Ol' China taught us how to fish and fly
Out here, we simply sit. And multiply.

a cormorant who multiplies

To code less and to multiply

prcviously by Michacl Loccff

Having large matrices is all well and good. But we also want to be able to multiply them (if they are compatible).

In this quest, You get to implement a class that houses Matrix utilities (though, of course, you will only implement multiplication). Through friendship to the Matrix and Sparse Matrix classes from your previous quest, you can *carefully* access the guts of those classes and implement efficient Matrix (or Sparse_Matrix) functions.

I hope this relatively easier (and shorter) quest gives you some breathing room to get ready for the next one, which is loaded with rewards.

About Matrix Multiplication

Look up Matrix multiplication in a math text or online. You need to understand it clearly in order to complete this lab, especially the part to do with multiplication of sparse matrices.

Figure 1 shows how Matrix multiplication works: Consider three matrices A x B = C:

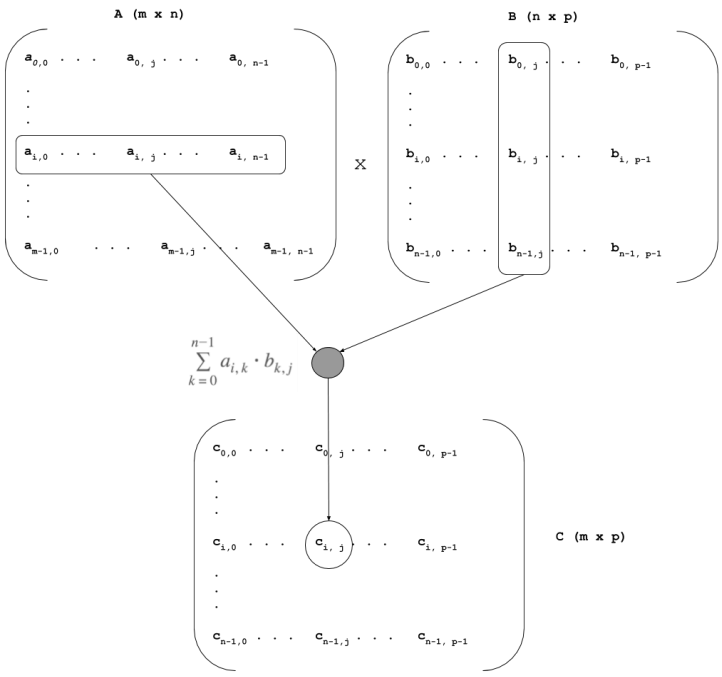

Figure 1. Matrix Multiplication

So...

1. This quest only makes sense for numerical `Matrices` and `Sparse_Matrices`.
2. These must be compatible for multiplication. The number of columns in the first matrix must equal the number of rows in the second (no constraint on the rows of the 1st and cols of the 2nd). Yes, that means that matrix multiplication is not commutative.
3. Each element of a product matrix, C, is obtained by summing the products of corresponding elements from A and B. Specifically, `c[i,j]` is obtained by summing the corresponding elements from the dot-product of the `i`'th row of A and the `j`'th column of B. If A and B and compatible for multiplication, then the number of elements in the `i`'th row of A must equal the number of elements in the `j`'th column of B. Thus you are guaranteed to have a correspondence for each element of either vector. This sum, the dot product between two vectors, is written as:

$$\bar{a} \cdot \bar{b} = \sum_k a_k b_k \quad - \textit{ which is just shorthand for } - \quad a_0 b_0 + a_1 b_1 + \cdots + a_{n-1} b_{n-1}$$

   Not so clear? Please discuss on our sub. Most likely, you're not the only one.

It's important to completely understand how this works because when you get around to implementing multiplication for sparse matrices, you can't afford to do unnecessary lookups. In a regular matrix, it doesn't matter how many times you refer to a certain cell because access happens in constant time.

In a sparse matrix, element access time is proportional to the number of non-default elements contained in the row in which the element exists (according to the specs of the Stilt lab). Your utility functions must be aware of the underlying implementations of the data structures they are dealing with and handle them appropriately, even though the external presentations of both structures (`Matrix` and `Sparse_Matrix`) are the same and conceptually the same operations apply to both.

Implementation specifics

Implement this quest in three files (suggested number):

1. `Matrix.h` (from the Stilt quest - needs editing)
2. `Sparse_Matrix.h` (from the Stilt quest - needs editing)
3. `Matrix_Algorithms.h` (new file)

Set up your project by copying over the first two header files from your Stilt quest. Create a new file called `Matrix_Algorithms.h`

In this file (I suggest) define a new class called `Mx (See Figure 2)`. It will be home to all of your matrix[1] manipulation methods. In this quest you will only need to implement the matrix multiplication function. But you can imagine that this class may eventually be home to many other matrix functions.

Note that `Mx` is itself not a template class although it provides template utility functions over matrices.

In order for `Mx` to be able to directly access internal data elements of the `Matrix` and `Sparse_Matrix` classes, it must be their friend. So insert the required friendship statements into the original classes.

In addition, here are assumptions I made in my reference implementation. Your assumptions had better match if you want to max out on your rewards:

- When dealing with floating point values, it's usually a bad idea to compare for exact equalities. So we define *ranges of values* to look for. The default value in your sparse matrices is 0.0. But to avoid the risky exact comparison, define a `static constexpr` called `Sparse_Matrix<T>::FLOOR` whose value is 10^{-10} (`1e-10`).
- Then define a utility method:

```
bool is_default(const double &val);
```

[1] Some of what is rewarding for a regular matrix may earn you even more rewards with sparse matrices. You'll never know until you try.

It should return `true` if the absolute value of the difference between `val` and the instance's `_default_val` member is smaller than your `FLOOR`, and `false` otherwise.

```
s Mx {
ic:
template <typename T> static bool can_multiply(const Matrix<T> &a, const Matrix<T> &b) {
    // TODO
    return true;
}

template <typename T> static bool multiply(const Matrix<T> &a, const Matrix<T> &b, Matrix<T> &res) {
    // TODO
    return true;
}

// Sparse_Matrix utils ------------------------------------------------

template<typename T> static bool can_multiply(const Sparse_Matrix<T> &a, const Sparse_Matrix<T> &b)
    // TODO
    return true;
}

template<typename T> static bool add_to_cell(Sparse_Matrix<T> &spmat, size_t r, size_t c, const T &v
    // TODO
    return true;
}

template <typename T> static bool multiply(const Sparse_Matrix<T> &a, const Sparse_Matrix<T> &b, Spa
    // TODO
    return true;
```

Figure 2. A View of `Mx`

All you gotta do in this quest is to fill in the // TODO portions of the five inline methods.

Here is a little more detail that is hopefully helpful:

- `can_multiply(a, b)` returns `true` if matrices `a` and `b` are compatible for multiplication in the given order (i.e. `a` x `b`). Be sure to check for corner cases.
- `add_to_cell(r, c, val)` will add `val` to the sparse matrix cell at `(r,c)`. You can imagine its functionality to be similar to but yet subtly different in important ways from the `Sparse_Matrix`'s `set()` method from the Stilt quest. Remember that if the addition of `val` to the existing value makes it equal to the default value (as determined by `is_default()`) then the node should be removed from the `Sparse_Matrix`. The reference sparse matrix preserves its sparseness through operations like this, and yours needs to match in order to get past a miniquest. It should return false if the supplied coordinates are invalid or if the spine of the matrix is not long enough.

When you have passed all the other miniquests, the site will tell you how long my multiplication code took on 2 relatively large sparse matrices. It will also tell you how long yours took.[2]

Feel free to discuss these times in our sub. You should be able to run as fast as the reference code, or better. Discuss and vote-up optimizations and unclever hacks.[3]

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to https://quests.nonlinearmedia.org
2. Enter the secret password for this quest in the box.
3. Drag and drop your source files[4] into the button and press it.

Wait for me to complete my tests and report back (usually a minute or less).

Happy Questing,

&

[2] Hmm… if I multiply two sparse matrices with density ρ each, what can I say about the density of the product?
[3] *unclever hacks* are hacks you can be sure to understand when you're much much older and not so much much cleverer.
[4] These would contain the definitions of the three necessary classes.