Outline

**Visual Studio**: start project, debug program

**Program Structure**: iostream, std, main, sequential, return, comment

**Integer Variable**: declare, assign, initialize, arithmetic, input/output, round, unsigned, constant.

**Array Variable**: declaration, index, initialization, multi-dimensional array, sparse matrix

**Instruction**: arithmetic/increment, relational/logical, bitwise, if-else, switch, while-loop, for-loop, break

**Pointer Variable**: declaration, relation to integer variable, access content, relation to integer array

**Other Variables**: float, char, enum, vector, string, structure

**Function**: definition, declaration, assignment passing, local/global variable, overload

**Class**: definition, public/private member, constructor/destructor, declaration, memory
Declare an array of size T.

```cpp
int main()
{
    int x[3];  // syntax: int (array name) [array size]
    x[0] = 4;
    cin >> x[2];
    x[1] = x[0] + x[2];
    return 0;
}
```

declare an array of integers with size T.
every element in the array is an integer.
Index elements in an array.

```cpp
int main()
{
    int x[3];

    x[0] = 4;
    syntax: (array name)[i-1]

    cin >> x[2];

    x[1] = x[0] + x[2];

    return 0;
}
```

Access the ith element by x[i-1].

index range: 0, 1, ..., T-1.

x[0] is the first element.
Operate on array elements.

```cpp
int main()
{
    int x[3];
    x[0] = 4;
    cin >> x[2];
    x[1] = x[0] + x[2];
    return 0;
}
```

Play with indexed elements as if they are regular integer variables.

- `x[0]` is the 1st element.
- `x[1]` is the 2nd element.
- `x[2]` is the 3rd element.
An intuitive example on array.

```cpp
int main()
{
    int x[3];
    x[0] = 4;
    cin >> x[2];
    x[1] = x[0] + x[2];
    return 0;
}
```

An intuitive example on array.

```c
int main()
{
    int x[3];

    x[0] = 4;
    cin >> x[2];

    x[1] = x[0] + x[2];

    return 0;
}
```
An intuitive example on array.

```c
int main()
{
    int x[3];
    x[0] = 4;
    cin >> x[2];
    x[1] = x[0] + x[2];
    return 0;
}
```

suppose we input 7.
An intuitive example on array.

```cpp
int main()
{
    int x[3];
    x[0] = 4;
    cin >> x[2];
    x[1] = x[0] + x[2];
    return 0;
}
```

array “x”

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>11</td>
<td>7</td>
</tr>
</tbody>
</table>

x[0]  x[1]  x[2]
Initialize an array.

```cpp
int main()
{
    x[2] = x[0] + x[1];  // syntax: { Val1, Val2, Val3, ... }.
    cout << x[2];
    return 0;
}
```

We can also initialize elements of an array.

Val1 goes to 1st element.
Val2 goes to 2nd element.
Val3 goes to 3rd element.
......
Intuitive example on array initialization.

```cpp
int main()
{
    x[2] = x[0] + x[1];
    cout << x[2];
    return 0;
}
```
Multi-Dimensional Array: Declaration

```cpp
int main()
{
    int x[3][2];  // syntax: int (name)[size of 1st dim][size of 2nd dim]
    x[2][0] = 5;
    cin >> x[1][1];
    x[0][1] = x[1][1] * x[2][0];
    return 0;
}
```

The 1st dimension has three elements.
The 2nd dimension has two elements.
In total, there are 6 = 3*2 elements.
Multi-Dimensional Array: Indexing

Access an element

\[ \text{x}[i-1][j-1] \]

It is intersection of the

- \( i \)th in the 1st dimension
- \( j \)th in the 2nd dimension

```c
int main()
{
    int x[3][2];
    x[2][0] = 5;
    cin >> x[1][1];
    x[0][1] = x[1][1] * x[2][0];
    return 0;
}
```

Each index cannot exceed the range in its own dimension, e.g., \( x[i-1][j-1] \)

- \( i \) in \{1,2,3\}
- \( j \) in \{1,2\}
int main()
{
    int x[3][2];

    x[2][0] = 5;

    cin >> x[1][1];

    x[0][1] = x[1][1] * x[2][0];

    return 0;
}
Multi-Dimensional Array: An intuitive example

```c
int main()
{
    int x[3][2];
    x[2][0] = 5;
    cin >> x[1][1];
    x[0][1] = x[1][1] * x[2][0];
    return 0;
}
```

A 2D array is a matrix!

1st dimension is row;
2nd dimension is column.

<table>
<thead>
<tr>
<th>x[..][0]</th>
<th>x[..][1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Multi-Dimensional Array: An intuitive example

```c
int main()
{
    int x[3][2];
    x[2][0] = 5;
    cin >> x[1][1];
    x[0][1] = x[1][1] * x[2][0];
    return 0;
}
```

3rd in row, 1st in column.
Multi-Dimensional Array: An intuitive example

```cpp
int main()
{
    int x[3][2];
    x[2][0] = 5;
    cin >> x[1][1];
    x[0][1] = x[1][1] * x[2][0];
    return 0;
}
```

Suppose we input 7.
Multi-Dimensional Array: An intuitive example

```cpp
int main()
{
    int x[3][2];
    x[2][0] = 5;
    cin >> x[1][1];
    x[0][1] = x[1][1] * x[2][0];
    return 0;
}
```

1st in row, 2nd in column.
Multi-Dimensional Array: Initialization

We can initialize elements of a multidimensional array.

```cpp
int main()
{
    int x[3][2] = {
        {1,2},
        {3,4},
        {5,6}
    };

    cout << x[1][1];

    return 0;
}
```

A 1D array is initialized by a set.

A multi-dim array is by a *nested set*.

1. A set of three elements.
2. Each is a set of two elements.
3. Elements are separated by “,”.
Multi-Dimensional Array: An intuitive example.

```cpp
int main()
{
    int x[3][2] = {
        {1, 2},
        {3, 4},
        {5, 6}
    };

    cout << x[1][1];

    return 0;
}
```
Multi-Dimensional Array: An intuitive example.

```cpp
int main()
{
    int x[3][2] = {
        {1,2},
        {3,4},
        {5,6}
    };

    cout << x[1][1];  // What's the output?
    return 0;
}
```

<table>
<thead>
<tr>
<th></th>
<th>x[.][0]</th>
<th>x[.][1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0][.]</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>x[1][.]</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>x[2][.]</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Multi-Dimensional Array: An intuitive example.

```cpp
int main()
{
    int x[3][2] = {
        {1, 2},
        {3, 4},
        {5, 6}
    };

    cout << x[1][1];  // What's the output?

    return 0;
}
```

<table>
<thead>
<tr>
<th></th>
<th>x[.][0]</th>
<th>x[.][1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0][.]</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>x[1][.]</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>x[2][.]</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Multi-Dimensional Array: More examples on initialization and indexing

Declare a 3D array.

```cpp
int main()
{
    int x[3][2][2] = {
        {{1,2},{3,4}},
        {{5,6},{7,8}},
        {{9,10},{11,12}}
    };

    cout << x[1][1][1];

    return 0;
}
```

1. There are three sets.
2. Each set is a 2D array.

Can you guess the output?
int main()
{
    int x[3][2][2] = {
        {{1,2},{3,4}},
        {{5,6},{7,8}},
        {{9,10},{11,12}}
    };

    cout << x[1][1][1];

    return 0;
}
Multi-Dimensional Array: More examples on initialization and indexing

```cpp
int main()
{
    int x[3][2][2] = {
        {{1,2},{3,4}},
        {{5,6},{7,8}},
        {{9,10},{11,12}}
    };

    cout << x[1][1][1];

    return 0;
}
```
Multi-Dimensional Array: More examples on initialization and indexing

```cpp
int main()
{
    int x[3][2][2] = {
        {{1,2},{3,4}},
        {{5,6},{7,8}},
        {{9,10},{11,12}}
    };

    cout << x[1][1][1];

    return 0;
}
```
Sparse Matrix: Background

```cpp
int main()
{
    int x[3][2] = {
        {32, 4},
        {6, 7},
        {9, 11}};

    x[2][1] = 8;
    cout << x[0][0];

    return 0;
}
```

We already know a matrix can be implemented using a 2D array.

This is a matrix!

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>
A **sparse matrix** is one with massive repeated elements, typically 0’s.

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Sparse Matrix: why bother?

We can store a sparse matrix using a 2D array (int x[4][7]), but it is a waste of space...

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Or, we can store the matrix only through its non-zero entries, using a smaller 2D array.

Sparse Matrix: a more efficient data structure.

store 28 integers (=28*4 bytes)

\[
\begin{array}{cccccc}
0 & 2 & 0 & 0 & 6 & 0 \\
0 & 0 & 0 & 5 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\]

store 15 integers (=15*4 bytes)

\[
\begin{array}{ccc}
0 & 1 & 2 \\
0 & 5 & 6 \\
1 & 4 & 5 \\
3 & 0 & 3 \\
3 & 5 & 1 \\
\end{array}
\]
Sparse Matrix: an intuitive example of the efficient data structure.

Idea: only store location and value of non-zero entries.

Element at 1st row 2nd column is 2.
Sparse Matrix: an intuitive example of the efficient data structure.

Idea: only store location and value of non-zero entries.

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>
Sparse Matrix: an intuitive example of the efficient data structure.

Idea: only store location and value of non-zero entries.

Element at 2nd row 5th column is 5.
Sparse Matrix: an intuitive example of the efficient data structure.

Idea: only store location and value of non-zero entries.

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Element at 4th row 1th column is 3.
Sparse Matrix: an intuitive example of the efficient data structure.

Idea: only store location and value of non-zero entries.

Element at 4th row 6th column is 1.
Sparse Matrix: an intuitive example of the efficient data structure.

Idea: only store location and value of non-zero entries.

```
<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

All the undescribed elements are 0.

```
<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>
```
Sparse Matrix: Implementation

We can implement a sparse matrix using three 1D arrays. (row, column, value)

```java
int main()
{
    int r[5] = { 0,0,1,3,3 };
    int c[5] = { 1,5,4,0,5 };
    int e[5] = { 2,6,5,3,1 };
    return 0;
}
```

Need to know # non-zero elements upfront before declaration.
Sparse Matrix: Implementation

We can implement a sparse matrix using three 1D arrays. (row, column, value)

```c
int main()
{
    int r[5] = { 0,0,1,3,3 };  
    int c[5] = { 1,5,4,0,5 };  
    int e[5] = { 2,6,5,3,1 };  
    return 0;
}
```

use index i-1 to retrieve ith record, e.g., r[3],c[3],e[3].

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>
Sparse Matrix: Implementation

Or, using a 2D array, with each row storing (row, column, value).

```c
int main()
{
    int x[5][3] = {
        { 0, 1, 2 },
        { 0, 5, 6 },
        { 1, 4, 5 },
        { 3, 0, 3 },
        { 3, 5, 1 },
    };

    return 0;
}
```

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>
Sparse Matrix: Implementation

Or, using a 2D array, with each row storing (row, column, value).

```c
int main()
{
    int x[5][3] = {
        { 0, 1, 2 },
        { 0, 5, 6 },
        { 1, 4, 5 },
        { 3, 0, 3 },
        { 3, 5, 1 },
    };
    return 0;
}
```

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>
But there is no free lunch...

Efficient data structure of sparse matrix makes operations more complicated to implement.

Below is matrix addition, i.e., element wise addition of matrix elements.

<table>
<thead>
<tr>
<th>x[3][2]</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>y[3][2]</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

=  

<table>
<thead>
<tr>
<th>z[3][2]</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>
But there is no free lunch...

Addition is easy to implement if all matrices are stored using regular 2D arrays.

Run $z[i][j] = x[i][j] + y[i][j]$ over all indices (i,j).

* This can be more efficiently implemented using “for-loop” (will be introduced later.)
But there is no free lunch...

Q: How do we add the following two matrices based on the efficient data structure?

Sparse Matrix X (4-by-7)

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Sparse Matrix Y (4-by-7)

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

= ?
Sparse Matrix Addition: Discussion

Suppose Z is implemented by efficient data structure.

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Sparse Matrix X

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Sparse Matrix Y

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sparse Matrix Z
Sparse Matrix Addition: Discussion

Q1: Will these records appear in the table of Z? Why?

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Sparse Matrix X

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Sparse Matrix Y

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>?</td>
</tr>
</tbody>
</table>

Sparse Matrix Z
Sparse Matrix Addition: Discussion

Q2: What are the values of these records? Why?

Sparse Matrix X

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Sparse Matrix Y

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Sparse Matrix Z

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>?</td>
</tr>
</tbody>
</table>
Sparse Matrix Addition: Discussion

Let’s go over the idea again.

### Sparse Matrix X

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

### Sparse Matrix Y

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ \text{Sparse Matrix } X + \text{Sparse Matrix } Y = \text{Sparse Matrix } Z \]
A more challenging task: sparse matrix multiplication.

**Q:** How do we multiply the following two matrices based on the efficient data structure?

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Sparse Matrix X (4-by-7)

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Sparse Matrix Y (4-by-7)

Multiply the matrices to get the result.