Assignment 1 - Homework Exercises on Approximation Algorithms

Duy Pham - 0980384
Maciej Wikowski - 0927420
Pattarawat Chormai - 0978675

October 1, 2015

IO.I-1

(i)

In this problem, we will show that when $m = M/B + 1$, if we have the optimal replacement policy, then it performs only $O(n/B + \sqrt(n))$ I/Os.

Indeed, when $M > B$ (which is obvious) and $m > M$, the array can be separated into blocks as being shown in figure 1.

Basically, we move in the column order, then each step in the first column requires an I/O. In each column, after filling the memory, there are several items left which total size is less than B. This is when we need the replacement policy. The optimal replacement method (sometimes called MIN) removes the block that has the longest time to be reused again in the future. In this particular setting, the OPT will remove the block which the last item has the smallest index. E.g. the 3^{rd} selected block in figure 1.

If B does not divides m, then each column in the array has $(m-1)/B$ blocks which the last item index is that column index, which are also the blocks that have the longest time to be reused in the future. Moreover, these blocks are indeed never used again in the same row. Therefore, the number of I/Os that MIN performs is equal to the number of I/Os that the row order performs, plus an extra number of I/Os for the remaining column of size $s < B$ (because B does not divides m).

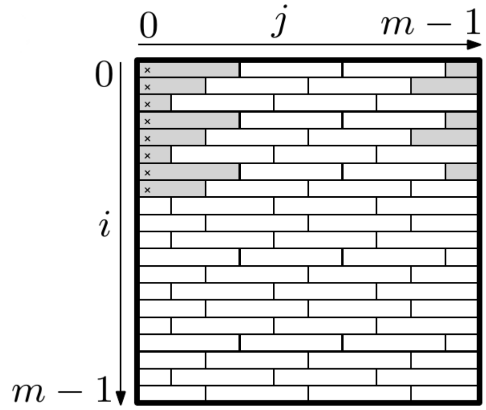

Figure 1: Blocks loaded into an 2-dimensional array

We have m rows, and $m/B+1$ I/Os for each row. Thus the number of I/Os is:

$$(\frac{m}{B}+1)m = O(\frac{n}{B}+\sqrt{n})$$

If B divides m, then we do not have the distribution as being shown in figure 1. Instead, all loaded blocks in a columns have the same ending index. Then the optimal replacement here is the Most Recently Used, that is, the policy evicts the latest used block. The array is now divided into m/B columns of size B. Each such column requires m I/Os for the first "real" column, and 1 I/O for each of the remaining columns. In general, the number of I/Os is:

$$\frac{m}{B}(m+B-1) = \frac{m^2}{B} + \frac{m(B-1)}{B} = O(\frac{n}{B}+\sqrt{n})$$

(ii)

In case $m > 2M/B$ and m is significantly larger than M, then each column of the array can store at least 2 memory-size regions, plus several individual items.

If we traverse the array in column-major order and the OS uses the optimal replacement strategy, then at the end of each column, all elements in the first $M - size$ part has been removed to get the space for the latest elements in the column. Therefore, whatever replacement method we use (even the optimal one), we cannot reuse any block when we move to the next column. Therefore, we need to load a new block everytime we want to fetch a new item. Thus, the I/O complexity is $\Omega(n)$.

If the OS uses the Most Recently Used replacement policy, then we can keep $m/2$ items for the first column, but we have to load-and-clear every item in the remaining $m/2$ items. Therefore, the I/O complexity is still a factor of m^2, which is $\Omega(n)$.

IO.I-2

(i)

Let $T(n)$ denote IO's of the problem and T_x, T_y and T_z denote IO's of matrix X, Y and T_z respectively. For each cell in Z, we need

$$\begin{aligned}
&T_x \leq \frac{\sqrt{n} + 2(B-1)}{B} \\
&T_y \leq \sqrt{n} \\
&T_z = 1
\end{aligned}$$

Then, the total IO's we need for computing a cell is :

$$\frac{\sqrt{n} + 2}{B} + \sqrt{n} + 1$$

Therefore, the total IO's that we need to compute the product $Z = XY$:

$$\begin{aligned}
T(n) &\leq (\frac{\sqrt{n} + 2(B-1)}{B} + \sqrt{n} + 1)n \\
&\leq \frac{n\sqrt{n} + 2n(B-1)}{B} + n\sqrt{n} + n \\
&= O(n\sqrt{n})
\end{aligned}$$

(ii)

If Y is stored in $column - major$ order. For each cell, we will use

$$T_y = \frac{\sqrt{n} + 2(B-1)}{B}$$

. Therefore, the total IO's is

$$
\begin{aligned}
T(n) &\leq (2\frac{\sqrt{n}+2(B-1)}{B}+1)n \\
&\leq \frac{2n\sqrt{n}+4n(B-1)}{B}+n \\
&= O(\frac{n\sqrt{n}}{B})
\end{aligned}
$$

(iii)

In order to compute sub-problem, all variables that we have in sub-problem should fit into main memory.

Let M_x, M_y and M_z denote memory space that is required by X, Y and Z when computing a sub-problem

For each sub problem, we need

$$
\begin{aligned}
M_x &= t(2t+2(B-1)) \\
M_y &= 2t(t+2(B-1)) \\
M_z &= t(t+2(B-1))
\end{aligned}
$$

Let M denote the total memory we have. Then, we find IO's recursive base case which happens when all variables in sub-problem can fit into main memory.

$$
\begin{aligned}
M_x + M_y + M_z &\leq M \\
5n^2 + 6(B-1)n &\leq M
\end{aligned}
$$

Thus, we can formulate recurrence IO's complexity function of this algorithm.

$$
T(t) = \begin{cases} \frac{5t^2+6(B-1)t}{B}, & \text{if } 5t^2+6(B-1)t \leq M \\ 4T(\frac{t}{2}), & \text{otherwise} \end{cases}
$$

Hence, we have a general form of the function where k is a depth of the recursive call.

$$
T(t) = 4^k T(\frac{t}{2^k})
$$

We know that if $\frac{n}{2^k} = 5n^2 + 6(B-1)n$ we will reach the base case. Then, we can find k.

$$
\begin{aligned}
\frac{t}{2^k} &= 5t^2 + 6(B-1)t \\
\frac{1}{2^k} &= 5t + 6(B-1) \\
2^k &= \frac{1}{5t + 6(B-1)} \\
k \log 2 &= \frac{1}{5t + 6(B-1)} \\
k &= \frac{1}{5t + 6(B-1)}
\end{aligned}
$$

Before we derive $T(t)$, we will denote A as

$$
\log_2 \frac{1}{5t + 6(B-1)}
$$

Next, we can simply derive the IO's complexity of the algorithm.

$$
\begin{aligned}
T(t) &\le 4^k T(\frac{t}{2^k}) \\
&\le 4^{log_2 A} T\Big(\frac{t}{2^{log_2 A}}\Big) \\
&\le A^2 T(\frac{t}{A}) \\
&\le A^2 \Big(\frac{5\frac{t}{A}^2}{B} + \frac{6(B-1)\frac{t}{A}}{B}\Big) \\
&\le \frac{5t^2}{B} + \frac{6(B-1)t}{AB} \\
&= O(\frac{t^2}{B})
\end{aligned}
$$

Therefore the IO's complexity of the algorithm is $O(t^2/B)$.

IO.1-3

(i)

The worst case of a binary search is when each step of comparison, we need to do an I/O to load the next center until the 2 consecutive centers are in the same block, which means the remaining array has to fit in the memory. Suppose that after k steps of binary searching, we reach that state, then:

$$2\frac{n}{2^k} \leq M \leftrightarrow \frac{n}{2^k} \leq \frac{M}{2}$$

Then,

$$k \geq log_2\frac{2n}{M}$$

This means we have to do $log_2\frac{2n}{M}$ I/Os before everything is inside the memory.

(ii)

To avoid the case that we perform an I/O everytime we make a comparison, we can organize the blocks differently. Now we group all of the sorted centers to blocks first, then the other elements. In this way, when we make a comparison, the other centers are loaded also in the same block, so it reduces the number of I/Os performed.

There are $2log(n)$ centers. So under this grouping policy, we need to perform $log(n)/B$ number of I/Os to find an element using binary search.

(iii)

Our proposed solution significantly increase the spatial locality. In the original approach, we load a block to read only 1 center. Meanwhile, in our solution, all items in a block are useful for the algorithm.

The temporal locality of the 2 approach is the same. It is because we never reuse the block again after reading it, in both methods.