CS4115 Week09 Lab Exercise

Lab Objective: Last week, after much huffing and puffing, we finally got to the stage of being able to rewrite a matrix with possibly many zeros into a compressed form by keeping only the non-zero elements. The car stickers tell us "A Dog isn't for Christmas: it's for Life" and the same applies here: a compressed matrix isn't just for reading/writing but for algorithm use, too. So, the final task of last week was to use linked lists to keep track of just the good stuff in a matrix under the assumption that the data would be used later by the program that read it.

Probably the most important operations that one needs to do with matrices are addition, subtraction and multiplication. As motivation for why sparse matrices need to be represented differently we have looked at street networks and later we will see how finding *paths* in a street network is closely related to matrix multiplication. So this – matrix multiplication – is the matrix operation we will focus on. This week we will lay the ground work for matrix multiplication by considering the problem of transposing a matrix.

Here's the lab summary:

❶ Using the C++ machinery introduced in Lab07 read a matrix from standard input, saving it internally in a linked list representation; put all of your work in a single file called `mtrans.cc`

❷ Build the *transpose* of the matrix you have just read in

❸ Write this to standard output

❹ Submit your program, called `mtrans`, for marking using `handin`

In Detail

❶ The focus up until now has been on `matz` a program to compress a matrix into a sparse representation for saving to a file. The input to our programs from now on will assume this format. So your first task this week is to read from standard input a stream of data given in the `formatz` representation, as described in Lab08 (see graphic at bottom of page 3).

Each row will be a linked list of "non zero" elements just like last week. And you will need to maintain a second data structure to keep track of all those linked lists.

So this first task of reading the matrix and setting up your data structure will be very similar to your first task of last week's lab. The input to this week's lab (and future labs) will be the output from last week's lab; that is, a file where each row of input will comprise

zero or more *pairs* of the form `(col, val)` that will make up a non-zero instance on the linked list for that row. (I called my non-zero class `nz`.) You should read the file using a `while(getline())` loop to iterate over the rows of input. Note that a line may be empty indicating that this row of the matrix is a row of all zeros.

Within this loop and having read a line of the file you could, as in previous labs, read in the pairs of data values with something like

```
while (lstream>> col>> val) {
  nz next(col, val);
  row.push_back(next);
```

creating a new non-zero instance using those two pieces of data, to be followed by a call of the list insertion method `push_back()` to append the element to that row.

However, the ambitious amongst you will want to adopt a more object-centric approach and do something like

```
nz next;
while(lstream>> next) {
  row.push_back(next);
```

The subtle but important difference here is that we read one *instance* after the other and have hidden the details of constructing the instance called `next`. You still have to account for the details you have hidden from the reader and this will require you to write an input stream extraction member function of your non-zero class. This has signature

```
istringstream& operator>>(istringstream& is, nz& inst);
```

Almost all of what you will need to implement this can be found at this link.

And now that you have learnt how to overload the input stream operator why not go the whole way and overload the output stream operator, `operator<<`, as well.[a]

[a] A well structured program should "read" easily. By that I mean that as you read down through a function, each line of code should be a single "high level" command. Within reason the details of the command should be relegated to further function calls. And this is what we are doing here when we overload the stream extraction (input) and stream insertion (output) operators.

You should now have a program that reads a sparse file and saves it internally in sparse format. So the memory requirements of our program are kept as small as we possibly can. On the other hand, imagine if you were given a 10,000 × 10,000 integer matrix where you knew only 10 entries of the entire matrix was non-zero. In `formatz` notation we would only require approx. 10 units of space whereas the way we did it in Week03 would have required space for 100 million integers.

❷ The processing that is required this week is to output the matrix in transposed form. The transpose of an m-row, n-column matrix is an n-row, m-column matrix. Reading across a row of a matrix should be the same as reading down a *column* of its transpose. More precisely, if the array A is made up of elements $(a_{i,j})$

$$A = (a_{i,j})$$

then, its transpose is

$$B = A^T = (a_{j,i})$$

If we had saved our matrix in the fat, turgid, bloated way of an array of one-dimensional arrays (refer back to Lab03 and `matmult`) then it would be quite easy to write out the transpose the matrix, `B`.

```
for (int r = 0; r < ar; r++) {
  for (int c = 0; c < bc; c++)
    cout<< B[r][c]<< " ";
  cout<< endl;
}
```

Simply change the order of the array subscripts so that it is

```
    cout<< B[c][r]<< " ";
```

instead.

With the sparse representation life is more complicated though. What you now have to do is create a data structure, `B`, of the same format as you used when reading in the matrix, `A`, earlier. `B` will be the transpose of `A`.

A suggested algorithm: as you scan across a row of A the column of a non-zero value must be the row of B that it is put in. Put another way, a non-zero element $a_{i,j}$ of `A` is on row i and column j; this element will show up in `B` as $b_{j,i}$.

As you accumulate the values in a row of the transpose make sure that you insert them into the list in the correct order. Is there a time penalty to be paid for doing this?

You should think carefully about your representation of a sparse matrix because how you represent it will have a major impact on the running time of your transposition. Last week I gave two possible implementations. Do you need something more sophisticated or will one of these be enough?
I will mark your program according to the efficiency of your chosen data structure and algorithm.

Thinking time

Another thing to think about is how to determine the number of rows in the transpose. If the input matrix was originally square (both dimensions n) then it is fairly obvious how many rows will be in the transpose but what if it was not?

❸ You should now write `B` the transpose of `A` to standard output. I have put in this week's lab directory an executable that, as far as I can tell, does this job. Please use it as a reference point when testing your program's output.

As a further aid in testing your program I have also placed there a perl script that will generate sparse matrices, `gen-matrix.pl`. The program takes as arguments the number of rows you wish to generate, the average density of non-zeros and a flag that indicates that the matrix should be symmetric. So

```
perl ~cs4115/labs/week09/gen-matrix.pl -n 500 -d 0.01 -s
```

should generate a symmetric random matrix of size 500×500, with approximately $500 \times 500 \times 0.01 = 2500$ non-zero entries.

Note that since the transpose of a symmetric matrix is itself then "filtering" a symmetric matrix through your `mtrans` program should result in identical output. That is

```
perl ~cs4115/labs/week09/gen-matrix.pl -n 500 -d 0.01 -s > m500.in
../week08/matz < m500.in > m500.out
mtrans < m500.out > m500.trans
diff m500.out m500.trans
```

should result in silence from `diff`. The `diff` program only squawks when it finds differences between the two files.

Next week we will look at multiplying `A` and `B` and we will see why this is such an important problem.

❹

Note that this lab will be inspected by me and marks will be allocated for a) the functional decomposition of your program and, b) your choice of data structure to support the mtrans algorithm.

The command to submit your program is

```
handin -m cs4115 -p w09
```

Labs that are to be handed in are open for handin at 09.00 on Friday of the week it is *assigned* and, in order to get full marks, are due by 15.00 on Friday of the *following* week; a lateness penalty applies to submissions made until 18.00, Monday after that. This gives you one week to work on each lab that is to be assessed and handin without penalty. The lab sheet will be available for reading earlier in the week so that you can read it beforehand and come to the lab with questions.

Subject to last-minute changes, the planned schedule of lab assignments due for handing in are:

Lab. Week	Assessed	DueDate
Week02	✗	
Week03	✓	Fri, Week04
Week04	✗	
Week05	✗	
Week06	✓	Fri, Week07
Week07	✓	Fri, Week08
Week08	✗	
Week09	✓	Fri, Week10
Week10	✗	
Week11	✓	Fri, Week12