

- Sparse matrix (희소행렬 : 행렬의 값이 대부분 0인 행렬).

• 희소행렬을 효율적으로 저장하기 위해 아래와 같은 자료구조를 생각할 수 있다.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	
[0]	0	0	2	0	0	0	12	<0, 2, 2>
[1]	0	0	0	0	7	0	0	<0, 6, 12>
[2]	23	0	0	0	0	0	0	<1, 4, 7>
[3]	0	0	0	31	0	0	0	<2, 0, 23>
[4]	0	14	0	0	0	25	0	<3, 3, 31>
[5]	0	0	0	0	0	0	6	<4, 1, 14>
[6]	52	0	0	0	0	0	0	<4, 5, 25>
[7]	0	0	0	0	11	0	0	<5, 6, 6>
[6]								<6, 0, 52>
[7]								<7, 4, 11>

희소행렬

해당 행렬의 값이 0이 아닌 것들만 이 리스트의 원소로 취해함.

• SparseMatrix 자료구조

- 행렬의 dimension m, n 을 입력받아 희소행렬 리스트를 $[[m, n, 0]]$ 로 초기 설정한다.
- append method: $[i, j, value]$ 형태의 리스트를 희소행렬 리스트에 추가한다.
- shape method: 희소행렬 dimension을 리턴한다.
- getValue method: i, j 행렬값을 리턴한다.
- print method: 희소행렬을 일반적 행렬 형태로 프린트한다. (0으로 구성된 m, n array를 만들고 i, j 행렬값을 업데이트한다.)

↳ 해당 희소행렬을
numpy array class type의
객체로 나타냄!!!

X. SparseMatrix의 '0'이 아닌 행렬 값을 전부 불러와야 할 때는 해당 SparseMatrix를 요약한 list의 '첫번째' 요소부터 검사해야한다. because, 해당 리스트의 0번째는 해당 SparseMatrix를 요약한 정보가 들어가있기 때문이다.

```

class SparseMatrix:
    def __init__(self, m, n):
        self.s = [[0, n, m]]
        self.m = m
        self.n = n

    def append(self, i, j, val):
        if val != 0:
            self.s.append([i, j, val])
            self.s[0][2] = len(self.s) - 1

    def shape(self):
        return (self.m, self.n)

    def getValue(self, i, j):
        val = 0
        for k in range(len(self.s)):
            if self.s[k][0] == i and self.s[k][1] == j:
                return self.s[k][2]
        return 0

    def print(self):
        import numpy as np
        _tmp = np.zeros((self.m, self.n))
        for i in range(self.m):
            _tmp[self.s[i][0]-1, self.s[i][1]-1] = self.s[i][2]
        print(_tmp)

a = SparseMatrix(3, 3)
b = SparseMatrix(3, 3)

a.append(1, 1, 1)
a.append(2, 2, 2)
a.append(3, 3, 3)
b.append(1, 1, 4)
b.append(1, 2, 7)
b.append(2, 3, 2)
b.append(3, 3, 1)

a.print()
b.print()

print(a.getValue(3, 3))

```

← SparseMatrix 리스트 내의 첫번째 원소는 해당 희소행렬의 행과 열에 대한 정보가 포함된 '리스트' 객체이다.

← 당연히 해당 리스트 내에는 행렬 값이 0이 아닌 것들만 추가되어야 한다.

← 해당 리스트 내에 '0'이 아닌 행렬값이 들어갔으면, SparseMatrix 리스트의 첫번째 요소를 업데이트 해주어야 함.

← SparseMatrix 원소들을 차례대로 return 은 함수를 종료하고 값을 해당 함수에 돌려주는 역할을 함.

← Matrix는 (1,1)부터 시작한다.

← 이걸이 (-1)을 빼줘야 우리가 배웠던 행렬형태가 나타난다.

← * numpy.array로 만든 행렬 (<=> 2차 배열)은 (0,0)부터 시작한다.

← 중요한 차이점!!!

```

[[[1, 0, 0],
 [0, 2, 0],
 [0, 0, 3]],
 [[4, 7, 0],
 [0, 0, 2],
 [0, 0, 1]]]
3

```

```

def transpose(a):
    transposed_a = SparseMatrix(a.n, a.m)

    for i in range(len(a.s)):
        if i == 0:
            transposed_a.s[0][2] = a.s[0][2]
        else:
            transposed_a.append(a.s[i][1], a.s[i][0], a.s[i][2])

    return transposed_a

transposed_b = transpose(b)
transposed_b.print()

```

← 이걸은 SparseMatrix를 위치한다.

← transposed 행렬 내이며, '0'이 아닌 행렬값의 개수를 위치시키기 위한 행렬과 동일하게 설정해줘야 함.

← ex) (2,1)이 존재하던 행렬값을 (1,2)에 넣어야 함.

```

[[[4, 0, 0],
 [7, 0, 0],
 [0, 2, 1]]]

```

```

def add(a, b):
    if a.shape() != b.shape():
        print("두 행렬의 차원이 달라 더할 수 없습니다.")
    else:
        c = SparseMatrix(a.shape()[0], a.shape()[1])
        u = a.s
        v = b.s
        for i in range(len(u)):
            u.add(a.s[i][0], a.s[i][1], u.s[i][2])
            for j in range(len(v)):
                u.add(b.s[j][0], b.s[j][1], v.s[j][2])
        for item in u:
            _tmp = a.getValue(item[0], item[1]) + b.getValue(item[0], item[1])
            if _tmp != 0:
                c.append(item[0], item[1], _tmp)
        return c

a.print()
b.print()
c = add(a, b)
c.print()

```

← 더 해질 두 개체는 동일한 모양을 띄고 있어야 한다.

← 더 해주어서 나온 결과의 행렬도 당연히 더 해지는 두 행렬의 모양과 같아야 한다.

← 더해진 두 좌표값의 행렬값을 더하여 'tmp' 변수에 저장함.

← 좌표값을 비어있는 집합에 넣는다!!!

```

[[[1, 0, 0],
 [0, 2, 0],
 [0, 0, 3]],
 [[4, 7, 0],
 [0, 0, 2],
 [0, 0, 1]],
 [[5, 7, 0],
 [0, 2, 2],
 [0, 0, 4]]]

```