Homework Exercises on I/O-Efficient Algorithms

The maximum number of points for all exercises is 30. The grade for this homework set is: (number of scored points)/3.

Exercise Set I/O-Efficient I

IO.I-1 (3 points) 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.

Furthermore, 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.

IO.I-2 (1.5+1.5 points) 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)
if $s = m$ **then**
    return "stack overflow"
else
    $A[s] \leftarrow x$; $s \leftarrow s + 1$
end if

Pop(A)
if $s = 0$ **then**
    return "stack is empty"
else
    return $A[s-1]$; $s \leftarrow s - 1$
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.

IO.I-3 $(1 + 1 + 2$ points) 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 n is much larger than M and that $M \geqslant B^2$. In the questions below, make sure you consider both read- and write-operations.

(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.

(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.

(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 $\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 1$, and between columns $\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 1$. 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.

Exercise Set I/O-Efficient II

IO.II-1 (1+1+1 points) 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 in row-major order 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. (In other words, construct an example where the performance ratio of LRU versus optimal caching is M/B.)

(ii) If we make the image only half the size—that is, n becomes half as large as in your example, but M and B stay the same—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?

IO.II-2 (2 points) Consider an algorithm that needs at most $cn/\sqrt{MB}$ I/Os if run with optimal caching, for some constant c. Prove that the algorithm needs at most $c'n/\sqrt{MB}$ I/Os when run with LRU caching, for a suitable constant c'.

Hint: Compare the performance of both versions to running the algorithm with optimal caching on a machine that has only half as much memory.

IO.II-3 (2+1 points) Let $A[0 \dots n-1]$ 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 $M - B$. 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 6.2 from the Course Notes. Apparently the lower bound does not hold when the displacement of any element in A is less than $M - B$. 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.

IO.II-4 (0.5+1.5 points) 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 $\{0,\dots,n-1\}$; 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)
for i ← 0 to n − 1 do
    Z[i].sum ← X[Z[i].x] + Y[Z[i].y]
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 $O(\textsc{Sort}(n))$ I/Os.

Exercise Set I/O-Efficient III

IO.III-1 ($1\frac{1}{2}$+1+$1\frac{1}{2}$ points) Consider a complete binary search tree $\mathcal{T}$ with $n = 2^k - 1$ nodes—that is, a binary search tree with $k = \lceil \log n \rceil$ levels that are all completely filled—which is stored in external memory. In this exercise we investigate the effect of different blocking strategies for the nodes of $\mathcal{T}$, where we assume that the internal memory can hold at least two blocks.

(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 $\mathcal{T}$.

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

(iii) Prove that for *any* blocking strategy—that is, any strategy to group the nodes into blocks with at most B nodes each—there is a root-to-leaf path that visits $\Omega(\log_B n)$ nodes.

IO.III-2 (3 points) Let S be an initially empty set of numbers. Suppose we have a sequence of n operations $op_0, \dots, op_{n-1}$ 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 $O(\textsc{Sort}(n))$ I/Os in total. Show that the problem can be solved more directly (using sorting) in $O(\textsc{Sort}(n))$ I/Os, without using a buffer tree.

IO.III-3 (3 points) 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 $O(\textsc{Sort}(|V| + |E|)$ I/Os.