward processing we therefore proceed differently. Instead of keeping separate lists for each of the nodes, we maintain one pool of f-values that we still need in the future. More precisely, we will keep an I/O-efficient min-priority queue $\mathcal{Q}$ storing pairs $(f(v_i), j)$ where j is such that v_j is an out-neighbor of v_i. The value j is the priority of the pair $(f(v_i), j)$; it serves as a time-stamp so that we can extract $f(v_i)$ at the time it is needed. Whenever we compute a value $f(v_i)$, we insert a pair $(f(v_i), j)$ into Q for every out-neighbor v_j of v_i. This leads to the following algorithm.

Algorithm 3.1 Computing a local function on a DAG with time-forward processing.

TimeForward-DAG-Evaluation(G)
1: Initialize an array $F[0..n-1]$ and a min-priority queue $\mathcal{Q}$.
2: **for** $i \leftarrow 0$ **to** $n-1$ **do**
3: Perform *Extract-Min* operations on $\mathcal{Q}$ as long as the extracted pairs have priority i. (This requires one more *Extract-Min* than there are pairs with priority i. The extra extracted pair, which has priority i' for some $i' > i$, should be re-inserted into $\mathcal{Q}$.)
4: Compute $f(v_i)$ from $A[i].\lambda$ and the extracted f-values; set $F[i] \leftarrow f(v_i)$.
5: **for** each j in $A[i]$.OutNeighbors **do**
6: Insert the pair $(f(v_i), j)$ into $\mathcal{Q}$.
7: **end for**
8: **end for**

How the value $f(v_i)$ is computed in Step 4 depends on the specific function f. Often the computation is straightforward and scanning the f-values of the in-neighbors of v_i is sufficient to compute $f(v_i)$. The computation could also be more complicated, however. If the number of in-neighbors of v_i is so large that their f-values do not all fit into main memory,

then computing $f(v_i)$ could even become the I/O bottleneck of the algorithm. Therefore we assume in the theorem below that $f(v_i)$ can be computed in $O(\text{SORT}(1+|N_{\text{in}}(v_i)|))$ I/Os. Note then when each vertex has less than M in-neighbors, which is usually the case in practice, this condition is trivially satisfied.

Theorem 3.2 *Let $\mathcal{G} = (V, E)$ be a DAG stored in topological order in external memory, using an adjacency-list representation. Let f be a local function on $\mathcal{G}$ such that each $f(v_i)$ can be computed in $O(\text{SORT}(1+|N_{\text{in}}(v_i)|))$ I/Os from $\lambda(v_i)$ and the f-values of the in-neighbors of v_i. Then we can compute the f-values of all nodes in $\mathcal{G}$ using $O(\text{SORT}(|V| + |E|))$ I/Os in total.*

Proof. We first prove that algorithm *TimeForward-DAG-Evaluation* correctly computes f. We claim that when v_i is handled, the f-values of the in-neighbors of v_i are all stored in $\mathcal{Q}$, and that they have the lowest priority of all elements currently in $\mathcal{Q}$. This claim is sufficient to establish correctness.

Define the *level* of a node v_i to be the length of the longest path in $\mathcal{G}$ that ends at v_i. We will prove the claim by induction on the level of the nodes. The nodes of level 0 are the nodes in $\mathcal{G}$ without in-neighbors; for these nodes the claim obviously holds. Now consider a node v_i with level $\ell > 0$. The in-neighbors of v_i have already been handled when we handle v_i, because the nodes are handled in topological order. Moreover, their level is smaller than ℓ and so by the induction hypothesis their f-values have been computed correctly. Hence, these f-values are present in $\mathcal{Q}$ when v_i is handled. Moreover, there cannot be any pair $(f(v_k), i')$ with $i' < i$ in $\mathcal{Q}$, since such a pair would have been extracted when $v_{i'}$ was handled. This proves the claim for nodes at level $\ell > 0$.

To prove the I/O bound, we observe that we perform $O(|V|+|E|)$ *Extract-Min* operations and $O(|V|+|E|)$ *Insert* operations on $\mathcal{Q}$. By Theorem 3.1 these operations can be performed using $O(((|V|+|E|)/B)\log_{M/B}(|V|+|E|))$ I/Os in total. The total number of I/Os needed for the computations of the f-values is $\sum_{i=0}^{n-1} O(\text{SORT}(1+|N_{\text{in}}(v_i)|))$, which is $O(\text{SORT}(|V|+|E|))$ since $\sum_{i=0}^{n-1} |N_{\text{in}}(v_i)| = |E|$. □

The condition that $\mathcal{G}$ is stored in topological order seems rather restrictive. Indeed, topologically sorting a DAG in external memory is difficult and there are no really I/O-efficient algorithms for it. However, in many applications it is reasonable to assume that the DAG is generated in topological order, or that the DAG has such a simple structure that topological sorting is easy. This is for instance the case when evaluating an expression tree. Note that the second condition of Theorem 3.2, namely that the f-value of a node can be computed in $O(\text{SORT}(1 + |N_{\text{in}}(v_i)|))$ I/Os, is trivially satisfied since $|N_{\text{in}}(v_i)| \leqslant 2$ in a binary expression tree.

Interestingly, evaluating a local function on DAG can also be used for a number of problems on undirected graphs. We give one example of such an application. Recall that an *independent set* of a graph $\mathcal{G} = (V, E)$ is a subset $I \subseteq V$ such that no two nodes in I are connected by an edge. An independent set is called *maximal* if no node can be added to I without losing the independent-set property. (In other words, each node in $V \setminus I$ has an edge to some node in I.) Computing a maximal independent set can be done by applying Theorem 3.2, as shown next. We assume that the input graph $\mathcal{G}$ is stored in an adjacency-list representation as above (including the blocking scheme), except that we do not assume that $v_0, \ldots, v_{n-1}$ is a topological order—indeed, $\mathcal{G}$ is undirected so the concept of topological order does not apply to $\mathcal{G}$. Note that in an adjacency-list representation of an undirected graph, each edge

$(v_i, v_j) \in E$ will be stored twice: once in the adjacency list of v_i and once in the adjacency list of v_j.

Corollary 3.3 *Let $\mathcal{G} = (V, E)$ be an undirected graph, stored in an adjacency-list representation in external memory. We can compute a maximal independent set for $\mathcal{G}$ in $O(\text{SORT}(|V| + |E|))$ I/Os.*

Proof. We first turn $\mathcal{G}$ into a directed graph $\mathcal{G}^*$ by directing every edge (v_i, v_j) from the node with smaller index to the node with higher index. Observe that $\mathcal{G}^*$ is a DAG—there cannot be cycles in $\mathcal{G}^*$. Moreover, the representation of $\mathcal{G}$ in fact corresponds to a representation for $\mathcal{G}^*$ in which the nodes are stored in topological order by definition. Thus, to obtain a suitable representation for $\mathcal{G}^*$ we do not have to do anything: we can just interpret the representation of $\mathcal{G}$ as a representation of $\mathcal{G}^*$ (where we ignore the "reverse edges" that $\mathcal{G}$ stores).

Now define a function f on the nodes as follows. (In this application, we do not need to introduce labels $\lambda(v_i)$ to define f.)

- If $|N_{\text{in}}(v_i)| = 0$ then $f(v_i) = 1$.
- If $|N_{\text{in}}(v_i)| > 0$ then

$$f(v_i) = \begin{cases} 0 & \text{if } v_i \text{ has at least one in-neighbor } v_j \text{ with } f(v_j) = 1 \\ 1 & \text{otherwise.} \end{cases}$$

It is easy to see that the set $I := \{v_i : f(v_i) = 1\}$ forms a maximal independent set. Note that f is a local function, and that each $f(v_i)$ can be computed in $O(\text{SORT}(1 + |N_{\text{in}}(v_i)|))$ I/Os—in fact, in $O(\text{SCAN}(1 + |N_{\text{in}}(v_i)|))$ I/Os—from the f-values of its in-neighbors. Hence, we can apply Theorem 3.2 to obtain the result. □

3.3 Exercises

Exercise 3.1 Consider a balanced binary search tree $\mathcal{T}$ with n nodes that is stored in external memory. In this exercise we investigate the effect of different blocking strategies for the nodes of $\mathcal{T}$.

(i) Suppose blocks are formed according to an in-order traversal of $\mathcal{T}$ (which is the same as the sorted order of the values of the nodes). Analyze the minimum and maximum number of I/Os needed to traverse any root-to-leaf path in $\mathcal{T}$. Make a distinction between the case where the internal memory can store only one block and the case where it can store more than one block.

(ii) Describe an alternative way to form blocks, which guarantees that any root-to-leaf path can be traversed using $O(\log_B n)$ I/Os. What is the relation of your blocking strategy to B-trees?

Exercise 3.2 Let $\mathcal{T}$ be a balanced binary search tree $\mathcal{T}$ with n nodes.

(i) Consider a blocking strategy—that is, a strategy to group the nodes from $\mathcal{T}$ into blocks with at most B nodes each— with the following property: for each block b, the nodes in b form a connected part of $\mathcal{T}$. Note that this implies that whenever a root-to-leaf path leaves a block, it will not re-enter that block. Prove that for such a blocking strategy there is always a root-to-leaf path that visits $\Omega(\log_B n)$ blocks.

(ii) Now prove that for *any* blocking strategy, there is a root-to-leaf path that visits $\Omega(\log_B n)$ blocks. *Hint:* Argue that any grouping of nodes into blocks can be modified into a grouping that satisfies the condition in (i), without increasing the number of visited blocks by any root-to-leaf path.

Exercise 3.3 Let $\mathcal{T}$ be a balanced binary search tree. We want to groups the nodes from $\mathcal{T}$ into blocks such that any root-to-leaf path accesses only $O(\log_B n)$ blocks, as in Exercise 3.1(ii). This time, however, your blocking strategy should be cache-oblivious, that is, it cannot use the value B. (Thus you cannot say "put these B nodes together in one block".) In other words, you have to define a numbering of the nodes of $\mathcal{T}$ such that blocking according to this numbering—putting nodes numbered $1, \ldots, B$ into one block, nodes numbered $B+1, \ldots, 2B$ into one block, and so on—gives the required property for any value of B.
Hint: Partition $\mathcal{T}$ into subtrees of size $\Theta(\sqrt{n})$ in a suitable manner, and use a recursive strategy.

Exercise 3.4 The I/O-efficient priority queue described above keeps a set S^* in internal memory that contains $M/4$ elements when S^* is created. The next $M/4$ operations are then performed on S^*, after which the elements that still remain in S^* are inserted into the buffer tree (and S^* is emptied). Someone suggests the following alternative approach: instead of only performing the next $M/4$ operations on S^*, we keep on performing *Extract-Min* and *Insert* operations on S^* until $|S^*| = M$. Does this lead to correct results? Explain your answer.

Exercise 3.5 Let S be an initially empty set of numbers. Suppose we have a sequence of n operations $op_0, \ldots, op_{n-1}$ on S. Each operation op_i is of the form $(type_i, x_i)$, where $type_i \in \{$*Insert, Delete, Search*$\}$ and x_i is a number. You may assume that when a number x is inserted it is not present in S, and when a number x is deleted it is present. (After a number has been deleted, it could be re-inserted again.) The goal is to report for each of the *Search*-operations whether the number x_i being searched for is present in S at the time of the search. With a buffer tree these operations can be performed in $O(\text{SORT}(n))$ I/Os in total. Show that the problem can be solved more directly (using sorting) in $O(\text{SORT}(n))$ I/Os, without using a buffer tree.

Exercise 3.6 A *coloring* of an undirected graph $\mathcal{G} = (V, E)$ is an assignment of colors to the nodes of $\mathcal{G}$ such that if $(v_i, v_j) \in E$ then v_i and v_j have different colors. Suppose that $\mathcal{G}$ is stored in the form of an adjacency list in external memory. Assume the maximum degree of any node in $\mathcal{G}$ is $d_{\max}$. Give an algorithm that computes a valid coloring for $\mathcal{G}$ that uses at most $d_{\max} + 1$ colors. Your algorithm should perform $O(\text{SORT}(|V| + |E|))$ I/Os.

Exercise 3.7 Let $\mathcal{G} = (V, E)$ be an undirected graph stored in the form of an adjacency list in external memory. A *minimal vertex cover* for $\mathcal{G}$ is a vertex cover C such that no vertex

can be deleted from C without losing the cover property. (In other words, there is no $v \in C$ such that $C \setminus \{v\}$ is also a valid vertex cover). Prove that it is possible to compute a minimal vertex cover for $\mathcal{G}$ using only $O(\text{SORT}(|V| + |E|))$ I/Os.

Exercise 3.8 Let $\mathcal{G} = (V, E)$ be a directed acyclic graph (DAG) stored in the form of an adjacency list in external memory (with blocks formed in the usual way), with the nodes stored in topological order. Define the *level* of a node $v \in V$ to be the length of the longest path in $\mathcal{G}$ that ends at v. For example, if $|N_{\text{in}}(v)| = 0$ (where $N_{\text{in}}(v)$ is the set of in-neighbors of v), then $level(v) = 0$. Show that it is possible to compute the levels of all vertices in V using only $O(\text{SORT}(|V| + |E|))$ I/Os.

Exercise 3.9 Let $\mathcal{G} = (V, E)$ be an undirected graph stored in the form of an adjacency list in external memory (with blocks formed in the usual way), where each node v_i has a label $\lambda(v_i) \in \mathbb{R}$. A *clique* in G is a subset $C \subseteq V$ such that $(u, v) \in E$ for all pairs $u, v \in C$. A clique C is *maximal* if there is no vertex $u \in V \setminus C$ such that $C \cup \{u\}$ is a clique. Show that it is possible to compute a maximal clique in $\mathcal{G}$ using $O(\text{SORT}(|V| + |E|)$ I/Os.

Exercise 3.10 Let $\mathcal{G} = (V, E)$ be an undirected graph stored in the form of an adjacency list in external memory (with blocks formed in the usual way), where each node v_i has a label $\lambda(v_i) \in \mathbb{R}$. We call v_i a *local maximum* if its label is larger than the labels of all its neighbors.

(i) Show that it is possible to compute all local maxima in $\mathcal{G}$ using $O(\text{SORT}(|V| + |E|)$ I/Os by applying Theorem 3.2 twice.

(ii) Give a simple, direct algorithm to compute all local maxima using $O(\text{SORT}(|V| + |E|))$ I/Os.