COSC2430: Programming and Data Structures
Sparse Matrix Multiplication: Search and Sorting Algorithms

1 Introduction

You will create a C++ program to multiply two sparse matrices as efficiently as possible depending if their entries are sorted or not. You will detect if matrices are sorted or not. You will exploit binary search and efficient sorting algorithms to improve efficiency. Sparse matrices have many zero entries, but such entries are omitted in the input file and corresponding data structures to save space (main memory) and reduce processing time. Your sorting algorithm must count the number of comparisons in order to understand time complexity (Big O). You will have to derive $O()$ and show it bounds the number of comparisons.

Multiplication: Matrix multiplication $C = AB$ is defined as c_{ij} being the dot product between row i from A and column j from B. Matrix multiplication requires that the number of columns of A is equal to the number of rows in B. That is, matrices can have different sizes and can be rectangular.

2 Input

The input are two text files, with one matrix per file. There will be ONE matrix entry per line in the file and each line will have a triplet of numbers i, j, v where i, j indicate the entry and $v \neq 0$ is a real number. Subscripts i, j are positive integers and the value v is a real number (possibly negative). Comments have a # at the beginning of the line. Examples of input and result file are below.

```
#example 1
#Matrix A, no zero entries
#i j value
1 1 10.11

#Matrix B
1 1 2
1 2 3

#Matrix C=A*B
1 1 20.22
1 2 30.33

# example 2
#Matrix A,
#i j value
1 1 2
2 2 3
```

```
#Matrix B
2 2 3
1 1 2

#Matrix C=A*B
1 1 4.00
2 2 9.00


#example 3
#A
3 1 1
3 2 1
3 3 1

#B
1 1 1
2 1 1
3 1 1
3 3 0

#C=A*B,
3 1 3.00
3 3 0.00

#example 4: incompatible
# A
1 1 10
10 10 1.0

#B
100 10 1.00
```

Example of ”bigO.txt” file:

```
algorithm  #entries(n)  countComparisons bigOcountComparisons
nlogn         10             100              120
nlogn        100             200             3000
nsquared      10            1000             1200
nsquared     100            4000             4500
..
```

3 Program and input specification

The main program should be called ”sparse”. The result matrix should be stored on the (sorted) output file (another text file), provided on the command line. Call syntax:

```
sparse "operation=<add|multiply>;storage=<array|linkedlist>;sort=<nosort|nsquared|nlogn>;A=<file>;B=<file>;result=<file>".
```

Notice the brackets $<>$ indicate the parameter value and they are not part of the input string. For this homework ”storage=linkedlist” will be ignored, but it is desirable your program for previous homeworks still works.

Assumptions for multiplication: Array is the only required storage mechanism. you can store the matrix as a single list of non-zero entries in one 1-dimensional array. You will need to sort the array to make multiplication faster.

```
# array is the default storage
sparse "operation=multiply;A=a.txt;B=b.txt;result=C.txt"

# array maybe slow if not sorted
sparse "operation=multiply;A=01.txt;B=02.txt;storage=array;sort=nosort;result=C.txt"

# array fast
sparse "operation=multiply;A=01.txt;B=02.txt;storage=array;sort=nsquared;result=C.txt"

# array faster
sparse "operation=multiply;A=01.txt;B=02.txt;storage=array;sort=nlogn;result=C.txt"
```

4 Requirements

- Phase 1: Arrays: plain multiplication, no sort algorithms; sequential/binary search. Detect if input matrices are sorted.

  Phase 2: Arrays: sorting (one $O(n\log n)$, one $O(n^2)$ algorithm) and always binary search. For sorting algorithms you must create a table counting comparisons and comparing that to your $O()$ function; this table should be written to a text file ”bigO.txt” (with columns algorithm, n, count, bound).

- Operations required (in this HW):

  matrix addition: NO

  matrix multiplication: YES.

- Storage: arrays of matrix entries: array of triples (i, j, v).

  Arrays: lists stored as arrays of non-zero elements. You can store the matrix as a single 1-dimensional array. You cannot store zero entries, except one corner of the matrix.

- Search algorithms: inside your program you ahould detect if a matrix is sorted. You should use sequential search when the input array is not sorted. Otherwise, if the array is sorted you should use binary search. This search can be used to find one matrix entry, or one column, or one row.

- Sorting algorithms: you should program the algorithm in its most common flavor (recursive or non-recursive). You can pick any $O(n\log n)$ algorithm and any $O(n^2)$ algorithm. In your README file indicate WHICH algorithms you picked and why (ease of programming, efficiency, intuitive). You must modify the sorting algorithms to count the number of comparisons and compare such count to its big $O()$ function.

- C++ programming requirements:

Sparse storage: you cannot store zeroes in your matrix data structure, other than the lower right corner a_{mn}. You cannot use simple 2-dimensional arrays to store the matrix (e.g. double A[10][10]). It is preferable your storage takes space $O(q)$, where q is the number of non-zero entries.

Your C++ code must include at least two ".h" files: arraysort.h, arraysearch.h, whose source code will be verified by the TAs. Your main list functions must be in these files. You can have more C++ files for parameters, base classes, text file manipulation, etc.

Subscripts in input and output matrices are based on a standard mathematical definition, indexed from 1. Matrices of size $m \times n$ start on entry [1][1] and go up to [max(i),max(j)] $\forall a_{ij}, 1 \leq i \leq m, 1 \leq j \leq n$. However, in your C++ code you can use arrays indexed from [0], from [1] or indexed by your own functions or pointers making a subscript conversion, but the input/output matrices always have subscripts starting at 1,1.

- Input:

  One file per matrix. In general you should assume input matrix entries are not sorted,l but you can make one pass to detect it.

  Prepare your program for slight changes in format (e.g. an extra carriage return, empty lines, an extra space separating numbers, etc). Your program should not get stuck due to a small format issue.

  Notice there may be 1, 2 or more comment lines at the top of the file and there may be comments elsewhere in the file.

  Numbers: can be positive or negative, where negatives start with - and one digit (we will not provide numbers like -.2, or -0.31416e+10). In general zeroes are omitted. But it is acceptable an input matrix has a few zero entries. Input numbers can have up to 5 decimals. You should write output numbers only with 2 decimals (for testing purposes).

  Your program must reject matrices whose sizes are incompatible for multiplication and produce an empty file as result. Incompatible matrices is the only case where your program should stop. Your program must be prepared to skip lines with bad input (bad subscripts,errors, strange symbols, etc). That is, it should attempt to process the matrices anyway.

  Size: Notice matrices are not necessarily squared ane they can even have only one entry: 1×1. Notice the lower right corner will always be provided in the input matrix and it should also be written in the output matrix: This entry determines matrix size. Always determine matrix theoretical size based on the lower right corner matrix entry.

- Output:

  The output matrix should be written in sorted order by row and column in sparse format (one entry per line) as well. That is, all non-zero entries for row 1, all entries for row 2, and so on. For sorting you can use any algorithm.

  The $O()$ should be written to a text file (call it "big0.txt").

- Testing:

  Your program will be tested for dynamic memory allocation and memory leaks (new, delete operators correctly programmed). In other words, the TAs will verify you do not exploit 2-dimensional arrays.

  Notice the program will be tested with sparse matrices whose theoretical size goes up to 1000000×1000000 (i.e. they will have large subscripts in some entries). In other words, it will be impossible to use C++ 2-dimensional arrays. Additional clarifications: matrix subscripts can be large, matrix files may have incorrect entries (skip them), matrix files may be empty or non-existent (report error).

- Grades:

  Phase 1 grade: you will receive a final grade for Phase 1 and a preliminary grade for Phase 2. It is highly recommended you make an effort to attempt Phase 2 on the 1st deadline so that you get feedback and a better grade on Phase 2.

  Phase 2 grade: you can fix errors from Phase 1 and make Phase 1 and Phase 2 requirements fully functional. Phase 2 is an opportunity for resubmission of a fixed program. Resubmissions or late submissions will not be allowed (30% penalty for each day).