Storing Matrices on Disk for Efficient Row and Column Retrieval

TR-99A

by

James R. Bitner*

Department of Computer Science
University of Texas
Austin, Texas 78712

May, 1979

Revised July, 1979

* - This work was supported in part by NSF grant MCS77-02705

Abstract: We study the problem of storing a matrix on disk assuming that the only accessing operation allowed is retrieving an entire row or column. The cost for retrieving a row or column is the number of different pages containing elements of that row or column. The cost of a matrix is the sum of the cost for retrieving each row and column. We give a lower bound on the cost of storing a matrix. We give one algorithm that asymptotically (for large matrices) achieves this bound if the page size is in a given set of integers, and another that achieves it if the page size is not in the set. We also analyze the asymptotic fraction of disk space wasted by these algorithms.

1. Introduction

We study the problem of storing a matrix on disk so it can be accessed efficiently, assuming that only two accessing operations are required: retrieving an entire row or retrieving an entire column. We measure the cost for retrieving a row or column by the number of different pages that must be read in from disk to obtain all elements of that row or column. The cost of a matrix is defined to be the cost of retrieving each row and column, summed over all rows and columns. This is motivated by the assumption that each row or column of the matrix must be retrieved equally often. We also use the phrase "cost of an algorithm" to mean the cost of a matrix stored according to a given algorithm.

Notation: m and n will always denote respectively the number of rows and columns in the matrix. s will denote the maximum number of matrix elements that can be stored on a page on disk.

Definition 1.1: Let COST(m,n,s) be the cost of a given algorithm and let $COST^*(m,n,s)$ be the optimal cost. Then the algorithm is asymptotically optimal for s iff

$$\lim_{m,n\to\infty} \frac{COST(m,n,s)}{COST^*(m,n,s)} = 1.$$

Note that we are interested in an algorithm which is asymptotically optimal for a fixed s; we are not interested in the limit as $s\to\infty$.

Definition 1.2: Let WASTE (m,n,s) be the number of unused locations in partially full pages for a given algorithm. Then

$\lim_{m,n\to\infty} \frac{WASTE(m,n,s)}{mn}$ is the asymptotic fraction of space wasted by the algorithm.

This paper has the following organization: Section 2 will give a lower bound (approximately $\frac{2mn}{\sqrt{s}}$) on the cost of storing a matrix. Section 3 will give an algorithm (Algorithm A) that asymptotically achieves the lower bound (and hence is asymptotically optimal) for all s in a specified set (see Theorem 2.2). Section 4 gives Algorithm B, which is asymptotically optimal for all s not in this set. The fact that two different strategies must be used to achieve the optimum is dictated by the form of the lower bound (see Theorem 2.1) which actually is the minimum of two quantities. One quantity is minimized by Algorithm A and the other by Algorithm B. Using the condition provided in Theorem 2.2 will allow us to easily tell which algorithm will give the best results for a given s. We also analyze the asymptotic fraction of space wasted by these algorithms. For Algorithm A, it is at most $\frac{1}{\lfloor\sqrt{s}\rfloor}$ (which is 4 or 5% for reasonable s), and for Algorithm B, it is zero. However, the storage scheme using Algorithm B is more complicated, requiring more calculation to determine which pages contain elements of a desired row or column. We also discuss this important aspect of the problem.

For purposes of comparison, we calculate the cost for storing the matrix in row major ordering, which is defined by visiting the elements of the matrix row-by-row (starting from the first row) and visiting the elements of each row from left to right. Elements $j(s-1)-1$ to js are stored in page j. The cost of this method is computed as follows (assume $s < n$): Each of the n columns will have cost m, and each of the m rows will have cost at least $\lceil\frac{n}{s}\rceil$. The total is then at least $nm+m\lceil\frac{n}{s}\rceil \geq nm(1+\frac{1}{s})$ compared to $\frac{2mn}{\sqrt{s}}$ for the optimal method.

2. A Lower Bound

In this section we derive a lower bound for cost of a matrix. Later, we will derive algorithms that asymptotically achieve this bound, proving it is tight. The first step is to assume the elements of the matrix are numbered 1,...,k (according to the page on which they are stored) and rewrite the cost which was defined to be

$$\text{COST} = \sum_{i=1}^{m} \text{number of different numbers in row } i + \sum_{i=1}^{n} \text{number of different numbers in column } i$$

as

$$\text{COST} = \sum_{i=1}^{k} (\text{the number of rows in which number } i \text{ occurs} + \text{the number of columns in which it occurs})$$

We refer to the $i^{\underline{th}}$ element of this sum as "the contribution of number i to the COST" (or just "the contribution of i"). We first derive a lower bound for this.

The following function is useful in defining the lower bound (see Figure 2.1).

Definition 2.1: Let $x = k^2 + j$ where $1 \leq j \leq 2k + 1$ (note $k+1 = \lceil \sqrt{x} \rceil$). Then $g(x)$ is defined by

$$g(x) = \begin{cases} \lceil \sqrt{x} \rceil + \lfloor \sqrt{x} \rfloor = 2k+1 & \text{if } j \leq k \\ 2\lceil \sqrt{x} \rceil = 2k+2 & \text{if } j > k \end{cases}$$

Values of x such that $g(x) < g(x+1)$ are especially important, motivating the following definition.

Definition 2.2: x is a square number iff it is of form k^2 for some $k \geq 1$. x is a rectangle number iff it is of form $k^2 + k$ for some $k \geq 1$. (This notation is somewhat unfortunate; the sets of square and rectangle numbers are disjoint.)

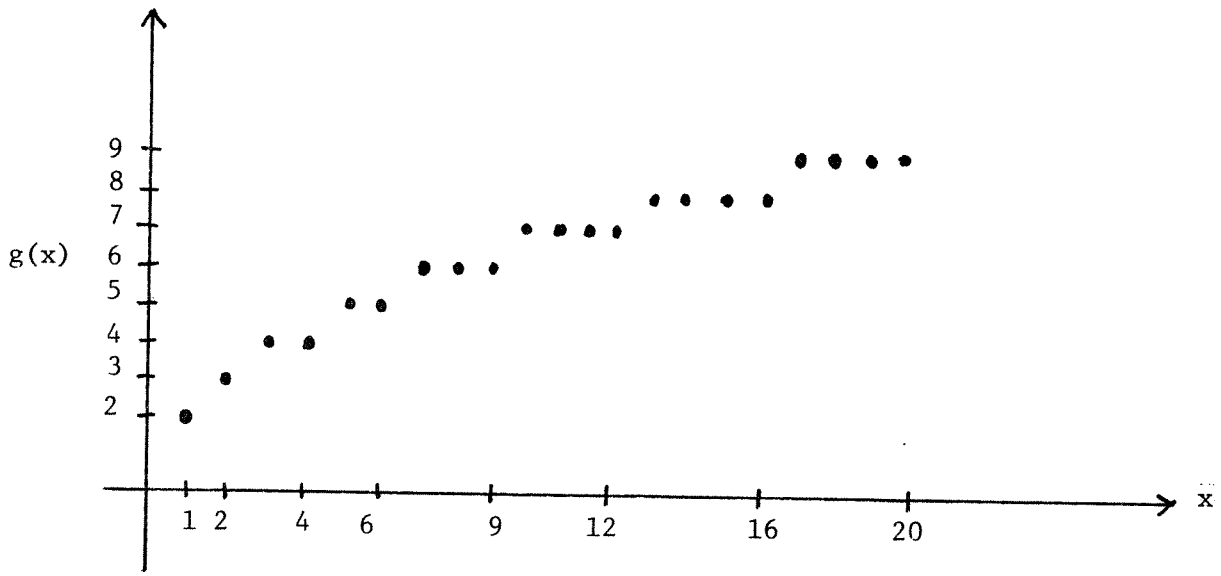

Figure 2.1
The function g(x)

Lemma 2.1: Let number i occur t times, then the contribution of i is at least g(t).

Proof: Suppose i occurs in a rows and b columns. The contribution of i is then a + b. Also, since each number i occurs at the intersection of one of these a rows and one of these b columns, there can be at most ab number i's and hence $t \leq ab$. We obtain a lower bound on the contribution of i by solving the following problem:

Minimize a + b

Subject to $ab \geq t$ and a,b integer

Obviously for a given a, choose $b = \lceil \frac{t}{a} \rceil$. The problem becomes:

Minimize $a + \lceil \frac{t}{a} \rceil$

s.t. $a \geq 1$, a integer

Define $f(x) = x + \lceil \frac{t}{x} \rceil$

$$\Delta(x) = f(x+1) - f(x) = 1 + \lceil \frac{t}{x+1} \rceil - \lceil \frac{t}{x} \rceil$$

Claim 1: If $x > y$ and $x-y \leq 1$

then $1 + \lceil y \rceil - \lceil x \rceil \geq 0$

Proof: Divide the real line into unit intervals $I_n = \{x \mid n-1 < x \leq n\}$ (Note, if $x \in I_n$, then $\lceil x \rceil = n$. We know $\lceil x \rceil \geq \lceil y \rceil$. Since $x - y \leq 1$, x and y are in the same or adjacent intervals. Hence $\lceil x \rceil - \lceil y \rceil \leq 1$

Claim 2: If $x > y$ and $x - y \geq 1$ then $1 + \lceil y \rceil - \lceil x \rceil \leq 0$

Proof: Here x and y cannot be in the same interval and $\lceil x \rceil - \lceil y \rceil \geq 1$

Claim 3 If $\frac{t}{x} - \frac{t}{x+1} \leq 1$ then $\Delta(x) \geq 0$

If $\frac{t}{x} - \frac{t}{x+1} \geq 1$ then $\Delta(x) \leq 0$

Proof: From claims 1 and 2

Claim 4: If for a given integer m,

$\Delta(x) \leq 0$ for all $x < m$

$\Delta(x) \geq 0$ for all $x \geq m$

then m is a minimum of f.

Proof: Note $f(x) = f(1) + \sum_{i=1}^{x-1} \Delta(i)$

If $x < m$ then $f(x) - f(m) = -\sum_{i=x}^{m-1} \Delta(i) \geq 0$

If $x > m$ then $f(x) - f(m) = \sum_{i=m}^{x-1} \Delta(i) \geq 0$

so $f(x) \geq f(m)$ for all x.

Claim 5: If for a given m

$\frac{t}{x} - \frac{t}{x+1} \geq 1$ if $x < m$ and

$\frac{t}{x} - \frac{t}{x+1} \leq 1$ if $x \geq m$

then x is a minimum of f

Proof: By Claim 3, such an m satisfies the conditions of Claim 4.

So we study the function $h(x) = \frac{t}{x} - \frac{t}{x+1} = \frac{t}{x(x+1)}$

Since we suspect that values of a around $\sqrt{t}$ will give us the minimum, we evaluate h at these points:

Notation: Let $t = k^2 + j$ where $1 \leq j \leq 2k + 1$ (then $\lceil \sqrt{t} \rceil = k + 1$)

$$h(k-1) = \frac{k^2+j}{(k-1)k} = \frac{k^2+j}{k^2-k}$$

$$h(k) = \frac{k^2+j}{k(k+1)} = \frac{k^2+j}{k^2+k}$$

$$h(k+1) = \frac{k^2+j}{(k+1)(k+2)} = \frac{k^2+j}{k^2+3k+2}$$

We observe that:

1. $h(k-1) \geq 1$ (because $j \geq 1 \geq -k$) and $h(x) \geq 1$ for $x \leq k-1$ because h is decreasing

2. $h(k+1) \leq 1$ (because $j \leq 2k+1 < 3k+2$) and $h(x) \leq 1$ for $x \geq k+1$

3. $h(k) \leq 1$ if $j \leq k$ and $h(k) \geq 1$ if $j \geq k$

Let $a = \begin{cases} k+1 & \text{if } j > k \\ k & \text{if } j \leq k \end{cases}$

then a is a minimum of f by Claim 5 and hence is a solution to our original problem. It is easy to see that $b = \left\lceil \frac{t}{a} \right\rceil = k+1$ (regardless of whether a is k or k + 1). The contribution of i is at least a + b which is equal to g(t). □

Theorem 2.1: The cost m x n matrix for page size s is at least $\min(\frac{g(p)}{p}, \frac{g(s)}{s}) \cdot mn$ where p is the largest square or rectangle number less than or equal to s.

Proof: Let y_j be the number of pages with j elements. Since each page of j elements has cost at least g(j), we can get a lower bound on the total cost by solving:

$$\text{minimize } \sum_{j=1}^{s} y_j g(j)$$

$$\text{s.t. } 0 \leq y_j$$

$$\text{and } \sum_{j=1}^{s} j y_j = mn$$

We now lift the restriction that the y_j be integers. This cannot increase the minimum value of the objective function, and hence, solving this modified problem will still give a lower bound on the total cost of the matrix. If we view each y_j as a number of objects (where we can take a fraction of an object) with size j and weight g(j) that are to be put into a knapsack of size mn, the objective is to fill the knapsack while minimizing the weight. This can clearly be done by using only the type of object with lowest weight to size ratio. This is proven in the following claim:

Claim 1: Let j_0 be the integer that minimizes

$\frac{g(j)}{j}$. Then a solution to the above problem is

$\{y_{j_0} = \frac{mn}{j_0}; \; y_j = 0 \text{ for } j \neq j_0\}$

Proof: Define $z_j = jy_j$ and $h(j) = \frac{g(h)}{j}$.

(Note that j_0 minimizes $h(j)$)

The problem becomes:

$$\text{mimimize} \sum_{j=1}^{s} z_j h(j)$$

$$\text{s.t.} \quad \sum_{j=1}^{s} z_j = mn \; ; \; z_j \geq 0$$

Because

$$\sum_{j=1}^{s} z_j h(j) \geq \sum_{j=1}^{s} z_j h(j_0) = h(j_0) \cdot mn,$$

$h(j_0) \cdot mn$ is a lower bound on the cost of any solution. Since the solution stated in the claim achieves this cost, it is an optimal solution, proving this claim.

The problem now is to determine j_0 for a given s. This is done in the following claim:

Claim 2: If $1 \leq j \leq s$, $\frac{g(j)}{j}$ is minimized by either s or the largest square or rectangle number less than or equal to s.

Proof: If j is neither a square number nor a rectangle number nor s, then $g(j+1) = g(j)$ and $\frac{g(j+1)}{j+1} < \frac{g(j)}{j}$. Hence j does not minimize $\frac{g(j)}{j}$. Suppose j is a square or rectangle number, but not the largest one less than or equal to s. Let r be the next largest square or rectangle number. We consider two cases:

Case 1 (j is a square number, say k^2, and r is k^2+k.): Then

$$\frac{g(j)}{j} - \frac{g(r)}{r} = \frac{2k}{k^2} - \frac{2k+1}{k^2+k} = \frac{1}{(k^2+k)} > 0$$

Case 2 (j is a rectangle number, say $k^2 + k$, and r is $(k + 1)^2$): Then

$$\frac{g(j)}{j} - \frac{g(r)}{r} = \frac{2k+1}{k^2+k} - \frac{2k+2}{(k+1)^2} = \frac{1}{k^2+k} > 0$$

In any case, $\frac{g(j)}{j} > \frac{g(r)}{r}$

Therefore, the only possibilities for the minimum are s and p, where p denotes the largest square or rectangle number less than or equal to s. Therefore $\frac{g(j_0)}{j_0} = \min(\frac{g(s)}{s}, \frac{g(p)}{p})$. Substituting this and Claim 1 in $\sum_{j=1}^{s} y_j \; g(j)$ gives $\min(\frac{g(s)}{s}, \frac{g(p)}{p}) \cdot mn$ □

The following theorem gives a simple condition for whether $\frac{g(s)}{s} < \frac{g(p)}{p}$.

Theorem 2.2: Let $s = k^2 + j$; $1 \leq j \leq 2k + 1$, let p be the largest square or rectangle number less than or equal to s and suppose $s \neq p$. Then

a) If p is a square number then $\frac{g(p)}{p} \leq \frac{g(s)}{s}$ iff $j \leq \frac{k}{2}$ (with equality occurring if k is even and $j = \frac{k}{2}$)

b) If p is a rectangle number then $\frac{g(p)}{p} < \frac{g(s)}{s}$ iff $j < \frac{k(3k+2)}{(2k+1)}$.

Proof:

Case 1 (p is a square, say k^2, and $s=k^2+j$; $1 \leq j \leq k - 1$) Then

$$\frac{g(p)}{p} - \frac{g(s)}{s} = \frac{2k}{k^2} - \frac{2k+1}{k^2+j} = \frac{2j-k}{k(k^2+j)}$$

Case 2: (p is a rectangle number, say k^2+k, and $s = k^2 + j$; $k+1 \leq j \leq 2k$) : Then

$$\frac{g(p)}{p} - \frac{g(s)}{s} = \frac{2k+1}{k^2+k} - \frac{2k+2}{k^2+j} = \frac{(2k+1)j-k(3k+2)}{(k^2+k)(k^2+j)}$$

In either case, the theorem is obviously true. □

Note that $\frac{g(p)}{p} \leq \frac{g(s)}{s}$ for about half the values of s. The following two sections will give algorithms that asymptotically achieve the lower bound in Theorem 2.1. Algorithm A is asymptotically optimal if $\frac{g(p)}{p} \leq \frac{g(s)}{s}$. Otherwise Algorithm B is optimal.

3. An Asymptotically Optimal Algorithm

In this section, we give an algorithm that is asymptotically optimal if $\frac{g(p)}{p} \leq \frac{g(s)}{s}$, where p is the largest square or rectangle number less than or equal to s. This algorithm also causes a fraction (at most $\frac{1}{\lfloor\sqrt{s}\rfloor}$) of the disk space to be wasted.

The strategy we must employ is clear from the proof of Theorem 2.1; the elements with a given number should be grouped into rectangles of dimension a x b such that $a = \lfloor\sqrt{s}\rfloor$ and $b = \lfloor\sqrt{s}\rfloor$ or $\lceil\sqrt{s}\rceil$, whichever will give ab=p. The algorithm follows this heuristic as much as possible.

Notation: Let p be the largest square or rectangle number less than or equal s. Define $a = \lfloor\sqrt{s}\rfloor$ and $b = \lceil\sqrt{s}\rceil$

and let y = m mod a and z = n mod b.

Algorithm A:

1. Partition the submatrix consisting of the first $\lfloor\frac{m}{a}\rfloor \cdot a$ rows and $\lfloor\frac{n}{b}\rfloor \cdot b$ columns into blocks of dimension axb. (See Figure 3.1). (We say these blocks are "type A".) Put each block on a different page on disk.

2. If $y \neq 0$, partition the elements in the last y rows into blocks (type B) of dimension $y \times \lfloor\frac{s}{y}\rfloor$. Put all the left-over elements (if any) in a separate block (type C). (If y = 0, step 1 will have left no rows to be partitioned and step 2 does nothing.)

3. Partition the elements in the last z columns (except the last y rows) into blocks (type D) of dimension $\lfloor\frac{s}{z}\rfloor \times z$. (If z = 0, step 3 does nothing). Again, put all the left over elements into a separate block (type E).

Theorem 3.1: Let p be the largest square or rectangle number less than or equal s. Then Algorithm A has cost at most

$$\frac{g(p)}{p} \cdot mn + O(m + n + \sqrt{s})$$

which is asymptotically optimal if $\frac{g(p)}{p} \leq \frac{g(s)}{s}$.

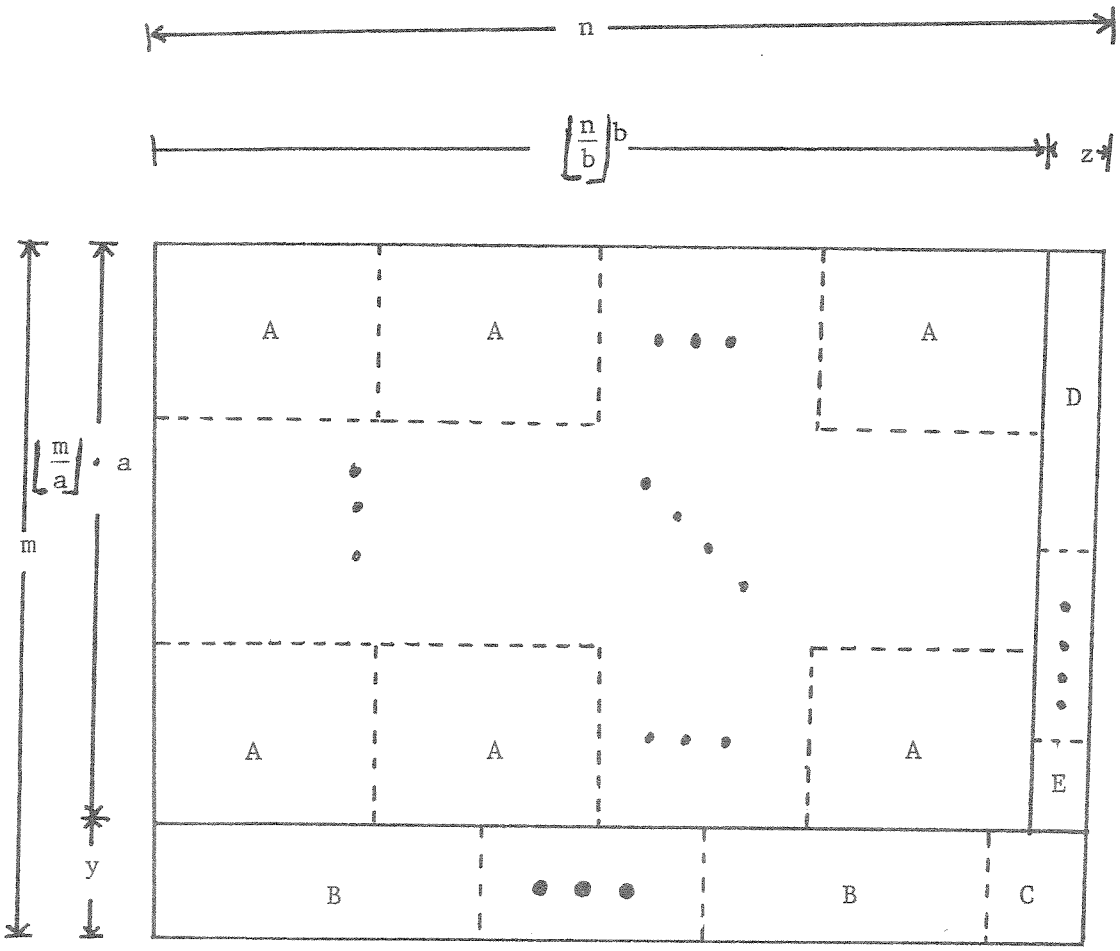

Figure 3.1

How Algorithm A partitions a matrix

Block type	Number	Dimensions	Zero If
A	$\lfloor \frac{m}{a} \rfloor \lfloor \frac{n}{b} \rfloor$	axb	never
B	$\left\lfloor \frac{n}{\lfloor \frac{s}{y} \rfloor} \right\rfloor$	$yx \lfloor \frac{s}{y} \rfloor$	y=0
C	1	$yx(n \bmod \lfloor \frac{s}{y} \rfloor)$	y=0 or $n \bmod \lfloor \frac{s}{y} \rfloor = 0$
D	$\left\lfloor \frac{m-y}{\lfloor \frac{s}{z} \rfloor} \right\rfloor$	$\lfloor \frac{s}{z} \rfloor xz$	z=0
E	1	$((m-y) \bmod \lfloor \frac{s}{z} \rfloor)\ xz$	z=0 or $(m-y) \bmod \lfloor \frac{s}{z} \rfloor = 0$

Table 3.1

The number and dimensions of blocks of each type. If the condition in the "Zero If" column is true then the "Number" and "Dimension" columns should be ignored and there are zero blocks of that type.

Proof: Table 3.1 is easily seen from Figure 3.1. Let $COST_A, COST_B, \ldots$ be the total cost for blocks of that type. (To get these costs, we take the cost for the given block type (the sum of the dimensions for a block of that type) and multiply by the number of blocks of that type.) the total cost is then given by

$$COST = COST_A + COST_B + COST_C + COST_D + COST_E \tag{1}$$

We first observe that

$$COST_A = (a+b)\left\lfloor \frac{m}{a} \right\rfloor \left\lfloor \frac{n}{b} \right\rfloor \leq (a+b) \cdot \frac{mn}{ab} \tag{2}$$

We next examine $COST_B + COST_C$. Substituting the values given by Table 3.1 gives:

$$COST_B + COST_C = (y + \left\lfloor \frac{s}{y} \right\rfloor) \left\lfloor \frac{n}{\left\lfloor \frac{s}{y} \right\rfloor} \right\rfloor + \left[y + (n \bmod \left\lfloor \frac{s}{y} \right\rfloor) \right] \tag{3}$$

if $y \neq 0$, and zero if $y = 0$. We only consider the case where $y = 0$ since the positive upper bound we obtain will also be an upper bound when $y = 0$.

(3) equals.

$$\left(\left\lfloor \frac{n}{\left\lfloor \frac{s}{y} \right\rfloor} \right\rfloor \cdot \left\lfloor \frac{s}{y} \right\rfloor + n \bmod \left\lfloor \frac{s}{y} \right\rfloor \right) + y \left(\left\lfloor \frac{n}{\left\lfloor \frac{s}{y} \right\rfloor} \right\rfloor + 1 \right) \tag{4}$$

The first parenthesized expression is simply n. The rest is increasing with respect to y. Since $y = m \bmod a$, $y \leq a-1$. Therefore (4) is less than or equal to

$$n + (a-1) \cdot \left(\left\lfloor \frac{n}{\left\lfloor \frac{s}{a-1} \right\rfloor} \right\rfloor + 1 \right) \tag{5}$$

since $a = \lfloor \sqrt{s} \rfloor$, we have $a-1 \leq \sqrt{s}$ and $\left\lfloor \frac{s}{a-1} \right\rfloor \geq a$

Substituting this last inequality, removing the last floor sign and simplifying gives (5) is less than or equal

$$\left(\frac{2a-1}{a}\right) n + (a-1) \leq 2n + (a-1)$$

Hence $COST_B + COST_C \leq 2n + (a-1)$ (6)

This analysis also holds for $COST_D+COST_E$ (a and b are interchanged, and m and z take the place of respectively, n and y). The expression equivalent to (5) is

$$m + (b-1)\left(\left\lfloor \frac{m}{\left\lfloor \frac{s}{b-1} \right\rfloor} \right\rfloor + 1\right) \tag{7}$$

Since $b \leq \lceil \sqrt{s} \rceil$, we have $b-1 \leq \sqrt{s}$ and $\left\lfloor \frac{s}{b-1} \right\rfloor \geq \lfloor \sqrt{s} \rfloor \geq b-1$.

Putting this into (7) gives:

$$COST_D+COST_E \leq 2m + (b-1) \tag{8}$$

Substituting (2), (6), and (8) into (1) gives

$$COST \leq (a+b)\frac{mn}{ab} + 2n + (a-1) + 2m + (b-1)$$

$$= \frac{g(p)}{p} \cdot mn + O(m + n + \sqrt{s})$$

□

Theorem 3.2 The wasted space for Algorithm A is at most $(s-ab)\frac{mn}{ab} + n+m+2s$. Asymptotically, the fraction of wasted space $\frac{s-ab}{ab}$ which is at most $\frac{1}{\lfloor \sqrt{s} \rfloor}$.

Proof: Let w(m,n) be the number of locations wasted for storing an mxn matrix. Table 3.1 is used to calculate the waste due to each type of block. (The waste is s minus the area of the block.) This gives

$$w(m,n) \leq (s-ab)\left\lfloor \frac{m}{a} \right\rfloor \cdot \left\lfloor \frac{n}{b} \right\rfloor + (s \bmod y)\cdot \left\lfloor \frac{n}{\left\lfloor \frac{s}{y} \right\rfloor} \right\rfloor + s + (s \bmod z)\cdot \left\lfloor \frac{m+y}{\left\lfloor \frac{s}{z} \right\rfloor} \right\rfloor + s$$

$$\leq (s-ab)\frac{mn}{ab} + y\cdot \left\lfloor \frac{n}{\left\lfloor \frac{s}{y} \right\rfloor} \right\rfloor + s + z\left\lfloor \frac{m}{\left\lfloor \frac{s}{z} \right\rfloor} \right\rfloor + s$$

This is maximized when y = a-1 and z = b-1. Using the inequalities $\left\lfloor \frac{s}{a-1} \right\rfloor \geq a$ and $\left\lfloor \frac{s}{b-1} \right\rfloor \geq b-1$ gives

$$w(m,n) \leq (s-ab)\frac{mn}{ab} + n + m + 2s$$

Finally, since $s \leq (a+1)b$, $\frac{s-ab}{ab} \leq \frac{1}{a} = \frac{1}{\lfloor \sqrt{s} \rfloor}$

□

Note that the size of $\frac{s-ab}{ab}$ depends on how close s is to the next larger square or rectangle number. In fact, if s is a square or rectangle number, the waste is 0. In any case, the waste is no more than $\frac{1}{\lfloor\sqrt{s}\rfloor}$ which is 4 or 5% for reasonable s.

Finally, a desirable property of Algorithm A is that it is very easy to compute which blocks need to be retrieved to obtain a given row and then assign the proper elements from each block to the appropriate positions in an array. Figure 3.2 gives a PASCAL procedure for retrieving a row; retrieving a column is similar. This procedure assumes that the blocks labeled "A" are numbered consecutively (row-by-row from left to right) then those labeled "B" are numbered, then "C", "D" and "E" and that the elements in each block are stored in row major order.

```
(* The value of global variable dstart is the number of the first block
    labeled "D".  It is defined as follows.                          *)

dstart := ⌊m/a⌋*⌊n/b⌋;
if y <> 0 then begin
        dstart := y*⌊s/y⌋;
        if n mod ⌊s/y⌋ <> 0 then dstart := dstart+1;
end;


(* Procedure rowget retrieves row r from disk and assigns it to
    array A[0..n-1]                                                  *)

procedure rowget(r : integer);
var i,j,blocklen : integer;
begin
j := 0;
if r <=⌊m/a⌋*a-1 then begin
        for i := ⌊r/a⌋*⌊n/b⌋ to (⌊r/a⌋+1)*⌊n/b⌋-1 do begin
                RETRIEVE block i;
                ASSIGN elements (r mod a)*b to (r mod a + 1)*b-1
                   of this block to positions j to j+b-1 of A;
                j := j + b;
        end;
        if z <> 0 then begin
                RETRIEVE block dstart +⌊r/⌊s/z⌋⌋;
                ASSIGN elements (r mod ⌊s/z⌋)*z to (r mod ⌊s/z⌋+1)*z-1
                   of this block to positions j to n-1 of A;
        end;
end
else begin
        for i := ⌊m/a⌋*⌊n/b⌋ to dstart-2 do begin
                RETRIEVE block i;
                ASSIGN elements (r-⌊m/a⌋*a)*⌊s/y⌋to
                   (r-⌊m/a⌋*a+1)*⌊s/y⌋-1
                   to positions j to j+⌊s/y⌋-1 of A;
                j := j+⌊s/y⌋;
        end;
        (* retrieve the final block, which might be shorter
           than the others                                           *)
        if n mod ⌊s/y⌋ = 0 then blocklen := ⌊s/y⌋
                           else blocklen := n mod ⌊s/y⌋;
        RETRIEVE block dstart-1;
        ASSIGN elements (r-⌊m/a⌋*a)*blocklen to
           (r-⌊m/a⌋*a+1)*blocklen-1
           to positions j to n-1 of A;
end;
end;
```

Figure 3.2
Retrieving a row from disk if the matrix is stored using Algorithm A. m,n,a,b,y,z are global constants whose values are defined at the beginning of this section.

4. Another Asymptotically Optimal Algorithm

We now discuss an algorithm that is asymptotically optimal if $\frac{g(s)}{s} \leq \frac{g(p)}{p}$ where p is the largest square or rectangle number less than or equal to s. The strategy of the previous section will not give an asymptotically optimal result here. Because $\frac{g(s)}{s} \leq \frac{g(p)}{p}$, the elements must be grouped as much as possible into pages of size s with each page having cost g(s). This can be done by partitioning the matrix into rectangles of size axb where $b = \lceil \sqrt{s} \rceil$ and $a = \lfloor \sqrt{s} \rfloor$ if ($\lfloor \sqrt{s} \rfloor \lceil \sqrt{s} \rceil \geq s$) or $\lceil \sqrt{s} \rceil$ (otherwise). Clearly, if s is not a square or rectangle number, there will be elements inside of this rectangle that cannot be put in this page. In order for the algorithm to be asymptotically optimal, these left-over elements must also be assigned to pages in an efficient manner. The algorithm does this by grouping all the left-over elements together and recursively applying itself to the submatrix formed of these elements.

Notation: Assume $s = k^2+j$ where $1 \leq j \leq 2k+1$.

Let $b = \lceil \sqrt{s} \rceil$ and

$$a = \begin{cases} \lfloor \sqrt{s} \rfloor & \text{if } j \leq k \\ \lceil \sqrt{s} \rceil & \text{if } j > k \end{cases}$$

(Note: a and b are defined differently in Section 3.) Further let y = m mod a and z = n mod b

Algorithm B: (see Figure 4.3 for a sample partitioning.)

If m < a or n < b then execute case 1

else execute case 2

Case 1: (Without loss of generality assume $m \leq n$.) Partition the matrix into blocks marked "A" in Figure 4.1 of dimension $m \times \lceil \frac{s}{m} \rceil$ with any remaining columns partitioned into a separate block(marked "C"). If (-s) mod m ≠ 0,

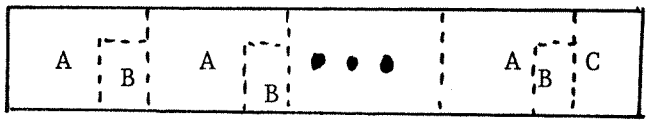

Figure 4.1 (a)

How Algorithm B partitions a matrix if m < a.

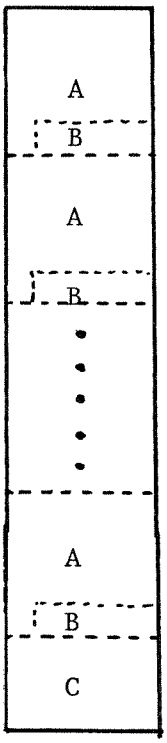

Figure 4.1 (b)

How Algorithm B partitions a matrix if n < b.

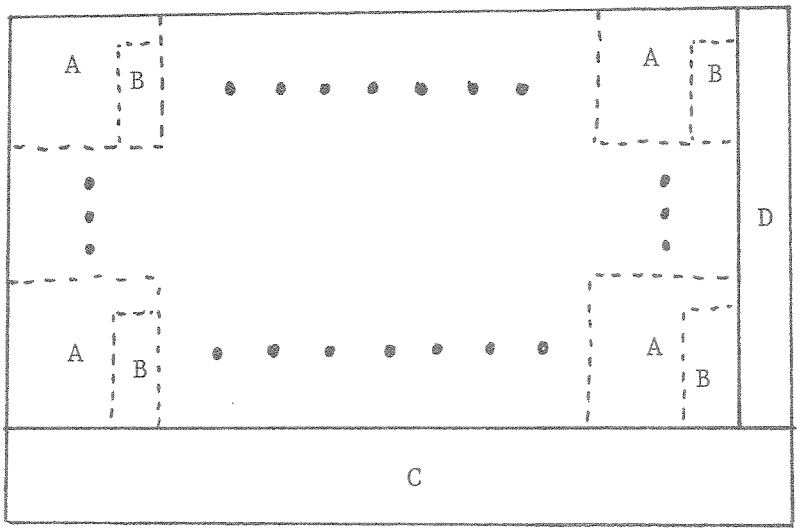

Figure 4.2

How Algorithm B partitions a large matrix ($m \geq a$ and $n \geq b$).

there will be left-over elements. Further partition each block so that the last (-s) mod m elements in the last row are in a separate block (marked"B"). These blocks are grouped together to form a submatrix, and Algorithm B is recursively called to partition it. (The algorithm does not require that the submatrix it is partitioning be contiguous, only that the coordinates of the elements of submatrix can be described as a Cartesian product of indices. This is because permuting the rows and columns of a matrix will not change its cost, and hence, any such submatrix can be permuted into a contiguous submatrix.)

Case 2: Partition the submatrix consisting of the first $\lfloor \frac{m}{a} \rfloor a$ rows and $\lfloor \frac{n}{b} \rfloor b$ columns (again, a and b are different from those in Section 3.) into blocks (marked "A" in Figure 4.2) of dimension a x b. If $ab \neq s$ there will be left-over elements. Further partition each block so that the last s-ab elements in the last row are in a separate block (marked "B"). Group these blocks together (as in (1)) and recursively call Algorithm B to partition the resulting submatrix. If $y \neq 0$ call Algorithm B to partition the last y rows (marked "C", and if $z \neq 0$ call Algorithm B to partition the last z columns except the last y rows (marked "D").

Notation: Let COST (m,n) be the cost of Algorithm B.

Clearly, we have the following recurrence relations for COST.

1) If $m \geq a$ and $n \geq b$

$$\text{COST}(m,n) = (a+b)\left\lfloor \frac{m}{a} \right\rfloor \left\lfloor \frac{n}{b} \right\rfloor + \text{COST}\left((ab-s)\left\lfloor \frac{m}{a} \right\rfloor, \left\lfloor \frac{n}{b} \right\rfloor\right) + \text{COST}(y,n) + \text{COST}(m-y,z)$$

2) If $0 < m < a$

$$\text{COST}(m,n) = \left(m+\left\lceil \frac{s}{m} \right\rceil\right)\left\lfloor \frac{n}{\left\lceil \frac{s}{m} \right\rceil} \right\rfloor + \left(m + n \bmod \left\lceil \frac{s}{m} \right\rceil\right) + \text{COST}\left((-s) \bmod m, \left\lfloor \frac{n}{\left\lceil \frac{s}{m} \right\rceil} \right\rfloor\right)$$

where the second parenthesized expression should be omitted if $n \bmod \lceil \frac{s}{m} \rceil = 0$.

3) If $0 < n < b$ (similar to 2)

4) COST $(0,n) = 0$ and COST $(m,0) = 0$ for all m and n.

We now derive a bound on COST . The difference from the lower bound will be $O(m\sqrt{s} + n)$ instead of $O(m + n + \sqrt{s})$ for Algorithm A because now the "edge effects" occur not only near the edges, but also during recursive calls of Algorithm B.

Lemma 4.1: If $m < a$ and $n \geq m$, COST$(m,n) \leq 6n$.

Proof: We first dispose of some trivial cases:

Case 1: $(a = 1)$ $m=0$ and COST$(m,n) = 0 \leq 6n$.

Case 2: $(a = 2)$: Either $m=0$ (and COST$(m,n) = 0$ again), or $m=1$ and

$$\text{COST}(m,n) \leq (1+s)\left\lfloor \frac{n}{s} \right\rfloor + (1+s \bmod n)$$
$$= n + \left\lfloor \frac{n}{s} \right\rfloor + 1 \leq 3n$$

Case 3 $(a \geq 3)$: Here we proceed by induction on m. If $m=0$, the lemma is trivial. For $m > 0$, we have

$$\text{COST}(m\ n) \leq \left(m + \left\lceil \frac{s}{m} \right\rceil \right)\left\lfloor \frac{n}{\left\lceil \frac{s}{m} \right\rceil} \right\rfloor + \left(m + n \bmod \left\lceil \frac{s}{m} \right\rceil \right)$$
$$+ \ \text{COST}\left((-s) \bmod m, \left\lfloor \frac{n}{\left\lceil \frac{s}{m} \right\rceil} \right\rfloor \right)$$

where we have equality unless $n \bmod \lceil \frac{s}{m} \rceil = 0$ in which case the second term is an upper bound on zero. (1) equals

$$n + m\left(1 + \left\lfloor \frac{n}{\left\lceil \frac{s}{m} \right\rceil} \right\rfloor \right) + \text{COST}\left((-s) \bmod m, \left\lfloor \frac{n}{\left\lceil \frac{s}{m} \right\rceil} \right\rfloor \right)$$

$$\leq n + m + m\left\lfloor \frac{n}{\left\lceil \frac{s}{m} \right\rceil} \right\rfloor + 6\left\lfloor \frac{n}{\left\lceil \frac{s}{m} \right\rceil} \right\rfloor$$

by induction. The third and fourth terms are maximized when $m = a-1$, and $\lceil \frac{s}{a-1} \rceil \geq a-1$. Hence

$$\text{COST}(m,n) \leq n+m+(a-1) \cdot \frac{n}{a-1} + 6 \cdot \frac{n}{a-1}$$

$$= 3n + 6 \cdot \frac{n}{a-1} \leq 6n$$

because $a \geq 3$. □

Lemma 4.2: If $n < b$ and $m \geq n$, $\text{COST}(m,n) \leq 6m$.

Proof: Same as Lemma 4.1. □

Theorem 4.1: $\text{COST}(m,n) \leq \frac{g(s)}{s} \cdot mn + 0\ (m\sqrt{s} + n)$ which is asymptotically optimal if $\frac{g(s)}{s} \leq \frac{g(p)}{p}$.

Proof: We show $\text{COST}(m,n) \leq \frac{g(s)}{s} \cdot mn + 6am + 12n$.

We first dispose of the case where b=1. If b=1, then a=1, s=1, and COST(m,n) is obviously 2mn, and the theorem is true. We consider any $b \geq 2$. The proof is by induction of n. At each step, we show the theorem is true for a given n and all m. The basis (for $n < b$) is given by Lemmas 4.1 and 4.2. Both bounds are less than the bound given in the theorem. We now assume $n \geq b$. For any m,

$$\text{COST}(m,n) = (a+b)\left\lfloor \frac{m}{a} \right\rfloor \left\lfloor \frac{n}{b} \right\rfloor + \text{COST}\left((ab-s)\left\lfloor \frac{m}{a} \right\rfloor, \left\lfloor \frac{n}{b} \right\rfloor\right)$$

$$+ \text{COST}(y,n) + \text{COST}(m-y,z)$$

By Lemmas 4.1 and 4.2, the last two terms are bounded by 6n and 6m, respectively. Then by induction,

$$\text{COST}(m,n) \leq (a+b)\frac{mn}{ab} + \frac{g(s)}{s} \cdot (ab-s)\frac{mn}{ab}$$

$$+ 6a(ab-s) \cdot \frac{m}{a} + 12\frac{n}{b} + 6m + 6n$$

$$\leq \frac{g(s)}{s} \cdot mn + \left[6(a-1)+6\right] \cdot m + \left(\frac{12}{b} + 6\right)n$$

because $ab-s \leq a-1$. Hence

$$\text{COST}(m,n) \leq \frac{g(s)}{s} \cdot mn + 6am + 12n$$

because $b \geq 2$. □

We now show that this algorithm uses disk space very efficiently.

Theorem 4.2 If $n \geq b > 1$ the wasted space for Algorithm B is at most $2s(a+b)\log_b n$. (If b=1, no space is wasted.) Asymptotically, the fraction of wasted space is zero.

Proof: Assume b > 1, otherwise the theorem is trivial. Let w(m,n) be the number of wasted locations for storing an mxn matrix.

Claim 1: If $m < a$, $w(m,n) \leq sm$.

Proof: (by induction on m): If m=0, w(0,n) = 0 and the claim is true. Otherwise we have

$$w(m,n) \leq (s-1) + w((-s) \bmod m, \left\lfloor \frac{n}{\lceil \frac{s}{m} \rceil} \right\rfloor)$$

because no waste results from the blocks marked "A" in Figure 4.1, at most s-1 locations are wasted in the block marked "C" and the final term gives the waste in the partition constructed by the recursive call. Let $m' = (-s) \bmod m$. Since $m' \leq m-1$, we have by induction

$w(m,n) \leq s + sm' \leq sm$

proving the claim.

Claim 2: If $n < b$, $w(m,n) \leq sn$.

Proof: Same as Claim 1.

We now consider general case. Because the blocks marked "A" in Figure 4.2 waste no disk space, we have

$$w(m,n) = w((ab-s)\left\lfloor \frac{m}{a} \right\rfloor, \left\lfloor \frac{n}{b} \right\rfloor) + w(m-y,z) + w(y,n)$$

We prove the theorem by induction on n

Basis: ($b \leq n < b^2$) Then $\left\lfloor \frac{n}{b} \right\rfloor < b$. Using Claims 1 and 2,

$$w(m,n) \leq s\left\lfloor \frac{n}{b} \right\rfloor + sy + sz \leq 2sb + sa$$
$$\leq 2s(a+b)\log_b n$$

Induction: ($n \geq b^2$) Using induction on the first term and Claims 1 and 2 on the second and third give

$$w(m,n) \leq 2s(a+b)\log_b \left\lfloor \frac{n}{b} \right\rfloor + sy + sz$$
$$\leq 2s(a+b)\log_b n - 2s(a+b) + sa+sb$$
$$\leq 2s(a+b)\log_b n$$

proving the theorem. Dividing by mn and letting m and $n \to \infty$ proves the asymptotic waste fraction is zero. □

We now describe a procedure for retrieving a row or column of a matrix according to Algorithm B. In this discussion it is more convenient to work with the following equivalent form of Algorithm B:

A modified (but equivalent) version of Algorithm B:

1. Divide the matrix into regions A,B,C and D as shown in Figure 4.2.
2. Divide regions C (if non-empty) and D (if non-empty) as shown in Figure 4.1 producing regions C_A, C_B, C_C and D_A, D_B, D_C.
3. Call the algorithm recursively for each of the regions B, C_B and D_B which are non-empty.

This reformulation allows the following useful definition:

Definition: The level of a block of a matrix stored using Algorithm B is defined to be the depth of the recursion (i.e. the number of calls) in the modified version of Algorithm B when this block is created. A group is defined to be the set of all blocks at the same level created by the same recursive call of the modified version of Algorithm B. (see Figure 4.3).

A small data structure, a ternary tree (see Figure 4.4), is required to store information about each group. The root of the tree contains information about the group at level 0, and the left, middle and right sons of a node for a given group contain, respectively, the information about the groups resulting from the partitioning region B, C_B and D_B in the modified version of Algorithm B. This data structure is initially constructed for the given m, n and s and is then used to direct the algorithm for accessing the rows and columns.

0	0	0	1	1	1	2	2	2	15	15
0	0	18	1	1	18	2	2	18	15	15
3	3	3	4	4	4	5	5	5	15	21
3	3	18	4	4	18	5	5	20	16	16
6	6	6	7	7	7	8	8	8	16	16
6	6	19	7	7	19	8	8	19	16	21
9	9	9	10	10	10	11	11	11	17	17
9	9	19	10	10	19	11	11	20	17	17
12	12	12	12	12	13	13	13	13	13	14

Figure 4.3

A 9x11 matrix stored using Algorithm B.

We assume a page size of 5 elements. The number in a given square is the block to which that element is assigned. There are four groups:

blocks 0-17	(level 0)
blocks 18 and 19	(level 1)
block 20	(level 2)
block 21	(level 1)

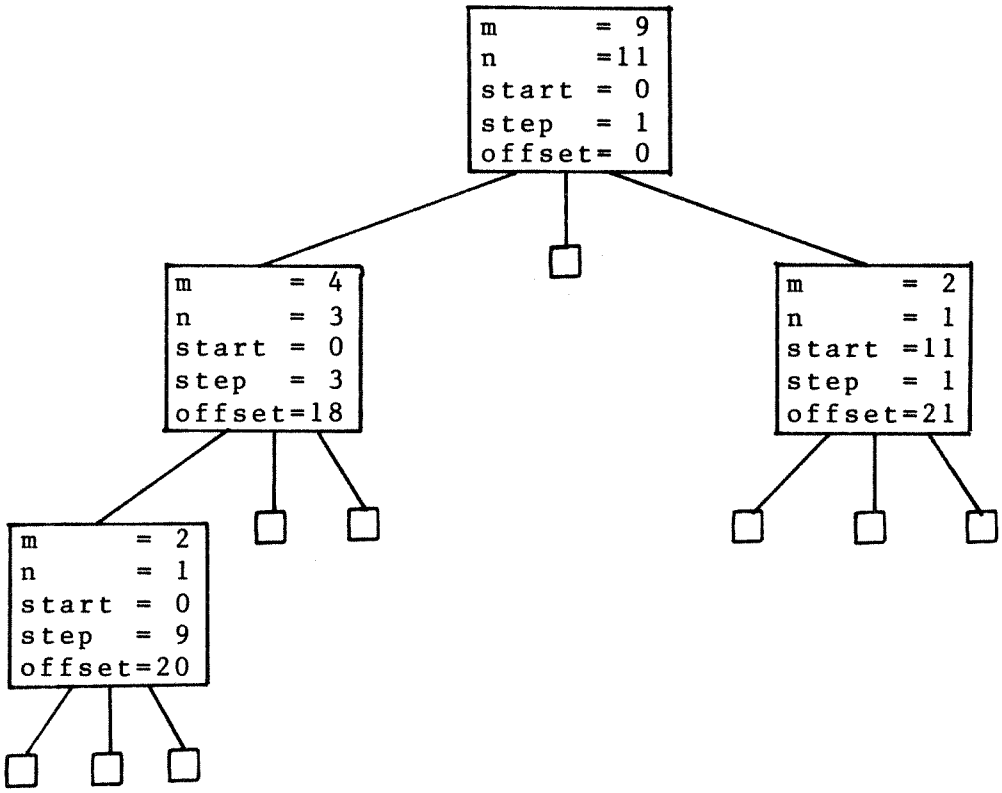

Figure 4.4

The ternary tree used by Algorithm B for storing information about the matrix in Figure 4.3.

The group of blocks 18 and 19 corresponds to the leftmost node at level 1, and the group consisting of block 21 corresponds to the rightmost.

We make the assumption that all the blocks in a group are numbered as in the accessing scheme for Algorithm A (see Figure 4.3). Outside of this restriction, the numbering may be arbitrary. We also assume that the elements of a block labeled A, C_A or C_C are stored in column major order (due to the fact that elements in the last column of such blocks have been removed.) Elements of a block labeled D_A or D_C are stored in row major order.

We now describe the method for retrieving a row (shown in the Appendix). Accessing a column is similar, but slightly more complicated. Most of the work is done in procedure getgroup (Figure A.2) which retrieves all the blocks in a given group. Due to our numbering assumptions, the accessing scheme for Algorithm A (with the following modifications) can be used to determine which blocks from a given group must be retrieved. The values used for a and b must be those defined in Section 4, and $\lfloor \frac{s}{y} \rfloor$ and $\lfloor \frac{s}{z} \rfloor$ must be replaced by $\lceil \frac{s}{y} \rceil$ and $\lceil \frac{s}{z} \rceil$ respectively. Further, the number of the first block in the group (called the group's offset) must be added into every block number. This is one of the pieces of information about the group stored in the tree. The algorithm is further complicated by the fact that the elements of some blocks are stored in row major order while others are stored in column major order; requiring two procedures, getblockrm and getblockcm for accessing elements within a block. The modified algorithm, getgroup, is shown in Figure A.2.

Getgroup is driven by procedure getrow (Figure A.1). Initially getrow is called with the root of the tree as one of its parameters Getrow retrieves all blocks in the group at level 0, then calls itself recursively (if necessary) to retrieve groups from lower levels. Since these groups results from partitioning regions B, C_B or D_B, the dimensions

of the matrix we are dealing with decrease as recursive calls are made, and therefore the number of rows and columns in the region is stored in fields m and n in the tree.

The other complication in getrow is that as recursive calls are made, the row number must be recalculated with respect to the reduced matrix. For example, to retrieve row 3 (with the row numbers starting at zero) of the matrix in Figure 4.3, getrow will first call getgroup, which retrieves blocks 3,4,5 and 16, then call itself recursively to retrieve the blocks from region B. r' (the row to be retrieved from region B) will be 1 in this case. Note that region B (blocks 18, 19 and 20) is a 4x3 matrix as given by the m and n fields of the left son of the root of the tree.

As the correct blocks are retrieved, the proper elements of each block must be stored in correct position in an array, A, such that A[i] is the i^{th} element in the given row. To illustrate this process, consider accessing row 5 in Figure 4.3. Getrow will call getgroup which retrieves blocks 6,7,8 and 16. Getblockcm will be called to store the proper two elements from blocks 6,7 and 8 in array A and leave one position empty after each pair. Then getblockrm will store the appropriate element from block 16. Getrow is now called twice recursively, once to fill in region B (block 19) and once to fill in D_B (block 21). These operations are controlled by the "start" and "step" fields in the tree. The positions to be filled by a call are always those i such that i - start = (step-1) mod step. The call to fill in region B will reference the node with start = 0 and step = 3 and hence will fill in positions 2,5 and 8, and similarly for the call for region D_B. It is easily seen that the set of positions to be filled in by a given call can be described

in this manner by start and step and that these two fields can easily be calculated when the tree is built.

A final issue is the amount of space required by this data structure. The following theorem shows it requires a very modest amount of storage. For reasonable m,n and s it is negligible compared to storage for a single row or column.

Theorem 4.3: For an mxn matrix, the tree described above will have at most

$$(a + b - 1)\log_b n + (b-1)$$

nodes. For large m and n this is approximately $2\sqrt{s}\ \log_b n$.

Proof: Let d(m,n) be the number of nodes in the tree for an mxn matrix. It is clear from the modified version of Algorithm B that

$$d(m,n) \leq n < b \text{ if } n < b. \tag{1}$$

In this case, regions A,B and C will be empty. The algorithm will be recursively called to partition region D_B, whose width decreases by at least one at each call. Similarly,

$$d(m,n) \leq m < a \text{ if } m < a \tag{2}$$

Now if $m \geq a$ and $n \geq b$, region A will be non-trivial. The tree will consist of a root, the tree for region B, whose dimensions are $(ab-s)\left\lfloor \frac{m}{a} \right\rfloor \times \left\lfloor \frac{n}{b} \right\rfloor$, and the trees for C and D whose size is bounded by (1) and (2). This gives

$$d(m,n) \leq 1+d((ab-s)\left\lfloor \frac{m}{a} \right\rfloor, \left\lfloor \frac{n}{b} \right\rfloor) + (a-1)+(b-1) \text{ if } m \geq a \text{ and } n \geq b$$

from which the result clearly follows. □

5. Acknowledgement

I would like to thank Jay Misra for his helpful discussions on this paper.

APPENDIX

```
(* We assume the following declarations for the tree:

type node = record
                m,n,start,step,offset : integer;
                b,cb,db : ↑node
            end;
     ptnode = ↑node;                                              *)

(* Procedure getrow retrieves row r from disk and assigns it to
    array A[0..n-1]                                               *)

procedure getrow(r : integer; p : ptnode);
begin
getgroup(r,p);
if r does not intersect region C then begin
        if r intersects region B then begin
                calculate r';
                getrow(r',p↑.b);
        end;
        if r intersects region D_B then begin
               calculate r'';
                getrow(r'',p↑.db);
        end;
end
else begin
        if r intersects region C_B then begin
               calculate r''';
                getrow(r''',p↑.cb);
end;
end;
```

Figure A.1
Procedure getrow

To retrieve row r from a matrix stored according to Algorithm B, Getrow(r,rootptr) is called where rootptr is a pointer to the root of the tree. a,b and s are global constants whose values are defined at the beginning of Section 4.

r',r'' and r''' give the number of the row that corresponds to row r in regions B, D_B and C_B respectively.

```
(* We assume the global declaration:
        type blocktype = array[0..s-1] of T;
   where elements are of type T                                    *)

(* procedure getgroup retrieves all elements in row r in the group
   associated with the node in the tree pointed to by p.           *)

procedure getgroup(r : integer; p : ptnode);
var i,j,dstart,y,z,blocklen : integer;
    m,n,start,step,offset : integer;
    block : blocktype;
begin
m      := p↑.m;
n      := p↑.n;
start  := p↑.start;
step   := p↑.step;
offset := p↑.offset;

y := m mod a;
z := n mod b;
j := start+step-1;

dstart :=⌊m/a⌋*⌊n/b⌋;
if y <> 0 then begin
        dstart := y*⌈s/y⌉;
        if n mod ⌈s/y⌉ <> 0 then dstart := dstart+1;
end;

if r <=⌊m/a⌋*a-1 then begin
        for i :=⌊r/a⌋*⌊n/b⌋ to (⌊r/a⌋+1)*⌊n/b⌋-1 do begin
                RETRIEVE block i+offset;
                getblockcm(a,b,j,step,r mod a,block);
                j := j + step*b;
        end;
        if z <> 0 then begin
                RETRIEVE block dstart +⌊r/⌈s/z⌉⌋+offset;
                getblockrm(⌈s/z⌉,z,j,step,r mod ⌈s/z⌉,block);
        end;
end
else begin
        for i := ⌊m/a⌋*⌊n/b⌋ to dstart-2 do begin
                RETRIEVE block i+offset;
                getblockcm(y,⌈s/y⌉,j,step,r-⌊m/a⌋*a,block);
                j := j + step*⌈s/y⌉;
        end;
        if n mod ⌈s/y⌉= 0 then blocklen := ⌈s/y⌉
                          else blocklen := n mod ⌈s/y⌉;
        RETRIEVE block dstart-1+offset;
        getblockcm(y,blocklen,j,step,r-⌊m/a⌋*a,block);
end;
end;
```

Figure A.2
Procedure getgroup

```
(* Access all elements in row r of a block of dimension m x n
   where the elements are stored in row major order          *)

procedure getblockrm(m,n,j0,step,r : integer; block : blocktype);
var col,k : integer;
begin
k := r * n;
col := 0;
while (k <= s) and (col < n) do begin
        A[j0] := block[k];
        k := k + 1;
        j0 := j0 + step;
        col := col + 1;
end;
end;
```

```
(* Access all elements in row r of a block of dimensions m x n
   where the elements are stored in column major order       *)

procedure getblockcm(m,n,j0,step,r : integer; block : blocktype);
var col,k : integer;
begin
k := r;
col := 0;
while (k <= s) and (col < n) do begin
        A[j0] := block[k];
        k := k + m;
        j0 := j0 + step;
        col := col + 1;
end;
end;
```

Figure A.3
Procedures getblockrm and getblockcm