Matricies and Vectors

COMP2611: Data Structures
2019

Outline

- Basic Definitions
- Sparse Matrices
  - DOK, LIL, COO, CSR, CSC
- Special square matrices
  - Lower Triangular
  - Upper Triangular
  - Symmetric

What are Matricies?

- Rectangular array (2D array) of numbers or expressions. We call this a rank 2 tensor
- If a matrix has **n** rows and **m** cols, we call it an **n** $\times$ **m** matrix. We call these the matrix's dimensions

$$A = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix}$$

A is a 3 $\times$ 2 matrix

What are Matricies

- Each number or expression in the matrix is called an element or component
- Matricies are indexed by their rows and columns
  - Each element can be identified by a row, column pair
  - Matricies are typically `1`-indexed in Maths, but we implement using `0`-indexing. Need to translate between the two!
  - E.g. $A_{1,2}$ refers to the entry of matrix `A` at row `1`, column `2`. From the last slide, $A_{1,2} = 3$

Matricies in Programming

Python

```python
A = [
    [2, 3, 4],
    [5, 6, 7]
]
```

C++

```cpp
float A[][] = {
  {2, 3, 4},
  {5, 6, 7}
};
```

In Both of these languages, $A_{1,2}$ is at `A[0][1]`

Implemented as arrays of arrays

Vector

- Special shapes of matrices where we have either `1` row or `1` column.
- Since we know that it is has only a single column or row, we usually use a single index to identify a value when writing expressions mathematically.
- Depending on implementation, we still need both indexes when programming

Column Vectors

- Vectors where we have `1` column
- Usually we take column vectors as default
- When Maths textbooks say vector, translate to column vector!

$$v = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$$

$\mathtt{v}_3 \ \mathtt{=}\ \mathtt{4}$

Row Vectors

- Vectors with one row
- Regular single-dimensional arrays are seen as row vectors in many implementations
- Not the default in Maths though

$$w = [4 \quad 5 \quad 6]$$

$w_2 = 5$

Why Care About Matricies and Vectors?

- Primary objects of study in Linear Algebra - field of Mathematics
- Linear Algebra is important in the sciences and engineering
- CS is a field of Maths, Science, and Engineering
  - Should care be default!
- On the applied side Linear Algebra is part of reason behind computer's increased popularity

Why Care About Matricies and Vectors?

- Fortran was designed with Linear Algebra in mind!
- Matricies and Vectors used throughout CS:
  - Machine Learning
  - Simulation
  - Computer Modelling
  - Bioinformatics
  - Computer Graphics
  - Optimization
  - Game Programming, etc …

Libraries for Linear Algebra

- BLAS - Basic Linear Algebra Subroutines
- LAPACK - Linear Algebra Package
- Foundation for most other linear algebra libraries
  - Called using Foreign Function Interface (FFI)
- Other examples include:
  - Armadillo (C++)
  - NumPy (Python)
  - Julia - standard library
  - MATLAB - standard library

MAJOR ORDERS

Row major order
(Raster format)

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{23} & a_{33} \end{bmatrix}$$

Column major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{23} & a_{33} \end{bmatrix}$$

Some Operations on Matricies and Vectors

- Many operations, some include:
- Component-wise addition
  - Only if dimensions match
- Component-wise multiplication (Hadamard product)
  - Only if dimensions match
- Transpose
- Scalar Multiplication
- Matrix-Matrix multiplication
  - Dimensions must **agree**
  - Matrix-vector multiplication
  - Affine Transformation

Some Operations on Matricies and Vectors

- Many operations, some include:
- Component-wise addition
  - Only if dimensions match
- Component-wise multiplication (Hadamard product)
  - Only if dimensions match
- **Transpose**
- Scalar Multiplication
- **Matrix-Matrix multiplication**
  - Dimensions must **agree**
  - Matrix-vector multiplication
  - Affine Transformation

Transpose

- If **A** is a matrix (or vector) its transpose is written as $\mathbf{A}^\mathrm{T}$
- "Make rows columns"

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad A^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Tranpose Pseudocode

```
function transpose(A)
   num_rows = length(A)
   num_cols = length(A[0])
   A_t = create_array((num_cols, num_rows))
   for i = 0 to num_rows - 1:
      for j = 0 to num_cols - 1:
         A_t[j][i] = A[i][j]
   return A_t
```

Matrix Multiplication

- Important, but expensive operation
- Technically a form of function composition on linear transformation
- "Row into column"
- If **A** and **B** are matrices with dimensions $\mathbf{p} \times \mathbf{q}$ and $\mathbf{q} \times \mathbf{r}$ respectively , and $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, then

$$C_{i,j} = \sum_{k=1}^{q} A_{i,k} \cdot B_{k,j}$$

Matrix Multiplication

- number of columns in the 1st matrix = number of rows in the 2nd

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{bmatrix} = \begin{bmatrix} ? & ? \\ ? & ? \end{bmatrix}$$

Matrix Multiplication

- number of columns in the 1st matrix = number of rows in the 2nd

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{bmatrix} = \begin{bmatrix} 1 \cdot 4 + 2 \cdot 6 + 3 \cdot 8 & ? \\ ? & ? \end{bmatrix}$$

Matrix Multiplication

- number of columns in the 1st matrix = number of rows in the 2nd

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{bmatrix} = \begin{bmatrix} 40 & ? \\ ? & ? \end{bmatrix}$$

Matrix Multiplication

- number of columns in the 1st matrix = number of rows in the 2nd

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{bmatrix} = \begin{bmatrix} 40 & 1 \cdot 5 + 2 \cdot 7 + 3 \cdot 9 \\ ? & ? \end{bmatrix}$$

Matrix Multiplication

- number of columns in the 1st matrix = number of rows in the 2nd

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{bmatrix} = \begin{bmatrix} 40 & 46 \\ ? & ? \end{bmatrix}$$

Matrix Multiplication

- number of columns in the 1st matrix = number of rows in the 2nd

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{bmatrix} = \begin{bmatrix} 40 & 46 \\ 4 \cdot 4 + 5 \cdot 6 + 6 \cdot 8 & ? \end{bmatrix}$$

Matrix Multiplication

- number of columns in the 1st matrix = number of rows in the 2nd

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{bmatrix} = \begin{bmatrix} 40 & 46 \\ 94 & ? \end{bmatrix}$$

Matrix Multiplication

- number of columns in the 1st matrix = number of rows in the 2nd

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{bmatrix} = \begin{bmatrix} 40 & 46 \\ 94 & 4 \cdot 5 + 5 \cdot 7 + 6 \cdot 9 \end{bmatrix}$$

Matrix Multiplication

- number of columns in the 1st matrix = number of rows in the 2nd

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{bmatrix} = \begin{bmatrix} 40 & 46 \\ 94 & 109 \end{bmatrix}$$

Matrix Multiplication Pseudocode

```
function matrix_mult(A, B)
   num_rows = length(A)
   num_cols = length(A[0])
   num_rows1 = length(B)
   num_cols1 = length(B[0])
   if num_cols != num_rows1:
      error()
   C = create_array((num_rows, num_cols1))
   fill_array(C, 0)
   for i = 0 to num_rows - 1:
      for j = 0 to num_cols1 - 1:
         for k = 0 to num_rows1 - 1:
            C[i][j] = A[i][k] * B[k][j]
   return C
```

Matrix Multiplication

- if **`n = max(p, q, r)`**, then this algorithm is **$O(n^3)$**
- Expensive for large **`n`**
- Approximation algorithms and parallelism used in practice to perform matrix multiplication

Sparse Matricies

- Matrix multiplication is the backbone of many algorithms using matrices
- Matrix multiplication, as we just saw, is expensive
- A matrix where most entries are 0 is called a sparse matrix
- What happens if most of our matrix is 0?

Sparse Matrix Multiplication

$$\begin{bmatrix} 1 & 0 & 3 \\ 4 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 5 \\ 0 & 0 \\ 8 & 0 \end{bmatrix} = \begin{bmatrix} ? & ? \\ ? & ? \end{bmatrix}$$

Sparse Matrix Multiplication

$$\begin{bmatrix} 1 & 0 & 3 \\ 4 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 5 \\ 0 & 0 \\ 8 & 0 \end{bmatrix} = \begin{bmatrix} 1 \cdot 0 + 0 \cdot 0 + 3 \cdot 8 & 1 \cdot 5 + 0 \cdot 0 + 3 \cdot 0 \\ 4 \cdot 0 + 0 \cdot 0 + 0 \cdot 8 & 4 \cdot 5 + 0 \cdot 0 + 0 \cdot 0 \end{bmatrix}$$

Sparse Matrix Multiplication

$$\begin{bmatrix} 1 & 0 & 3 \\ 4 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 5 \\ 0 & 0 \\ 8 & 0 \end{bmatrix} = \begin{bmatrix} 24 & 5 \\ 0 & 20 \end{bmatrix}$$

Sparse Matrices

- Matrix multiplication becomes "simplier" if most entries are `0`s.
- All of these `0`s are stored, yet have no effect on outcomes
  - This is wasted space
- Different ways to store sparse matrices to minimise space wastage and make write faster algorithms
- Some are better under some circumstances than others

Sparse Matrix Representations

- DOK - Dictionary of Keys
- LIL - List of Lists
- COO - Coordinate List
- CSR - Compressed Sparse Row
- CSC - Compressed Sparse Column

Dictionary of keys

- Store only non-zero entries in a dictionary
  - Keys are coordinates (indicies)
  - Values are the entries
- Fine for addition, scalar multiplication, and hadamard product
- Not that great for matrix multiplication
- Great as intermediary between other formats and great for direct access of entries

```
function mat_to_dok(A)
    D = create_dict()
    num_rows = length(A)
    num_cols = length(A[0])
    for i = 0 to num_rows - 1:
        for j = 0 to num_cols - 1:
            if A[i][j] != 0:
                insert(D,(str(i)+”,”+str(j)), A[i][j])
    return D
```

Key

Value

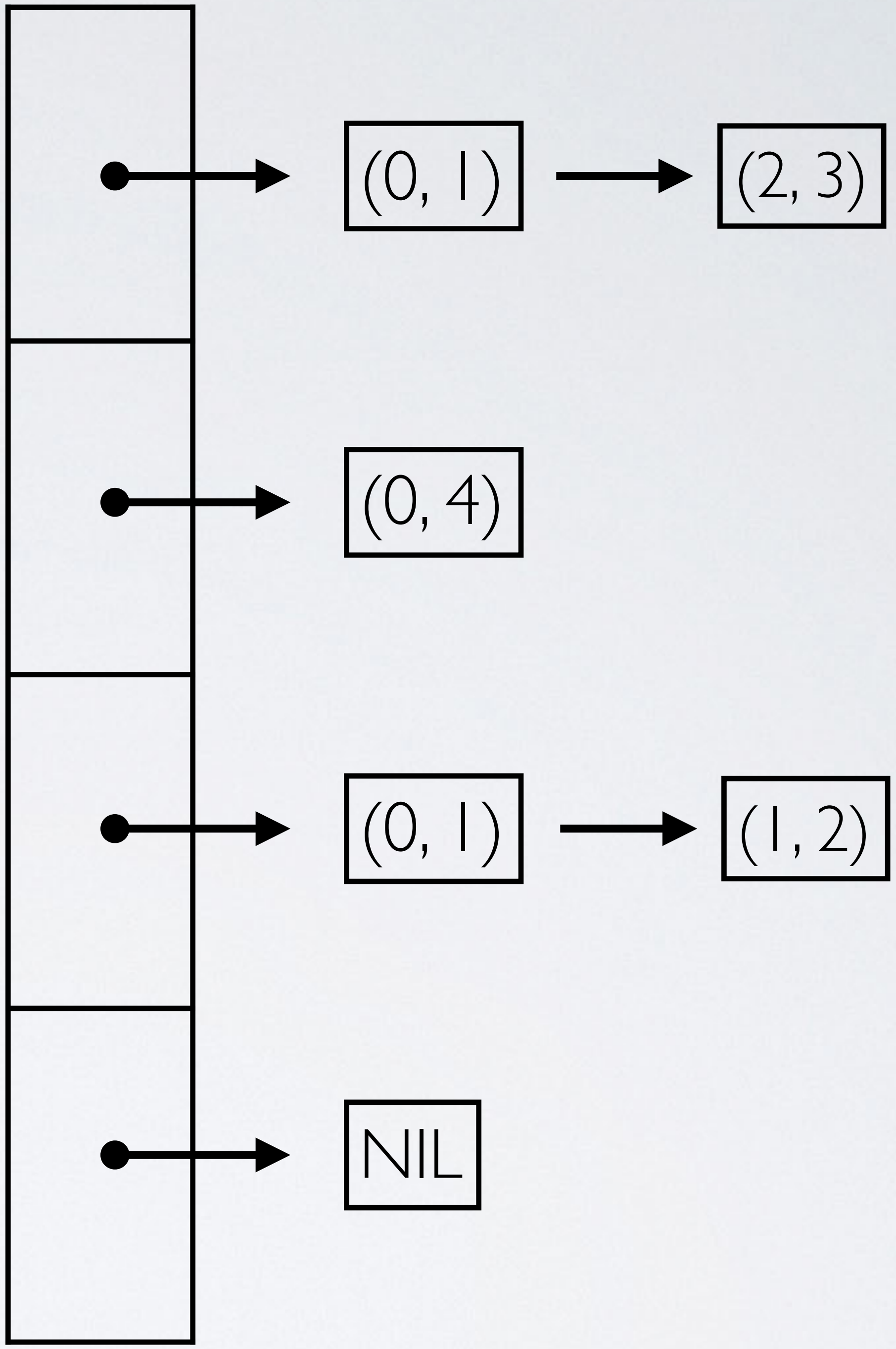
```
function mat_to_dok(A)
    D = create_dict()
    num_rows = length(A)
    num_cols = length(A[0])
    for i = 0 to num_rows - 1:
        for j = 0 to num_cols - 1:
            if A[i][j] != 0:
                insert(D,(str(i)+”,”+str(j)), A[i][j])
    return D
```

```
function dok_get_entry(D, i, j)
    key = str(i) + "," + str(j)
    value = search(D, key)
    if value == NIL:
        return 0
    return value
```

LIL

- Store only non-zero entries in a jagged 2D array or an array of link lists
  - jagged 2D array is an array of arrays
  - inner arrays need not be the same length
  - each inner array associated with a row
  - each entry in LIL comprises of (col, entry) pair sorted by col
- Good for addition and scalar multiplication
- Not that great for matrix multiplication
- Great as intermediary between other formats and good for direct access of entries (have some searching for entry)

$$A = \begin{bmatrix} 1 & 0 & 3 \\ 4 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$



```
function mat_to_lil(A)
    L = create_array()
    num_rows = length(A)
    num_cols = length(A[0])
    for i = 0 to num_rows - 1:
        L.append(create_array())
    for i = 0 to num_rows - 1:
        for j = 0 to num_cols - 1:
            if A[i][j] != 0:
                L[i].append((j, A[i][j]))
    return L
```

```
function lil_get_entry(L, i, j)
    l = L[i]
    for (col, entry) in l:
       if col == j:
          return entry
       elif col > j:
          break
    return 0
```

COO

- Store non-zero entries as sorted array of triples
  - Each triple contains (row, col, entry)
  - Sorted by row, then by col if rows are equal
- Can use binary search for retrieval
- Good as intermediary format

$$A = \begin{bmatrix} 1 & 0 & 3 \\ 4 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

(0, 0, 1)	(0, 2, 3)	(1, 0, 4)	(2, 0, 1)	(2, 1, 2)

```
function mat_to_coo(A)
    L = create_array()
    num_rows = length(A)
    num_cols = length(A[0])
    for i = 0 to num_rows - 1:
        for j = 0 to num_cols - 1:
            if A[i][j] != 0:
                L.append((i, j, A[i][j]))
    return L
```

```
function cmp(i1, j1, i2, j2)
    if i1 == i2:
        if j1 == j2:
            return 0
        return (j1 > j2) ? 1: -1
    return (i1 > i2) ? 1: -1
```

```
function coo_get_entry(L, i, j)
    lo = 0
    hi = length(L) - 1
    while hi >= lo:
        mid = (hi + lo) / 2
        (i2, j2, entry) = L[mid]
        if cmp(i, j, i2, j2) == 0:
            return entry
        if cmp(i, j, i2, j2) > 1:
            hi = mid - 1
        else:
            lo = mid + 1
    return 0
```

Yale Formats

- CSR and CSC are two versions of the Yale formats of sparse matrix representation
- Use three arrays to store non-zero entries
- **Best** format for sparse matrix multiplication
- **Great** for splicing across rows (CSR) or columns (CSC)

Yale Formats

- Both CSR and CSC use three arrays:
  - **`A`** - stores the non-zero entries
  - **`IA`** - recursively defined as follows
    - **`IA[0] = 0`**
    - **`IA[i] = IA[i - 1] +`** (num non-zero elements in **`i-1`** row (CSR) or column (CSC))
  - **`JA`** - stores the column index (CSR) or row index (CSC) of the entries in A
- **`A[IA[i]]`** to **`A[IA[i+1] - 1]`** hold the entries from row **`i`** (CSR) or column **`i`** (CSC)

$$M = \begin{bmatrix} 1 & 0 & 3 \\ 4 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$A = [1 \quad 3 \quad 4 \quad 1 \quad 2]$$
$$IA = [0 \quad 2 \quad 3 \quad 5 \quad 5 \quad 5]$$
$$JA = [0 \quad 2 \quad 0 \quad 0 \quad 1]$$

$$A = [1 \quad 3 \quad 4 \quad 1 \quad 2]$$
$$IA = [0 \quad 3 \quad 4 \quad 5]$$
$$JA = [0 \quad 0 \quad 1 \quad 2 \quad 2]$$

```
function mat_to_csr(M)
    A = create_array()
    IA = create_array()
    IA.append(0)
    JA = create_array()
    num_rows = length(A)
    num_cols = length(A[0])
    for i = 0 to num_rows - 1:
        count = 0
        for j = 0 to num_cols - 1:
            if A[i][j] != 0:
                A.append(A[i][j])
                JA.append(j)
                count += 1
        IA.append(last_element(IA) + count)
    return A, IA, JA
```

```
function row_slice_csr(A, IA, i)
    return A[IA[i]:(IA[i + 1] - 1)]
```

```
function get_entry_csr(A, IA, JA, i, j)
    starting_index = IA[i]
    ending_index = IA[i + 1] - 1
    for k = starting_index to ending_index:
        if JA[k] == j:
            return A[k]
    return 0
```

Will look at CSC in Lab. Should try implementing on your own first

Other Special Matrices

- A matrix with the same number of rows and columns is called a square matrix
- There are special types of square matrices:
  - Triangular
    - Upper and Lower
  - Symmetric

Triangular Matrices

- Either the upper half or lower half is completely filled

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{bmatrix}$$

Upper triangular

$$\begin{bmatrix} 0 & 0 & a_{13} \\ 0 & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Lower triangular

Storing Triangular Arrays

- Can store using a single array. Map 2D index to 1D
- Array contains $\sum_{i=1}^{n} i$ entries
    - $$\sum_{i=1}^{n} i = \frac{1}{2}n(n+1)$$

Lower Triangular

- How do we map 2D indices to 1D?
- Use AP
- If we want entry at `(i, j)` we need to move past all elements before index `i`. The entry at `(i, j)` is then `j` positions away

$$\begin{bmatrix} 0 & 0 & 3 \\ 0 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad \begin{bmatrix} 3 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

Lower Triangular

- `index(i, j) = (j - 1) +` $\sum_{p=1}^{i} p$
- `index(i, j) = (j - 1) +` $\frac{1}{2}i(i+1)$

$$\begin{bmatrix} 0 & 0 & 3 \\ 0 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \begin{bmatrix} 3 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

Upper Triangular

- Reverse situation of Lower triangular
- If we want entry at `(i, j)` we need to move past all elements such that we have index `i + (i - 1) + (i - 2) + . . . 1` entries left. The entry at `(i, j)` is then `j` positions away

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 5 & 6 \\ 0 & 0 & 9 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 5 & 6 & 9 \end{bmatrix}$$

Upper Triangular

- `index(i, j) = (j - 1) +` $(\sum_{p=1}^{n} p) - \sum_{p=1}^{i} p$
- `index(i, j) = (j - 1) +` $\frac{1}{2}n(n+1) - \frac{1}{2}i(i-1)$

n is the dimension

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 5 & 6 \\ 0 & 0 & 9 \end{bmatrix} \qquad [1 \quad 2 \quad 3 \quad 5 \quad 6 \quad 9]$$

Returning 0s

- If Upper triangular where `j` is the column and `i` is the row, if `j > i`, then return `0`
- If Lower triangular where `j` is the column and `i` is the row, if `j < i`, then return `0`

Symmetric Matrices

- Both half of the matrix are the same
- The transpose of such a matrix is itself
- Examples: any undirected graph
- Store as Lower Triangular
  - Except if we query `j < i`, return the same entry at `index(i, j)`