AN SMART ALGORITHM FOR TRANSPOSING A MATRIX

T. Hürlimann

Working Paper No. 196

February 1992

INSTITUTE FOR AUTOMATION AND OPERATIONS RESEARCH

University of Fribourg

CH-1700 Fribourg / Switzerland

Bitnet: HURLIMANN@CFRUNI51

phone: (41) 37 21 95 60 fax: 37 21 96 70

An smart Algorithm for Transposing a Matrix

Tony Hürlimann, Dr. rer. pol.

Key-words: transposed matrix, permutation, algorithm.

Abstract: This paper presents a fast algorithm written to transpose a m×n matrix A. The matrix elements a_{ij} are supposed to be in a sequential, lexicographic order stored in an array in the random access memory. The transposed matrix will replace the original. A short analysis of the algorithm is given and the source code in PASCAL and C can be found at the end of the paper. The idea of the algorithm can also be used to permute a given vector. The algorithm was used as a subroutine within the LPL compiler program.

Stichworte: Transponierte Matrix, Permutation, Algorithmus.

Zusammenfassung: Es wird in diesem Papier ein Algorithmus vorgestellt, welcher eine m×n Matrix A transponiert. Es wird angenommen, dass die Matrix in einem eindimensionalen Feld sequentiell und lexiko-graphisch im Hauptspeicher vorliegt. Das Orginal wird überschrieben. Der Programmkode liegt in PASCAL und C vor und kann am Ende des Papiers konsultiert werden. Die Idee des Algorithmus kann auch für die Permutation eines Vektors verwendet werden. Der Algorithmus wurde als eine Subroutine im LPL Kompilerprogramm verwendet.

1 PROBLEM

Suppose a m×n matrix A is given. The elements may be of any type. A element in row i and column j is noted as a_{ij}. (note that the first row and column is zero). The matrix is *not* stored as a two-dimensional array as it may be typically the case in most application. So the matrix is *not* stored in the following PASCAL type TM

```
TYPE
  TM : array[0..m-1,0..n-1] of ElementType;
```

If stored in the TM type, the task would be trivial: exchange each element with its transposed element directly. This can be done, because a_{ij} and a_{ji} can be addressed directly and they can be exchange without further computation.
The matrix is stored in a one-dimensional array or in vector X, where the number of the elements must be at least m*n. So suppose M is stored in lexicographic order in X, where X is of type TX which is defined as following using PASCAL syntax

```
TYPE
  TX : array[0..m*n-1] of ElementType;
```

The question may arise, why a matrix has to be stored in a one-dimensional array rather they in a two-dimensional array. One may argue, that in a two-dimensional array the programmer has to fix *two* (the maximal row numbers and the maximal column numbers), not only *one* constant (the maximal length of the vector TX). This argument is serious, if we work with different matrices dimensions: a matrix with several 100 of rows but very few columns, or a huge number of columns and only some rows. A even more serious argument is, that a one-dimensional vector can be easily created at run-time (especially in C), whereas a two-dimensional representation is more difficult to manage at run-time. This frees the programmer even to fix any maximal length at compile-time.
So we suppose the following data structure (written in PASCAL):

```
CONST
  MAXLENGTH = ...;
TYPE
  TX : array[0..MAXLENGTH] of ElementType;
VAR
  X : TX;
  m,n : integer;
```

where MAXLENGTH is the maximum length of the array, X is the array, m is the number of rows, and n the number of columns within the matrix A. Of course, *MAXLENGTH >= n*m* must hold. We store the element a_{ij} at position *i*n+j* within the array X (note that the first position is zero). So to copy a two-dimensional array to a one-dimensional and vice versa, we need the following procedures *M2V* and *V2M*:

```
procedure M2V(var M:TM; var X:TX; n,m:integer);
```

```
{ copy Matrix to Vector }
var i,j:integer;
begin
for i:=0 to m-1 do
  for j:=0 to n-1 do
    V[i*n+j] := a[i,j];
end;


procedure V2M(var M:TM; var X:TX; n,m:integer);
{ copy Vector to Matrix }
var i,j:integer;
begin
for i:=0 to m-1 do
  for j:=0 to n-1 do
    a[i,j] := V[i*n+j] ;
end;
```

The matrix elements a_{ij} are stored in the sequential order within the memory cells (figure 1).

a_{00} 1	a_{01}	a_{02}	...	a_{0n-1}	a_{10}	a_{11}	...	a_{20}	a_{21}	...	a_{m-1n-}

Figure 1.

Transposing the matrix means to produce the resulting order as in figure 2.

a_{00} 1	a_{10}	a_{20}	...	a_{m-10}	a_{01}	a_{11}	...	a_{02}	a_{12}	...	a_{m-1n-}

Figure 2.

The proposed transposing algorithm may also be used for the following problem: Given 21 cubes, seven of them are red, seven are blue, and seven are yellow. Within each color, the cubes are numbered from 1 to 7. They are arranged in a line by colors and within the colors by the number as in figure 3.

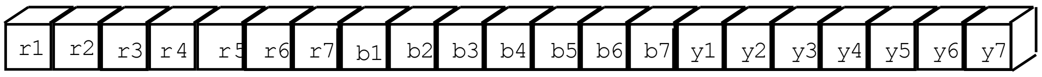

Figure 3.

Sort the line such that the order in figure 4 will by obtained.

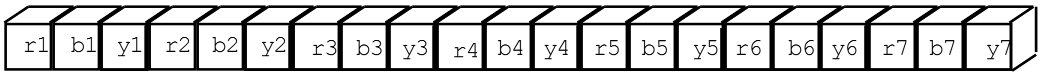

Figure 4.

The numbers are sorted together and they are arranged in the same order of color as in the first line. Note that this problem is the same as the original problem where we transpose a matrix.

2. SOLUTION

To solve this problem, one must think about the process, how to move an single element from one place its original or initial place to its destination place. There exists a function y=f(x), which calculates the destination place y after the move of a element at a initial position x. The function is

$$y = (x/n)+(x\%n)*m \qquad (1)$$

where / is the integer division operator, % is the modulo operator, and * is the multiply operator.
In other words: Each element at a initial position *x* within the array V must be moved to position *(x/n)+(x%n)*m* to fulfill the new arrangement to get the transposed order of the matrix. (Note that the first position is zero).

The inverse function x=g(y) can also be found. It can easily be proven that it is

$$x = (y/m)-(y\%m)*n \qquad (2)$$

The function x=g(y) calculates the position x, where an array element comes from, to be placed at position y within the new arrangement.

With these formulas one may easy verify that the cube b2 in our example, initially placed at position 8, must move to position (8/7)+(8%7)*3 = 1+1*3 = 4 (using formula 1). On the other hand, the cube y2, which is moved to position 5, was initially in position (5/3)+(5%3)*7 = 15 (using formula 2). Two elements never move: the first and the last. This may be verified using the formula (1).

element at position 0 :	new position = (0/n) + (0%n)*m = 0
element at position n*m-1 :	new position = ((n*m-1)/n) + ((n*m-1)%n)*m =
	= m-1 + (n-1)*m = n*m-1

The proof that the formula hold for every position can be formulated as following

Every element a_{ij} is at position i*n+j and must be moved to the position of the element a_{ji} with is $j*m+i$. Therefore, we have $x=i*n+j$ and $y=j*m+i$. Since *i=x%n* and *j=x/n* hold for every position x , as well as *i=y/m* and *j=y%m* hold for every position y, the formulas follows immediately. End of proof.

The formulas suggest an simple algorithm: Take every element for left to right and copy it to a new array at the right place. Then copy the array back to the original. This is what the *CopySort* procedure below does. This algorithm has the complexity of O(m*n), which is the same as if the data were stored in two-dimensional array. The obvious disadvantage of this algorithm is, that a second array is used to hold the copy.

In fact, we do not need to copy the data. The original array can be used to arrange the data in place. This is used in the *SmartSort* procedure below. SmartSort only needs a additional cell *Temp* of type *ElementType* to hold a temporary copy of an element.

```
VAR Temp : ElementType;
```

From a given position y, calculate the original position x using the function *x=(y/m)+(y%m)*n*. The element at position x must be moved to position y. Before we move the element, the element at position y is copied to *Temp* in order to save the element. Now the element at position x is moved to position y. Then we go to position x and calculate the position z, where the element comes from using again the formula. The element is again moved from position z to x without any further instruction. At the end, we must come back to position y where we started. The resting element in Temp must now be placed at position w=(y/n)+(y%n)*m.
We have executed the following program

1. start from any position y
2. copy the element at position y to Temp
3. copy the index y to a (a:=y;)
4. calculate an new position x = (y/m) + (y%m)*n
5. if x=a then goto 8 else goto 6
6. copy the element at position x to position y
7. assign : y:=x (y becomes x) and goto 4
8. copy Temp to the position y

Note that no exchanges are needed except at the beginning where the element is stored to the Temp variable and at the end where the inverse copy is executed. All other elements are just moved. This is three times quicker than an exchange process of element a_{ij} with a_{ji} in a two-dimensional data-structure.

Unfortunately, this moves attain only a limited numbers of elements. The moved element form a circle within the array. This may be shown using the cube example. Figure 5 shows one of the circles.

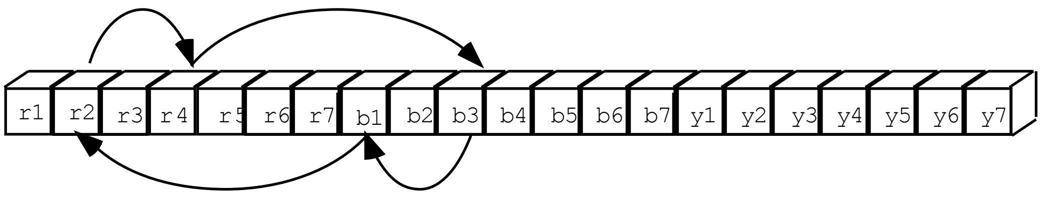

Figure 5.

Element r2 at position 1 is moved to position 4, where r4 is placed, r4 in turn is moved to position 9 where b3 is placed, b3 is moved back to position 7 where b1 is placed, and b1 is moved to the initial position 1, where the circle started. This circle involves four elements. There are also circle involving only one element. The first and the last elements, which are not moved at all, may be interpreted to be moved around a circle with only one position.
All the moves are visualized with the figure 6.

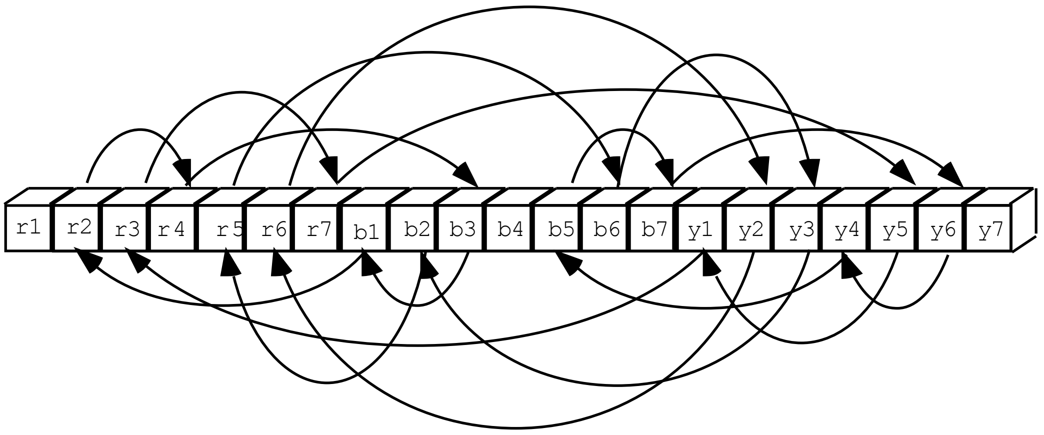

Figure 6.

To move all elements to the right place, all circles must be processed with the program above. To be sure that each element is moved to the right place and all circles are found, the algorithm proceeds from left to right in the array. The following algorithm will do this:

1. initialize i:=0 and j:=1 (we skip position 0, and begin directly with position 1)
2. Start a circle: Calculate each position of the elements in circle which starts with position j using the formulas, but do not move any element. If at least one element in the circle was found having a position lower than the starting position j, then the circle has already been processed. Skip it an goto 5 otherwise goto 3
3. move the elements within the circle using the algorithm above, add the

number of moves to i (i:=i+number_of_moves_in_that_circle).
4. if i>=n*m-2 STOP, all elements have been moved else goto 5.
5. j:=j+1 and goto 2

Example using the cube problem: n=7 m=3:

1. Leave position 0 at that place and set i:=0.
2. Begin with position j:=1 (r2). This gives the circle positions: 1-3-9-7-1, with the elements: r2-r4-b3-b1. There is no position less than j (=1) in the circle, so move them. Since 4 moves have taken place, add 4 to i, (now i = 4).
3. Proceed to the next position j=2. This yields the circle: 2-18-14-6-2 with the elements r3-r7-y5-y1. Again all position are greater or equal j, so move them and add 4 to i (i=8).
4. Now go to position j=3, which yields the circle positions: 3-9-7-1-3. There is a position within this circle which is less than j. The circle has already been processed. Skip it.
5. The next circle begins with j=4, It contains the elements r5-b6-y3-b2 at position 4-12-16-8. Move them and add 4 to i (i=12).
6. The next circle is r6-y2 at positions 5-15. Move the elements and add 2 to i (i=14).
7. There is no circle to move up to element b4 at position 10. It is a circle of one element. No move is needed since b4 stay at its place, but we count it as a move in i (i=15).
8. The only circle to move is b5-b7-y6-y4 at positions 11-13-19-17. Move it and set i=19.
9. since i>=n*m-2 if true, the whole algorithm can be aborted, because all elements have been moved, although j has only the value 11. There is no need to process the resting elements.

The algorithm is actually used in the LPL compiler, to arrange some data tables in the symbol table.

An interesting point is, that the original arrangement is restored, if the algorithm is applied *two* times with the m and n value interchanged. Therefore, it is rather easy to test, whether the algorithm works correctly: just apply the procedure two times to the same data and compare the resulting ordering with the original. They must be the same.

4. ANOTHER APPLICATION OF THE ALGORITHM

The idea of the proposed algorithm can also be used to perform permutations of arrays with N elements, a task often found in matrix manipulation. It is well known, that for the Gaussian elimination process, one often needs to permute some rows or columns in order to keep the fill-in low and to garantee a maximum numerical stability. If the rows or columns must be physically exchanged, this may be done with the proposed algorithm.

As an example, take an array of 21 entries in the order 1-2-3-...-21. Permute the elements to the order 10-6-17-11-4-16-15-9-1-2-18-21-20-3-12-19-5-14-7-13-8. We call this order a permutation vector P1. Figure 7 shows all the moves within the array which take place to fulfil the requirements.

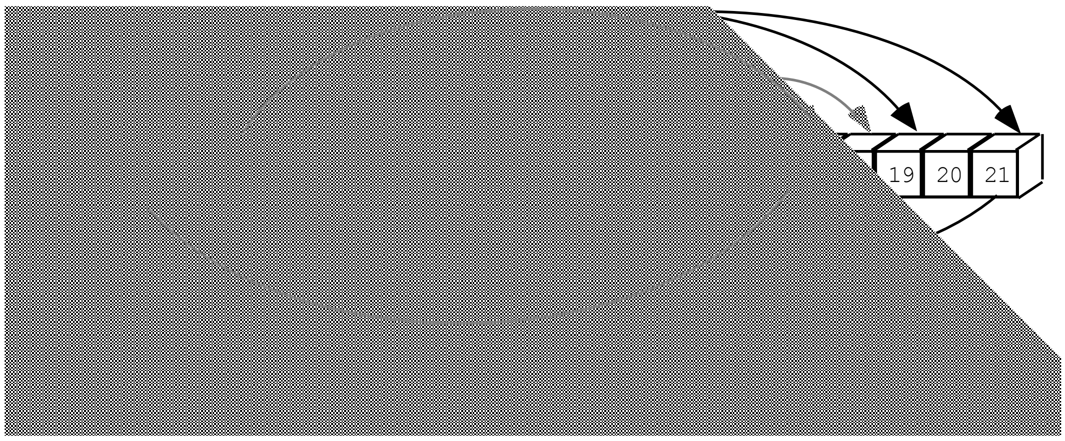

Figure 7.

The rearrangement using our algorithm will produce three circles:

a) 1-9-8-21-12-15-7-19-16-6-2-10-1 (containing 12 elements)
b) 3-14-18-11-4-5-17-3 (containing 7 elements)
c)= 13-20-13 (containing 2 elements).

Two modification are necessary to adapt the algorithm to the permutation problem:
- The 'source' position is not calculated using formula 2, but is given directly by the permutation vector.
- The algorithm must stop only if *all* elements N are permuted (i>=N) and not only on (n*m-2>=i) as in the first version.

Note, that inverting the process - to find the original arrangement within the array - is not so easy.

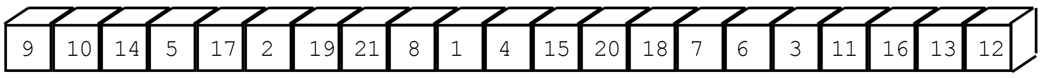

Figure 8.

We need another permutation vector P2 shown in figure 8. Then we can apply the same proposed algorithm to move the elements using the new permutation vector P2. To find the permutation vector P2, each element in the original permutation vector P1 is moved backwards by one position within the circle. This can be done by moving forwards within the circle and to remember the last position to move the element backwards. So we *must* traverse the circle in the opposed direction than in the original algorithm, which means that three copies per move within the circle must be made. This is awful and not very handy. An alternative would be to make a copy of the permutation vector P1 using the following loop:

```
for i:=0 to N-1 do
  CopyVector[Original[i]]:=Original[i];
```

This would cost much less and the algorithm is simple!

The permutation algorithm is useful for a matrix permutation, however. Given the following matrix manipulation

$$B = PAQ$$

where A is a arbitrary matrix, P and Q are permutation matrices, and B is its resulting permuted matrix. The algorithm is first applied to permute the rows using the Q matrix. In a second step, the columns are permuted using the P matrix. The permutation is executed in place, there is no need to copy a row or a column to a auxiliary storage an back to the destination row or column.

4. THE ALGORITHM CODED IN C

```
/* this program sort an array rgbrgbrgbrgbrgb to rrrrrgggggbbbbb */

  char TX[1000];
  char temp;
  int m,n;

void createarray()
{
  int i;
  for (i=0; i<m*n; i++)     TX[i]=(char)(48+i/n+16*(i%n));
  TX[m*n]='\x0';
}

void printarray()
{
  printf("\n%s\n",TX);
}

void smartsort(int m, int n)
{
  int i,j,x,y;

  x=j=1; i=0;
  temp=TX[x];
  while (1) {
    y=(x/m)+(x%m)*n;
    if (j==y) {
      TX[x]=temp; i++;
      if (i>=n*m-3) return;
      x=++j;   /* j++; x=j; */
      while (j<n*m) {
        y=(x/m) + (x%m)*n;
        x=y;
        if (y<j) x=++j;
        else if (y==j) break;
      }
      temp=TX[x];
    }
    else {
      TX[x]=TX[y]; i++;
      x=y;
    }
  }
}

void copysort(int m,int n)
{
  int i;
  for (i=0; i<m*n; i++)     TX[m*n+(i/n)+(i%n)*m]=TX[i];
  for (i=0; i<m*n; i++)     TX[i]=TX[m*n+i];
  TX[m*n]='\x0';
}

main()
{
  puts("Enter two positive integer: ");
  scanf("%d %d",&m,&n);
  if (m<1) m=2;
  if (n<1) n=1;
  if (m*n>1000) n=1000/m;
  createarray();
  printarray();
  smartsort(m,n);
  printarray();
}
```

5. THE ALGORITHM CODED IN PASCAL

```
(* this program sort an array rgbrgbrgbrgbrgb in rrrrrgggggbbbbb *)

var
  TX: array[1..1000] of char;
  temp:char;
  m,n:integer;

procedure createarray;
var
  i:integer;
begin
  for i:=0 to m*n-1 do begin
    TX[i+1]:=char(48+i div n+16*(i mod n));
  end;
end;

procedure printarray;
var i:integer;
begin
  writeln;  for i:=1 to m*n do write(TX[i]);  writeln;
end;

procedure smartsort(m,n:integer);
var
  i,j,x,y:integer;
  flag:boolean; {needed to break to while loop}
begin
  x:=1; j:=1; i:=0;
  temp:=TX[x+1];
  while true do begin
    y:=(x div m)+(x mod m)*n;
    if j=y then begin
      TX[x+1]:=temp; i:=i+1;
      if i>=n*m-3 then exit;
      j:=j+1; x:=j;
      flag:=true;
      while (j<n*m) and flag do begin
        y:=(x div m) + (x mod m)*n;
        x:=y;
        if y<j then begin j:=j+1; x:=j; end
        else if y=j then flag:=false; {break};
      end;
      temp:=TX[x+1];
    end
    else begin
      TX[x+1]:=TX[y+1]; i:=i+1;
      x:=y;
    end;
  end;
end;

begin
  writeln('Enter two positive integer: ');
  read(m); read(n);
  createarray;
  printarray;
  smartsort(m,n);
  printarray;
end.
```