Group 4: Isaac Supeene (isupeene) & Braeden Soetaert (bsoetaer)

####################
Design Overview
####################

We are using Ruby version 1.9.3. Our design is to have a matrix builder rather than a matrix factory that builds
up matrices. The matrix factory builder will be able to build different types of matrices that have different
implementations but all the matrices hold the same MatrixContract. The matrices we are going to implement have
the same functional interface as Ruby's Matrix library and provide the same functionality but for sparse and
tridiagonal matrices. As with the Ruby Matrix class, our matrices themselves are immutable but the matrices
can be built up through the MatrixBuilder by adding elements to a Matrix of a fixed dimension and then when
a matrix is requested from the builder it spits out an immutable matrix.

We also have a VectorContract which will be the contract for vectors which are basically matrices with one size
dimension being equal to 1. These vectors areused as return values from some matrix functions as well as they
can be used as arguments to some matrix functions. The main reason for the existence of these vectors is to provide
an interface that is common to Ruby's Matrix library so that transitioning over to our library from it is a simple
task.

Our contracts wrap the matrix methods of a class when the contract is included into the class. The contracts all
do this through methods that are present in the Contract module. The Contract module does this by redefining the
class methods with the contracts added in to surround the original method. All contract modules extend the Contract
module and define the contracts for a given type.

##############################
Answers to design questions
##############################

* What is a sparse matrix and what features should it possess?

Matrix where most elements are 0. Compact representation, fast operations.
Our implementation should focus on making operations with truly huge sparse matrices possible in Ruby.
Extreme optimization should not be a goal, as Ruby is not suited for it.

Instead, we should make it feasible for our users to prototype matrix code with real datasets in Ruby, so that they can go off and implement it in a language more suited for the numerical computations.

* What sources of information should you consult to derive features? Do analogies exist? If so with what?

Information from other implementations of sparse matrix packages will be useful.
A sparse matrix is also very much like a regular old matrix, in terms of the contract it will fulfill.

Many users may already be using the built-in Matrix class in the Ruby standard library,
and finding that it's not fast enough for their needs.
If they can simply swap in our library and get the increase in performance we need, that will be a success.

* Who is likely to be the user of a sparse matrix package? What features are they likely to demand?

Engineers, primarily. Also professors and financial analysts.
Useful features will include:

- Basic matrix operations (addition, scalar multiplication, transposistion, etc.)
- Matrix Multiplication
- Inversion
- Linear Solvers and Preconditioners (e.g. LU Decomposition)
- Eigenvalues and Eigenvectors

* What is a tri-diagonal matrix?

A matrix where, for all non-0 entries A(i, j), $|i - j| <= 1$

* What is the relationship between a tri-diagonal matrix and a generic sparse matrix?

Generally speaking, a tri-diagonal matrix is a sparse matrix,
but a generic sparse matrix is not necessarily a tridiagonal matrix.

From the perspective of contracts though, if the matrices are mutable,
then a tridiagonal matrix can't be treated as a generic sparse matrix,
because elements off of the tri-diagonal of a tri-diagonal matrix can't be set to non-zero values.

In other words, the contract of a mutable tri-diagonal matrix is not compatible with that of a mutable generic sparse matrix but their immutable contracts are.

* Are tri-diagonal matrices important? And should they impact your design? If so, how?

They are important, because some functions can be implemented more efficiently on tri-diagonal matrices.

We should try to utilize a more efficient representation of the matrix wherever possible,
perhaps by using a tri-diagonal representation when the matrix is tridiagonal, and another representation otherwise.

* What is a good data representation for a sparse matrix?

Coordinate, Compressed Row, and Compressed Column are good representations for a sparse matrix,
but are pretty bad when it comes to adding new elements.
A hash could be used for arbitrary additions to the matrix,
then the representation could be switched to one of the aforementioned when the matrix is frozen.

Tridiagonal matrices can be implemented as three arrays representing the tri-diagonal elements.

* Assume that you have a customer for your sparse matrix package. The customer states that their primary requirements as:
for an N x N matrix with m non-zero entries. Storage should be ~O(km), where k << N. Adding the m+1 value into the matrix
should have an execution time of ~O(p) where the execution time of all method calls in standard Ruby container classes is
considered to have a unit value and p << m ideally p = 1. In this scenario, what is a good data representation for a sparse matrix?

A hash could be a good representation in this case.
Alternatively, we could use coordinate format, and use a binary search to decide where to put the new element.

Aside from taking less space, there's no real benefit to using coordinate format though –
it would be better to use the hash for insertions,
then change to compressed row or compressed column when the matrix is frozen.

* Design Patterns are common tricks which normally enshrine good practice. Explain the design patterns: Delegate and Abstract Factory

Delegate: Where an object, when asked to perform a task, delegates the task to another object.

Abstract Factory: Separates the logic of creation from the logic of usage.
A factory will be acquired from the abstract factory (which can be implemented as an entire class, or just a method),
and provided to the code using the class, which can call the factory to get an instance.

* Explain how you would approach implementing these two patterns in Ruby

The delegate pattern can be implemented by holding the object to be delegated to in an instance variable.
The few methods the delegator needs to actively do something with can be implemented in the

delegator class.
Otherwise, the delegator can implement method_missing and simply forward any other messages to the delegate class.

The abstract factory pattern can be implemented as a module, say “MatrixFactory”, with a method “GetMatrixFactory”.
That method can return a callable object which will return matrices,
or perhaps an object with a number of different methods for returning matrices.

the type of factory returned by GetMatrixFactory can depend on the arguments provided to it,
but the exact class provided need not be known by the caller.
Likewise, the user of the resulting factory need not know the runtime type of the matrix he is using.
Regardless of what is returned by the abstract factory method,
it should return something which satisfies a “MatrixFactoryContract”,
and any matrix provided by the factory should satisfy the “MatrixContract”

* Are these patterns applicable to this problem? Explain your answer! (HINT: The answer is yes)

Yes.

The abstract factory pattern can be used to swap different sparse matrix classes in and out.
For example, one might want a matrix factory that creates matrices in coordinate format to save memory
while the matrix is being constructed, or a matrix factory that only returns instances of the built-in Matrix class.

The delegate pattern can be used to swap out different matrix implementations on the fly.
A top-level matrix class, which is accessed directly by the user,
might delegate to a hash-based implementation while the matrix is being constructed,
then switch to another matrix representation when the construction is complete –
either a compressed row / column format, or a tri-diagonal format if the matrix is tri-diagonal.

Alternatively, if we want to add extra functionality to our library, but still keep a complete representation,
we could use the delegate pattern (or the decorator pattern) to add functionality on top of the existing matrix class.

Potentially a better solution than using Delegate and Abstract Factory in such a way, though,
would be to use the builder pattern.
A typical use case for a sparse matrix might involve creating an empty matrix of known size,
then reading in the elements from a file in coordinate form and adding them one at a time until the matrix is complete,
at which point, no more changes will be made to the matrix.

This can be accomplished by having a mutable MatrixBuilder, which will eventually be converted into an immutable matrix.
Using the Builder pattern and having immutable matrices also solves the problem of whether a tri-diagonal matrix is a sparse

matrix or not. If the matrices are immutable, a tri-diagonal matrix is unequivocally a sparse matrix.

* What implementation approach are you using (reuse class, modify class, inherit from class, compose with class, build new standalone class);
justify your selection.

Obviously reusing the built-in Matrix class won't do, as it has no provisions for sparsity.
Modifying the built-in Matrix class would be more trouble than it's worth,
and composing our class from it would bring along too much implementation baggage
(i.e. the instance variables that represent the complete matrix).

Inhereting from Matrix would be a possibility, if we override the Matrix's initialize method and initialize
its state with sparse vectors that behave just like arrays -
however, that's a very awkward implementation that relies heavily on understanding the internals of the Matrix class.

However, we can compose with Hash, for some implementations, and with Array for others.
For a complete matrix implementation, if we choose to have one as part of our library,
composing the built-in matrix class seems most suitable.

* Is iteration a good technique for sparse matrix manipulation? Is “custom” iteration required for this problem?

There aren't too many operations on matrices that involve sweeping through every single element of the matrix in order.
However, for cases where that's useful, a useful addition would be only to iterate over the non-zero elements
(similar to the way the built-in matrix class's 'each' and 'each_with_index' methods offer ways to specify which elements will be iterated over).

Iterating the rows or columns could also be useful, or iterating the elements row-wise or column-wise.

* What exceptions can occur during the processing of sparse matrices? And how should the system handle them?

Errors if the matrix operation only works on square matrices and the matrix isn’t square.
Error if a matrix is singular so it is not invertible and an inverse is requested.
Mainly the attempt of a matrix operation that only works on certain types of
matrices being used on a matrix it does not work for.

But we avoid all these problems because of our contracts!

The only real exceptions that can occur are when the system runs out of memory,
or our process is killed, and there's nothing that our puny process can do about that.
So we use contracts to prevent vexing exceptions before they occur, and pay no mind to fatal

exceptions.

* What information does the system require to create a sparse matrix object? Remember you are building for a set of unknown customers – what will they want?

For a mutable matrix, we need to know the dimensions of the matrix to create it.
For an immutable matrix, we also need to know the values of all the elements.
taking a bunch of arrays is a little impractical for a sparse matrix,
because that requires the whole thing to be in memory.
Creating a zero matrix and then inserting the elements seems most practical,
then freezing the matrix to a more compact representation.

Alternatively, rather than building up the matrix, then freezing it, all matrix objects could be immutable,
and we can use the Builder pattern to build up the matrix state, representing it as a hash,
then when you call “to_matrix”, it would create an immutable matrix in a compact form.

Another option could be to pass the factory a coordinate-format (or compressed row / compressed column format) array
that it uses to place the non-zero elements.

* What are the important quality characteristics of a sparse matrix package? Reusability? Efficiency? Efficiency of what?

The sparse matrix package should definitely be more efficient for sparse matrices than the built-in matrix class.
It should be fast enough to create workable prototype code working with large sparse matrices.
Mainly it should avoid taking a large amount of memory,
and operations like finding the LU decomposition or the eigensystem should be much faster
than for a complete matrix of the same size.

The sparse matrix package should be maintainable (readable, testable, modifiable).

The sparse matrix package should be reusable in a number of different context,
which is why it's useful to provide different representations (i.e. support a tri-diagonal representation).

The sparse matrix package should be compatible with the built-in matrix class to assist users in switching to our software. From a business perspective, this can make our software much more attractive to customers.

* How do we generalize 2-D matrices to n-D matrices, where n > 2 – um, sounds like an extensible design?

Generalizing to n-D matrices from 2-D matrices is a tricky task because operations on n-D matrices have different implementations based
on their number of dimensions. For instance, matrix multiplication is a different operation/algorithm

based on the number of
dimensions. Our design could try to only use algorithms that work for n-D matrices but by doing this we are not able to use
the more efficient algorithms that only work on 2-D matrices. Another problem that comes up with generalizing is the way we store matrices
as with an n-D matrix we would need to store the matrices in such a way that any number of dimensions can be stored.

Also if our design was to be built to be extensible to n-D matrices then all our functions that deal with rows and columns would need to be
built to take a variable number of arguments such as for indices. Building our design to be extensible would add a large degree of complexity
to our design and would slow down our 2-D matrix operations so making our design extensible in this way is not worth the tradeoffs.