I bet that if I got meself some 6 or 7 stilts
Altho' I be tall alreddy, I be taller even stills.

i scares em with my stilts and pecks em with my bilts

Matrix and Sparse Matrix

previously by Michael Loceff

Sometimes we want to store an enormous matrix. So many rows and cols that maybe even all the available bits on the planet wouldn't suffice to hold em.

But it may be that most of the elements happen to be the exact same value, say zero.

Then it should technically be possible to store any (possibly infinite)[1] amount of such data on a significantly small finite device as long as the *effective data density* (EDD) is below some threshold. By EDD, I mean the ratio of the number of non-default values to the total number of values.

You can offer the user a partial illusion of the existence of such a data set by making your opaque getters generate the bulk of their values algorithmically. This is a cool way in which you let the software take on more of the hardware's functionality. (Come back when you've found out where this is happening in this quest and post your thoughts on our sub)

Here, you'll get to implement one such data structure as a template class - A Sparse Matrix. You can think of it as a two dimensional array in which some relatively small number of cells are expected to contain non-default values.

Clearly, you can't afford to have a vector of vectors. That would defeat the purpose, right? So you'll store your data in a vector of lists, which is a second best from a storage perspective. (Why second?)

Read on to learn more.

Implementation Specifics

First things first. Starting with this quest, you will stop seeing explicitly spelled out miniquests. The spec will give a high-level description of what is required and discuss implementation specifics. You'll find that rewards automatically become yours as you do more things right.

How many more?

Well. That you gotta find out yerself.

[1] Can you think of a quest you did (may not be in 2C) in which you had to store *infinite data* using clever abstractions?

Before you get to implementing the sparse matrix, you must first implement a regular matrix, also a template class: `Matrix<T>`. This class will allow you to extract relatively small rectangular regions, called *slices*, from a large Sparse Matrix (to come).

Refer to Figure 1 for a hazy picture of the Matrix class.

```
template <typename T>
class Matrix {
protected:
    vector<vector<T> > _rows;

public:
    Matrix(size_t nr = 0, size_t nc = 0);

    size_t get_num_rows() const { return _rows.size(); };
    size_t get_num_cols() const { return (_rows.size()>0? _rows[0].size() : 0); };
    void clear() { _rows.clear(); }

    void resize(size_t nr, size_t nc);
    T &at(size_t r, size_t c); // throws OOB
    string to_string() const;

    class OOB_exception : public exception {
        public: string what() { return "Out of bounds access"; }
    };

    // Friends ----------------------------------------------------------------

    friend ostream &operator<<(ostream& os, const Matrix<T> &mat) {
        return os << mat.to_string();
    }

    friend bool operator==(const Matrix<T> &m1, const Matrix<T> &m2) {
        if (m1.get_num_rows() != m2.get_num_rows()) return false;
        if (m1.get_num_cols() != m2.get_num_cols()) return false;

        return t;
    }

    friend bool operator!=(const Matrix<T> &m1, const Matrix<T> &m2) {
        return !(m1 == m2);
    }

    friend class Tests; // Don't remove this line
};
```

Figure 1. A fuzzy photo of the Matrix<T> class

Most of what you will find rewarding with the Matrix class is shown in Figure 1. But, alas! Some of it is missing. Just fill in the missing implementations.

Imagine that I would instantiate a Matrix of doubles or strings and that I'd invoke each of the public methods you see in the figure, and the friend methods of Matrix.

As you read ahead, remember that you may still not understand everything even after reading this spec. If so, that's great because you can start a discussion about it on our subreddit and improve its next version.

About the Matrix class

There is nothing sparse about your non-sparse Matrix. It is a vector full of vectors. The inner vectors are your rows and your outer vector is your vector of rows. Each row has as many cells as the number of columns of this matrix. This vector should be called `_rows`, as shown in the figure. Don't change it.

Here are some rewarding facts about the `Matrix`:

The Matrix constructor

The constructor takes two `size_t` params: `nr,` which gives the number of rows, and `nc`, which gives the number of columns. You must correctly size the member, `_rows`, to hold as many elements and sub-elements as needed. Each element will be automatically assigned the default value of the type `T` by the `vector` constructor.

Done correctly, you should *not* have to maintain two separate members tracking the height and width of your matrix.

My personal preference is to try and keep constructor code as lean as possible, I think I discussed this somewhere on our 2B sub. However, this time, I let the interests of better code factoring override this desire.

Since I have a `resize()` implemented and verified *beyond reasonable doubt* AND it is lexically positioned adjacent to my constructor, I gave myself the luxury of invoking it from within.

You can choose to do it differently, of course. See how it goes. One way may be rewarding and another not. I have no way of telling what works better for you.

Equality

Yeah. It's important. If equal mats ain't equal then yer quest ain't got no sequel.

At

This one's a biggie. The thing about `Matrix::at()` is that it is the ONLY interface `Matrix` presents to get at its innards. You will make it return a reference to the element and completely eschew the unchecked `get()` method, or the bracket operators.[2]

[2] What are the implications if `at()` returned a copy of the element rather than a reference to it?

In terms of naming, I picked `at()` over `get()` because our functionality is more similar to that of `vector::at()`, which returns a reference, rather than to a traditional const getter.

`Matrix::at()` should be a checked method that throws an `OOB_exception` if either `r` or `c` is an invalid index.

To string

Simply print the line "`#  Matrix`" by itself, followed by data lines, at one row per line. Each element of each row should be in a field 6 characters wide[3] with one space before every field except the first. There is one newline after each line (including the last). Figure 2 shows sample output from my `Matrix<int>::to_string()`

```
# Matrix
437605 642314 836725 159909 506331 815035 816302
409351 452462 833879 296251  89079 975107 252029
241127 392454  28736  10327 994523 901792 811147
597315  24573 826460 198260 683811 440188 348233
584421 302083 764371 720702 733245 806895 514515
351438 150475  76576 105777 542228 875713 405431
198472  37512 408578 413258 251483 111951 367854
344877 860043    685 290380 308258 762205 836872
709951 691467 841524  48538 487976 931525 460761
```

Figure 2. Sample `Matrix::to_string()` output

[3] Use `std::setw(6)`

The Sparse Matrix

Like it said in the introduction, you will implement your sparse matrix as a vector of lists.[4]

A user should be able to instantiate a concrete version of your template sparse matrix class like this (example uses `double`):

```
Sparse_Matrix<double> my_sparse_matrix;
```

The constructor of the sparse matrix should take 3 parameters:

```
Sparse_Matrix(size_t nr, size_t nc, const T &default_val)
```

The number of rows is given by `nr` and the number of columns by `nc`. Indexing should begin at 0, and so the actual rows and columns will span indices `0` to `nr-1` and `0` to `nc-1`.

Why do we even need `nr` and `nc`? After all, isn't our sparse matrix supposed to accommodate arbitrary numbers of elements (limited only by available memory)?

Good questions. Rewards may await thoughtful discussion in our sub.

But what is this other thing called `default_val`?

This I can help with. It is a value stored in the instance (not the class) and returned to the user whenever they ask for the contents of a cell *not physically stored* in the representation. Since the sparse matrix may be as large (e.g. 1 billion cells tall and wide) it's not feasible to store every single value explicitly. Rather, it will only store the significantly fewer non-default values, and return `default_val` whenever a request for an unstored value arrives.[5]

I managed to buy a photo off a character on the Milk[6] road. But the seller was shady and the pic was even worse. Still, I managed to salvage nuff of it, though some parts are still obscure. FWIW, it's in Figure 3. See if it helps.

---

[4] What might be some interesting similarities and differences between this and the general tree from CS2B?

[5] Hey, quiet! Can anyone hear a *mynah* in here somewhere?

[6] I'm told it's one of dem places where you could buy a quest solution for something that rhymes with bitcoin.

```cpp
template <typename T>
class Sparse_Matrix {
private:
    static double constexpr FLOOR = 1e-10;

    struct Node { // private inner
        size_t _col;
        T _val;

        Node(size_t c, const T &v) : _col(c), _val(v) {}
        size_t get_col() const { return _col; }
        const T get_val() const { return _val; }
        void set_val(const T &val) { _val = val; }
        virtual const T &operator=(const T& val) { return _val = val; }

        friend ostream& operator<<(ostream& os, const Node &node) {
            return os << "{ C: " << node.col << ", V: " << node.value << " }";
        }
    };

    vector<list<Node> > _rows;
    size_t _num_rows, _num_cols;
    T _default_val;

public:
    Sparse_Matrix(size_t nr = 0, size_t nc = 0, const T &default_val = T()) :
      _num_rows(nr), _num_cols(nc), _default_val(default_val) {

    }
    size_t get_num_rows() const { return _num_rows; };
    size_t get_num_cols() const { return _num_cols; };

    bool is_valid(size_t r, size_t c) const;
    void clear();
    const T get(size_t r, size_t c) const;
    bool set(size_t row, size_t col, const T &val);
    Matrix<T> get_slice(size_t r1, size_t c1, size_t r2, size_t c2) const; // rect slice

    friend class Tests; // don't remove
```

Figure 3. The partially obscure `Sparse_Matrix<T>` class

As you can see, your sparse matrix maintains a *spine*, which is a vector of lists of nodes. Each node contains two things: a value, and the column number the value occupies.

Given a row, `r`, and column, `c`, you can now index into the corresponding list, `_rows[r]`, and scanning this list linearly, find whether the column of interest resides in it or not. If it does, you would have located non-default values for the contents of the requested cell. If it doesn't then you can assume that the cell contains `default_val` in virtual space.

I hope that makes sense. See Figure 4 for a pictorial representation of this idea.

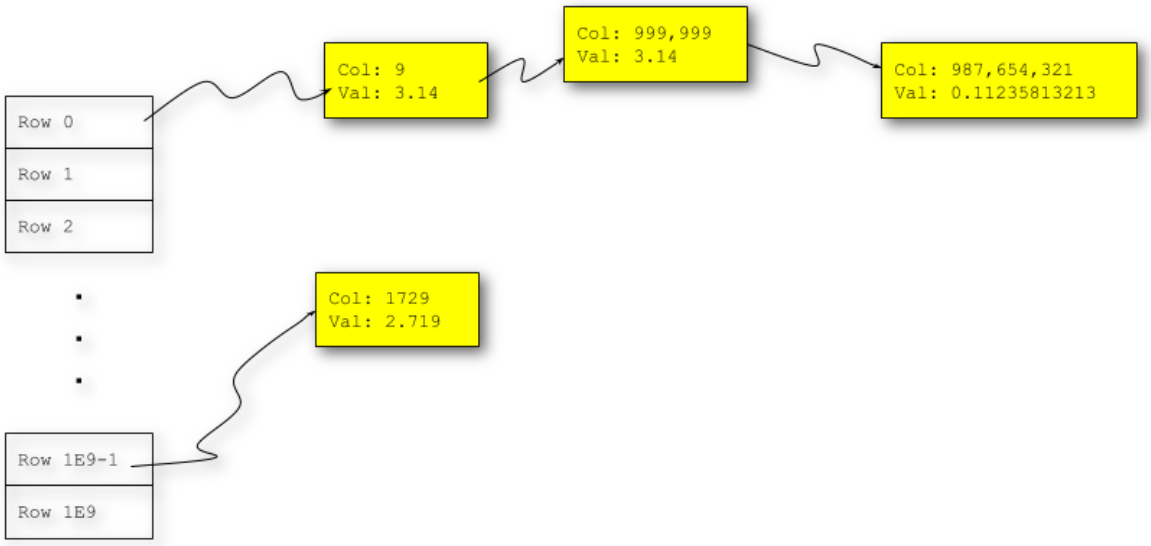

Figure 4: The Sparse Matrix as a vector of linked lists of Nodes containing cols and vals. Any value that cannot be found in a yellow cell is assumed to be `default_val`. This sparse matrix has a ++billion rows and a billion columns. 4 of these billion billion cells have non-default values

Important subtlety

Since your rows are linked lists and not vectors, you may as well exploit their O(1) in-place insertion complexities. Your linked lists of Nodes must be kept in ascending[7] order by column number. Although not terribly significant in our use case, this does buy you about half the time you might have spent searching on average. Why?

Here are further rewarding things to know about the `Sparse Matrix`:

- I found it useful to have a private `is_valid(r,c)` helper to make sure I didn't accidentally access something that didn't belong to me.
- `clear()` should keep the spine of the sparse mat, but clear out its elements.
- `get(r,c)` should return the stored value at the requested location or the default value if a value couldn't be found. It doesn't have to mess around with exceptions.[8] If the requested row and column are either out of bounds or invalid, it may return the default value.
- `set(r,c,val)` is more subtle:
    - Make sure the Nodes in the list at `_rows[r]` preserve their ordering property (ascending order by column id). To achieve this you simply have to do this:
        - Scan the list from head to tail looking for the target column, `c`. There are only 3 cases as you test each node:

[7] Do I need to say non descending instead? Why? Or why not?

[8] Riley Short (Spring, 2022) asked why we don't throw an OOB in this case because a Sparse_Matrix, though large, is not infinite and has real boundaries that can be breached. Can you think of good reasons to do it one way or the other? Link to the discussion on our sub.

1. the current column `<` `c`: move on to the next node
2. The current column `>` `c`: remember this location and break out of the loop.
3. (else) The current column `=` `c`: if `val` is the default value, delete this node and you're done. Else, whatever the current value of this node, simply reset it to `val` and you're done.

- This is when you have broken out of the loop, but are not yet done (when does this happen?) If you're here, it means that you're exactly at the node behind which a new node with column `c` should be added, but of course, only if `val` is not the default value. Go figure.

- `get_slice(r1,c1,r2,c2)` is straightforward:
  - Rearrange `r1,c1` and `r2,c2` so they form a rectangle with `r1,c1` at the top left and `r2,c2` at the bottom right.[9]
  - Extract values from the sparse matrix at every location from within this rectangle (includes the end points) and stuff it into corresponding locations in a regular matrix whose size is just enough to hold the extracted slice.
  - Use `spmat.get(r,c)` to extract values. Use `mat.at(r,c) = val` to insert them.
  - I guess this is your ultimate test of comfort with corners.

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 `Matrix.h` and `Sparse_Matrix.h` files into the button and press it.

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

Happy Questing,

&

[9] Hmm… Did Meg say there was an easy way to do this?