159.201 Algorithms & Data Structures Tutorial 1 Answers

1. What is the largest 2-dimensional array of integer numbers that you can create using your usual compiler? (to be answered when you get access to a computer)

Use the following C/C++ code:

```
#define SIZE 1000000000
main( ){
     int array[SIZE][SIZE];
}
```

Change SIZE and try to compile. Eventually you will find that over a certain limit the compiler complains:

"error: size of array 'array' is too large"

2. Discuss the conditions in which sparse matrices are better represented by linked-lists. (tip: how many bytes do you need to represent an element using arrays, and using linked-lists?)

Suppose you use the following struture:

```
struct Node {  //declaration
     int row;
     int column;
     int value;
     Node *next;
};
```

We can count the number of bytes in each solution. Each element of LL occupies a lot more space in memory than sizeof(int) in the arrays. In a certain architecture with GCC, we have:

***sizeof**(int) = 4 bytes*
***sizeof**(Node)= 24 bytes*

So, every value of the matrix occupies 24 bytes using LL, instead of the 4 bytes required by the integer. Considering that the number of non-zero elements is Z, and N is the size of the matrix (number of rows=N and columns=N):

$4\ N^2 = 24\ Z$

Therefore, to save space in memory using LL, Z has to be smaller than:

$Z < (N^2 / 6)$

*OBS: different architectures will present different sizeof() results. Also, alignment forces the compiler to use a bit more space than simply add the integers and the pointer sizes. E.g., in the example above sizeof(Node) should be 3*4+8 = 20 instead of 24.*

3. Find out how to represent a matrix and how to add two matrices.

We can represent a matrix using LL by not creating a node when value=0.

$$\begin{vmatrix} 1 & 3 & 4 \\ 2 & 0 & 0 \\ 0 & 0 & 0 \end{vmatrix}$$

We should add all the correspondent values that have the same position (row,column).

E.g.,

$$\begin{vmatrix} 1 & 3 & 4 \\ 2 & 0 & 5 \\ 0 & 0 & 1 \end{vmatrix} + \begin{vmatrix} 1 & 1 & 0 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 2 & 4 & 4 \\ 2 & 1 & 7 \\ 0 & 0 & 1 \end{vmatrix}$$

4. Check how to set up a linked list, insert elements and delete elements. Consider the following questions:

- In what order should matrix values be stored in the linked list?

There is no single answer. You can choose to store the matrix in an inverse order using the AddNode function given in class. In this case the first element read from the file becomes the last element of the LL.
Modifying the AddNode function to add to the tail of the LL makes it easier to read and print the matrix.

- Should you insert values at a particular place in the linked list?
- Or only append values to the front of the list?
- Or only append values to the rear of the list?

Depends on the choices you make, you need to modify the AddNode and Search functions, and create a PrintMatrix function to implement assignment 1.

- Which insertion method is the easiest? Why?

The easiest way to write an add matrix function is to AddNode to the head the way we did in class. However, when reading the two matrices they will be inverted. When you add the nodes to the resulting matrix, the resulting matrix may be in the correct order (or not...) making it easier to print. For example, consider that the following functions are available:

```
void AddNode(Node *& listpointer, int row, int column, int value);//add nodes to the head
int Search(Node * listpointer, int row, int column);//return a value given (row, column)
void PrintMatrix(Node * listpointer);//print values if they exist and print zeros if they don't
```

The simplest implementation for the addition would look like:

```
for (i=0;i<number_rows;i++){
  for (j=0; j<number_columns;j++){
    int result = Search(A, i, j) + Search(B, i, j);
    if(result != 0) AddNode(C, i, j, result);
  }
}
```

However, matrix C would be inverted. The following code snippet shows how to implement the addition of the two matrices (C = A + B), inverting the order of the addition of the nodes so matrix C is the right order:

```
for (i=number_rows-1; i>=0 ;i--){
  for (j=number_columns-1;j>=0;j--){
    int result = Search(A, i, j) + Search(B, i, j);
    if(result != 0) AddNode(C, i, j, result);
  }
}
```

This solution is terribly inefficient though. The Search function keeps traversing the LL repeatedly. Can you

think of a better solution that only traverses each matrix once? Can you think of a function to always store A and B and C in the right order? Adding nodes to the tail instead of the head would solve the problem.

For the printing problem, the simplest (but slow) solution is:

```
void PrintMatrix_Slow(Node *listpointer, int size) {//added size to help
  for(int i=0;i<size;i++){
    for( int j=0;j<size;j++){
      printf("%d ",Search(listpointer,i,j));
    }
    printf("\n");
  }
  printf("\n");
}
```

The previous function needs the Search() function.

If the matrix is sorted in the right order, we can save time traversing the LL only once:

```
void PrintMatrix_Fast(Node *listpointer, int size) {
     int trackrow=0; int trackcol=0;
     Node *current;
     current = listpointer;
     while (true) {
       if (current == NULL) { break; }
       if(trackrow==current->row && trackcol==current->column){
             printf("%d ",current->value);
             current = current->next;
       }
       else {
             printf("0 ");
       }
       trackcol++;
       if (trackcol>(size-1)){
             printf("\n");
             trackcol=0;
             trackrow++;
       }
     }
     while(trackrow < size){//it may still have more rows
       printf("0 ");
       trackcol++;
       if (trackcol>(size-1)){
             printf("\n");
             trackcol=0;
             trackrow++;
       }
     }
     printf("\n");
}
```