Data Structure and Algorithm

Class 3

Seongjin Lee
Updated by Songsub Kim

insight@gnu.ac.kr
http://resourceful.github.io
Systems Research Lab.
GNU

Table of contents

Array

Array I

an **array** is a set of pairs, **<index, value>**, such that each index that is defined has a value associated with it

- "a consecutive set of memory locations" in C
- logical order is the same as physical order

operations

- creating a new array
- retrieves a value
- stores a value
- insert a value into array - delete a value at the array

ADT: Array

- **Object**: A set of pairs <*index, value*> where for each value of *index* ther is a value from the set *item*. *Index* is a finite ordered set of one or more dimensions.
- **Functions**: for all $a \in Array, i \in index, x \in item, j, size \in$ integer
  - *Array* `create(j, list)`: **return** an array of j dimension
  - *Item* `Retrieve(A, i)`: **if** ($i \in index$) **return** the item in index value i in array A **else return** error
  - *Array* `Store(A, i, x)`: **if** (i in $index$) **return** an array that is identical to array A excep the new pair $< i, x >$ has been inserted **else return** error

Array III

An one-dimensional array in C is declared implicitly by appending brackets to the name of a variable

```
int list[5], int *plist[5];
```

Always remember that starting index of array is `0` in C

Array IV

Let’s consider implementing an one-dimensional arrays

```
int list[5];
```

- allocates five consecutive memory locations
- each memory location is large enough to hold a single integer
- base address is address of the first element

```
list = &list[0]
```

list[0]	list[1]	list[2]	list[3]	list[4]
trash value	trash value	trash value	trash value	trash value

Array V

Variable	Memory Address
`&list[0]`	base address = α
`&list[1]`	α+ `sizeof(int)`
`&list[2]`	$\alpha + 2\cdot$ `sizeof(int)`
`&list[3]`	$\alpha + 3\cdot$ `sizeof(int)`
`&list[4]`	$\alpha + 4\cdot$ `sizeof(int)`

- Because different architectures have different int sizes, we have to use "sizeof"

`&list[i]` in a C programs

- C interprets it as a pointer to an integer or its value

Array VI

```
int *list1; // pointer variable to an int
```

```
int *list2[5]; // list2 : pointer constant to an int,
                  // and five memory locations for holding
                  // integers are reserved
```

```
int list[] = { 0,1,2,3,4,5 };
^^I
printf("%d\n", list[3]); // 3
printf("%d\n", *(&list[0]+3)); // 3
printf("%d\n", *(list+3)); // 3 ...All is the same expression
```

`(list2+i)` equals `&list2[i]`, and `*(list2+i)` equals `list2[i]`

- regardless of the type of the array list2

Array VII

How about this ?

```
printf("%d\n", ++list[0]);
```

- `++list[0] -> list[0]+1`
- the result is same as `list[1]`

Array VIII

consider the way C treats an array when it is a parameter to a function

- the parameter passing is done using call-by-value in C
- but array parameters have their values altered

Practice Problem (5 min)

Array: Ex. 2.1 Analyze and comprehend the code before running it

```
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
int i;
void main(void) {
    for(i = 0; i < MAX_SIZE; i++)
    ^^Iinput[i] = i;
    answer = sum(input, MAX_SIZE);
    printf{"The sum is: %f\n", answer);
}

float sum(float list[], int n) {
    int i;
    float tempsum = 0;
    for(i = 0; i < n; i++)
        tempsum += list[i];
    return tempsum;
}
```

Recap: On Pointer I

Pointer Variable stores address

- & : Starting address of allocated variable
- * : Value stored on the address of the pointer variable

```c
// pointer_variable.c
#include <stdio.h>

int main(){
    int nNum, *pNum;

    nNum = 10;
    pNum = &nNum;

    printf("nNum = %d, &nNum = %d \n", nNum, (int)&nNum);
    printf("pNum = %d, pNum = %d, &pNum = %d \n",
                         *pNum, (int)pNum, (int)&pNum);
}
```

Recap: On Pointer II

Do Not!

- pointer variable is not referencing an address, so cannot store a value

```
int *ptr;
*ptr = 100;
```

It is better to handle NULL for the pointer that do not refer to address right away

Recap: On Pointer III

Do Not!

- the data type must equal

```
double Pi = 3.14;
int *pPi = &Pi;
```

- cannot dereference a non-pointer variable

```
int num;
*num = 100;
```

Recap: On Pointer IV

Do!

- it is recommended to initialize a pointer value with NULL ('\0') See the example

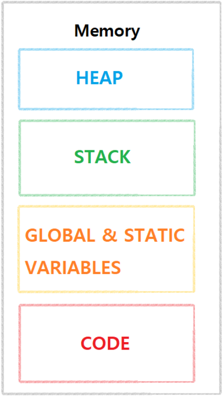
```c
// null_pointer.c
#include <stdio.h>

int main(){
    int *pNum = NULL, Num=103;
    if (pNum == '\0')
        pNum = &Num;
    else
        *pNum = 100;
    printf("pNum %d", (int)*pNum);
    return 0;
}
```

Recap: On Pointer V

```
#include <stdlib.h>

void *malloc(size_t size); // allocates size bytes of memory and returns
     a pointer to the allocated memory

void *free(void *ptr); // frees allocation that were created via the
    preceding allocation function

void *calloc(size_t count, size_t size); // contiguously allocates
    enough space for count objects that are size bytes of memory each
    and returns a pointer to the allocated memory. The allocated memory
     is filled with bytes of value zero.

void *realloc(void *ptr, size_t size); // change the size of the
    allocation pointed to by ptr to size, and returns ptr
```

Recap: On Pointer VI

if the memory is not freed after being allocated, a memory leak will occur



Practice Problem (5 Min)

Array: Ex 2.2, 1-dimensional array addressing

```
int one[] = {0, 1, 2, 3, 4};
```

write a function that prints out both the address of the i^{th} element of the array and the value found at this address

Practice Problem (Solution) I

Array: Ex 2.2, 1-dimensional array addressing

```c
// codes/array_address.c
#include <stdio.h>

void print1(int *ptr,int rows) {
    int i;
    printf("Address\t\tContents\n");
    for(i=0;i<rows;i++)
        printf("%8u\t%5d\n", (unsigned int)ptr+i, *(ptr+i));
    printf("\n");
}

int main() {
  int one[] = {0, 1, 2, 3, 4};
  print1(one, 5);
  return 0;
}
```

One-dimensional array accessed by address

- address of i^{th} element `ptr + i`
- obtain the value of the i^{th} value `*(ptr + i)`

Practice Problem (Solution) II

```
Address^^I^^IContents
1518325632^^I 0
1518325633^^I 1
1518325634^^I 2
1518325635^^I 3
1518325636^^I 4
```

One-dimensional array addressing

- the addresses increase by two on an Intel 386 machine
- Example shown is the result of Mac OS X on Intel Core i5 Machine

Structures and Unions

Structures and Unions: struct I

struct

- structure or record
- the mechanism of grouping data
- permits the data to vary in type

collection of data items where

- each item is identified as to its type and name

Structures and Unions: struct II

```
struct {
    char name[10];
    int age;
    float salary;
} person;
```

creating a variable

- whose name is person and
- has three fields
    1. a name that is a character array
    2. an integer value representing the age of the person
    3. a float value representing the salary of the individual

Structures and Unions: struct III

use of the .(period) as the structure member operator

```
strcpy(person.name, "james");
person.age = 30;
person.salaray = 35000;
```

- select a particular member of the structure

Structures and Unions: struct IV

`typedef statement`

- create our own structure data type

type 1

```
typedef struct human_being {
    char name[10];
    int age;
    float salary;
} human;
```

type 2

```
typedef struct {
    char name[10];
    int age;
    float salary;
} human_being;
```

Structures and Unions: struct V

`human_being`

- the name of the type defined by the structure definition

```
human_being person1, person2;

if(strcmp(person1.name, person2.name))
    printf("The two people do not have the same name\n");
else
    printf("The two people have the same name\n");
```

Structures and Unions: Assignment

assignment

- permits structure assignment in ANSI C

```
    person1 = person2;
```

- but, in most earlier versions of C assignment of structures is not permitted

```
     strcpy(person1.name,person2.name);
     person1.age=person2.age;
     person1.salary=person2.salary;
```

equality or inequality

- cannot be directly checked
- Example function to check equality of struct

```
int humans_equal(human_being person1, human_being person2) {
    if(strcmp(person1.name,person2.name))
        return FALSE;
    if(person1.age!=person2.age)
        return FALSE;
    if(person1.salary!=person2.salary)
        return FALSE;
    return TRUE;
}
```

Structures and Unions: Embedding Structure I

Embedding of a structure within a structure

```
typedef struct {
    int month;
    int day;
    int year;
} date;

typedef struct human_being {
    char name[10];
    int age;
    float salary;
    date dob; // embedded structure
};
```

Structures and Unions: Embedding Structure II

Ex. A person born on Feb 14 1992

```
human_being person1;
person1.dob.month = 2;
person1.dob.day = 14;
person1.dob.year = 1992;
```

Structures and Unions: Unions I

Unions

- similar to a structure, but
- the fields of a union must share their memory space
- only one field of the union is "active" at any given time

Structures and Unions: Unions II

```
typedef struct sex_type {
    enum tag_field {female,male} sex;
    union {
        int children;
        int beard; } u;
};

typedef struct human_being {
    char name[10];
    int age;
    float salary;
    date dob;
    sex_type sex_info;
};

human_being person1,person2;
```

Structures and Unions: Unions III

Assign values to person1 and person2

```
person1.sex_info.sex = male;
person1.sex_info.u.beard = FALSE; /* FALSE: 0 */
```

and

```
person2.sex_info.sex = female;
person2.sex_info.u.children = 4;
```

Differences between structure and union I

What are differences between structure and union ?

- Structure

```
typedef struct example{
^^Iint x, y;
^^Idouble d;
}struct_example;

struct_example se;

printf("%d\n", sizeof(se));
printf("%p %p %p\n", &se.x, &se.y, &se.d);
```

» 16
» 0x7ff0 0x7ff4 0x7ff8

- 16byte size, different address

Differences between structure and union II

- Union

```
^^Itypedef union example{
^^I^^Iint x, y;
^^I^^Idouble d;
^^I}union_example;

^^Iunion_example ue;
^^I
^^Iprintf("%d\n", sizeof(ue));
^^Iprintf("%p %p %p\n", &ue.x, &ue.y, &ue.d);
```

» 8
» 0x7ff0 0x7ff0 0x7ff0

- 8byte size, same address

Differences between structure and union III

```
struct {
    int i, j; float a, b;
}
```

or

```
struct {
    int i; int j; float a; float b;
};
```

stored in the same way

- increasing address locations in the order specified in the structure definition

size of an object of a struct or union type

- the amount of storage necessary to represent the largest component

- one or more of its components is a pointer to itself
- usually require dynamic storage management routines to explicitly obtain and release memory

```
typedef struct list {
    char data;
    list *link;
};
```

the value of `link`

- address in memory of an instance of `list` or `null` pointer

```
list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.link = item3.link = NULL;
```

Sparse Matrices

Sparse Matrices : The Abstract Data Type

The Abstract Data Type I

Matrix is a mathematical object that is used to solve many problems in the natural sciences

- our interest centers not only on the specification of an appropriate ADT
- but also in finding representations that let us efficiently perform the operations described in specification

The Abstract Data Type II

A matrix contains m rows and n columns of elements

- write as $m \times n$ and read as m by n (m rows, n columns)
- use two-dimensional array
- space complexity
  $S(m, n) = m * n$

$$
\begin{array}{cc}
 & \begin{array}{ccc} col0 & col1 & col2 \end{array} \\
\begin{array}{c} row0 \\ row1 \\ row2 \\ row3 \\ row4 \end{array} &
\begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{bmatrix}
\end{array}
$$

The Abstract Data Type III

When a matrix is represented as a two-dimensional array defined as `a[MAX_ROWS][MAX_COLS]`

- we can locate quikcly any element by writing `a[i][j]`
- i is the row index, j is the column index

The Abstract Data Type IV

There are some problems with `a[i][j]` notation.

- a matrix with many zero's: *sparse matrix*

`A[6,6]`

$$
\begin{array}{c c}
 & \begin{array}{cccccc} col0 & col1 & col2 & col3 & col4 & col5 \end{array} \\
\begin{array}{c} row0 \\ row1 \\ row2 \\ row3 \\ row4 \\ row5 \end{array} &
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & 15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

The Abstract Data Type V

common characteristics of a sparse matrix

- most elements are zero's
- inefficient memory utilization

solutions

- store only nonzero elements
- using the triple `<row,col,value>`
- must know
  - the number of rows
  - the number of columns
  - the number of non-zero elements

The Abstract Data Type VI

We first must consider the operations that we want to perform on these Matrices

- matrix creation
- addtion
- muliplication
- transpose

The Abstract Data Type VII

ADT: *Sparse Matrix*

- objects: a set of triples $< row, column, value >$, where row and $column$ are integers and form a unique combination, and $value$ comes from the set $item$
- Functions: for all $a, b \in SparseMatrix, x \in item, i, j, maxCol, maxRow \in index$

The Abstract Data Type VIII

- *SparseMatrix Create(maxRow, maxCol)*
  - Return: a *SparseMatrix* that can hold up to $maxItems = maxRow \times maxCol$ and whose maximum row size is $maxRow$ and whose maximum colum size is $maxCol$
- *Sparse Matrix Transpose*
  - Return: the matrix produced by intechaging the row and column value of every triple
- *SparseMatrix Add(a, b)*
  - Return: if the dimensions of a and b are the same
    return the matrix produced by adding corresponding items, namely those with identical row and $column$ values
    else return error
- *SparseMatrix Multiply(a,b)*
  - Return: if number of columns in a equals number or rows in b
    return the matrix d produced by multiplying a by b according to the formula: $d[i][j] = \sum(a[i][k] \cdot b[k][j])$ where $d(i, j)$ is the (i, j)th element
    else return error

Sparse Matrices : Sparse Matrix Representation

Sparse Matrix Representation I

Before implementing any of the ADT operations

- we must establish the representation of the sparse matrix
- We can characterize unquely any element within a matrix by using the triple $< row, col, value >$

Other considerations

- We want transpose operation to work efficiently, we should organize the triples so that the row indices are in ascending order
- Also requiring that all the triples for any row be stored so that the column indices are in ascending order
- To ensure that the operations terminate, we must know the number of rows and columns, and the number of nonzero elements in the matrix

Sparse Matrix Representation II

SparseMatrix `Create(maxRow, maxCol):`

```
#define MAX_TERMS 101 /* max number of terms +1 */

typedef struct {
    int col;
    int row;
    int value;
} term;

term a[MAX_TERMS];
```

- a[0].row: the number of rows
- a[0].col: the number of columns
- a[0].value: the total number of non-zeros
- choose row-major order

Sparse Matrix Representation III

	row	col	value
a[0]	6	6	8
a[1]	0	0	15
a[2]	0	3	22
a[3]	0	5	-15
a[4]	1	1	11
a[5]	1	2	3
a[6]	2	3	-6
a[7]	4	0	91
a[8]	5	2	28

- space complexity (*variable space requirement)
  `S(m,n) = 3 * t` where
  `t` : the number of non-zero's
- independent of the size of rows and columns

Practice Problem (5 Min)

Write down how the following matrix is represented in our definition of sparse matrix

$$
\begin{array}{c c}
 & \begin{array}{cccccc} col0 & col1 & col2 & col3 & col4 & col5 \end{array} \\
\begin{array}{c} row0 \\ row1 \\ row2 \\ row3 \\ row4 \\ row5 \end{array} &
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & 15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

Sparse Matrices : Transposing A Matrix

Transposing A Matrix I

Transposing the sample matrix

- interchange rows and columns
- move `a[i][j]` to `a[j][i]`

sample matrix

	row	col	value
a[0]	6	6	8
a[1]	0	0	15
a[2]	0	3	22
a[3]	0	5	-15
a[4]	1	1	11
a[5]	1	2	3
a[6]	2	3	-6
a[7]	4	0	91
a[8]	5	2	28

transposed matrix

	row	col	value
b[0]	6	6	8
b[1]	0	0	15
b[2]	0	4	91
b[3]	1	1	11
b[4]	2	1	3
b[5]	2	5	28
b[6]	3	0	22
b[7]	3	2	-6
b[8]	5	0	-15

Transposing A Matrix II

Algorithm: BAD_TRANSPOSE

```
for each row i
    take element <i, j, value>;
    store it as element <j,i,value> of the transpose;
end;
```

The problem

We will not know exactly where to place element $< j, i, value >$ in the transpose matrix until we have processed all the elements taht precede

(0, 0, 15),	which becomes	(0, 0, 15)
(0, 3, 22),	which becomes	(3, 0, 22)
(0, 5, -15),	which becomes	(5, 0, -15)

If we place these triples consecutively in the transpose matrix, then, as we insert new tiples we must move elements to maintain the correct order

Transposing A Matrix III

We can avoid this data movement by using the column indices to determine the placement of elments in the transpose matrix

Algorithm TRANSPOSE

```
for all elements in column j
    place element <i,j,value> in element <j,i,value>
end for;
```

Since the original matrix ordered the rows, the columns within each row of the tranpose matrix will be arraged in ascending order as well

```
void transpose(term a[], term b[]) {
    /* b is set to the transpose of a */
    int n, i, j, currentb;
    n = a[0].value; // total number of elements
    b[0].row = a[0].col; // rows in b = columns in a
    b[0].col = a[0].row; // columns in b = rows in a
    b[0].value = n;
    if(n > 0){ // non zero matrix
        currentb = 1;
        for (i = 0; i < a[0].col; i++)
        /* transpose by the columns in a */
            for(j = 1; j <= n; j++)
            /* find elements from the current column */
                if(a[j].col == i) {
                /* element is in current column, add it to b */
                  b[currentb].row = a[j].col;
                  b[currentb].col = a[j].row;
                  b[currentb].value = a[j].value;
                  currentb++;
                }
            }
    }
}
```

Transposing A Matrix V

Analysis of transpose

Computing Time

- nested **for** loops are the decisive factor
- two **if** statements and several assignments statments requires on constant Time
- outer **for** loop is iterated `a[0].col` times
- inner **for** loop requires `a[0].value` times
- the total time for the nested **for** loop is $columns \cdot elements$: $O(columns \cdot elements)$

The problem

- unnecessary loop for each column

Transposing A Matrix VI

In essence

```
for (j = 0; j < columns; j++)
    for (i = 0; i < rows; i++)
        b[j][i] = a[i][j];
```

The $O(columns \cdot elements)$ time for our transpose function becomes $O(columns^2 \cdot rows)$

- when the number of elements iof the order $columns \cdot rows$

Solution:

- Use a bit more storage

Practice Problem (5 Min)

Find the pros and cons of the transpose algorithm introduced in slide 22

```
void transpose(term a[], term b[]) {
  /* b is set to the transpose of a */
  int n, i, j, currentb;
  n = a[0].value; // total number of elements
  b[0].row = a[0].col; // rows in b = columns in a
  b[0].col = a[0].row; // columns in b = rows in a
  b[0].value = n;
  if(n > 0){ // non zero matrix
      currentb = 1;
      for (i = 0; i < a[0].col; i++)
      /* transpose by the columns in a */
          for(j = 1; j <= n; j++)
          /* find elements from the current column */
              if(a[j].col == i) {
              /* element is in current column, add it to b */
                b[currentb].row = a[j].col;
                b[currentb].col = a[j].row;
                b[currentb].value = a[j].value;
                currentb++;
          }
  }
}
```

Sparse Matrices : Fast Transpose Algorithm

Fast Transpose Algorithm I

Create better algorithm by using a little more storage

- row_terms the number of element in each row
- starting_pos the starting point of each row

We can transpose a matrix represneted as a sequence of triples in $O(columns + elements)$ time

- determining the number of elelments in each column of original matrix (number of elements in each row)
- we can determine the starting position of each row in the transpose matrix
- we can move the elements in the original maxtrix one by one into their correct position

Fast Transpose Algorithm II

```
void fast_transpose(term a[], term b[]){

    /* the transpose of a is placed in b */
    int row_terms[MAX_COL]; // number of rows in column
    int starting_pos[MAX_COL]; // column counts

    int i, j;

    // origianl matrix
    int numCols = a[0].col; // number of columns
    int numTerms = a[0].value; // number of elements

    // transposed matrix
    b[0].row = numCols; // number of rows
    b[0].col = a[0].row; // number of columns
    b[0].value = numTerms; // number of elements

    // for non zero matrix
    if(numTerms > 0) {
        // initializing matrix
        for(i = 0; i < numCols; i++)
            row_terms[i] = 0;

```

Fast Transpose Algorithm III

```
        // number of elements in a row
        for(i = 1; i <= numTerms; i++)
            row_terms[a[i].col]++;

        starting_pos[0] = 1;

        // accounting column position
        for(i = 1; i < numCols; i++)
            starting_pos[i] = starting_pos[i-1] + row_terms[i-1];

        // transposing
        for(i = 1; i <= numTerms; i++){
            j = starting_pos[a[i].col]++;
            b[j].row = a[i].col;
            b[j].col = a[i].row;
            b[j].value = a[i].value;
        }
    }
}
```

Fast Transpose Algorithm IV

Intial values:

```
numCols = a[0].col = 6
numTerms = a[0].value = 8

b[0].row = numCols = 6
b[0].col = a[0].row = 6
b[0].value = numTerms = 8
```

	row	col	value
a[0]	6	6	8
a[1]	0	0	15
a[2]	0	3	22
a[3]	0	5	-15
a[4]	1	1	11
a[5]	1	2	3
a[6]	2	3	-6
a[7]	4	0	91
a[8]	5	2	28

Fast Transpose Algorithm V

Number of elements in a row

```
20  // initializing matrix
21  for(i = 0; i < numcols; i++)
22   row_terms[i] = 0;
23
24  // number of elements in a row
25  for(i = 1; i <= numTerms; i++)
26   row_terms[a[i].col]++;
```

code	line 22	a[i].col	line 26
row_terms[0]	0	1+1	2
row_terms[1]	0	1	1
row_terms[2]	0	1+1	2
row_terms[3]	0	1+1	2
row_terms[4]	0	-	0
row_terms[5]	0	1	1

i	a[i].col	row_terms [a[i].col]++
1	0	1
2	3	1
3	5	1
4	1	1
5	2	1
6	3	2
7	0	2
8	2	2

Fast Transpose Algorithm VI

starting_pos caculation

```
starting_pos[0] = 1;

// accounting column position
for(i = 1; i < numcols; i++)
  starting_pos[i] =
  starting_pos[i-1]+row_terms[i-1];
```

start_pos[i]	start_pos[i-1] +rowterm[i-1]	result
start_pos[0]	1	1
start_pos[1]	1+2	3
start_pos[2]	3+1	4
start_pos[3]	4+2	6
start_pos[4]	6+2	8
start_pos[5]	8+0	8

Transposing routine

```
// transposing
for(i = 1; i <= numTerms; i++){
  j = starting_pos[a[i].col]++;
  b[j].row = a[i].col;
  b[j].col = a[i].row;
  b[j].value = a[i].value;
```

i	a[i].col	start_pos [a[i].col]++	a[i]	b[j]
1	0	1	a[1]	b[1]
2	3	6	a[2]	b[6]
3	5	8	a[3]	b[8]
4	1	3	a[4]	b[3]
5	2	4	a[5]	b[4]
6	3	7	a[6]	b[7]
7	0	2	a[7]	b[2]
8	2	5	a[8]	b[5]

Fast Transpose Algorithm VII

final form

ori	row	col	value	trans	row	col	value	final	row	col	value
a[0]	6	6	8	b[0]	6	6	8	b[0]	6	6	8
a[1]	0	0	15	b[1]	0	0	15	b[1]	0	0	15
a[2]	0	3	22	b[6]	0	3	22	b[2]	0	4	91
a[3]	0	5	-15	b[8]	0	5	-15	b[3]	1	1	11
a[4]	1	1	11	b[3]	1	1	11	b[4]	2	1	3
a[5]	1	2	3	b[4]	1	2	3	b[5]	2	5	28
a[6]	2	3	-6	b[7]	2	3	-6	b[6]	3	0	22
a[7]	4	0	91	b[2]	4	0	91	b[7]	3	2	-6
a[8]	5	2	28	b[5]	5	2	28	b[8]	5	0	-15

Fast Transpose Algorithm VIII

Analysis of `fast_transpose()`

- First two `for` loops compute the values for $rowTerms$
  - comptuting time: $numCols$ and $numTerms$
- the thrid `for` loop carries out the computation of $startingPos$
  - comptuting time: $numCols - 1$
- the last `for` loop places the triples into the transpose matrix
  - comptuting time: $numTerms$
- Complexity of the algorithm : $O(columns + elements)$
  - Worst case : $O(columns \cdot elements)$ when number of elements is of the order $columns \cdot elements$

Practice Problem (5 Min)

Analyze and understand the algorithm of `fast_transpose`

```
void fast_transpose(term a[], term b[]){
  ...
    if(numTerms > 0) {
        for(i = 0; i < numcols; i++)
            row_terms[i] = 0;
        for(i = 1; i <= numTerms; i++)
            row_terms[a[i].col]++;
        starting_pos[0] = 1;
        for(i = 1; i < numcols; i++)
            starting_pos[i] = starting_pos[i-1] + row_terms[i-1];
        for(i = 1; i <= numTerms; i++){
            j = starting_pos[a[i].col]++;
            b[j].row = a[i].col;
            b[j].col = a[i].row;
            b[j].value = a[i].value;
        }
    }
}
```