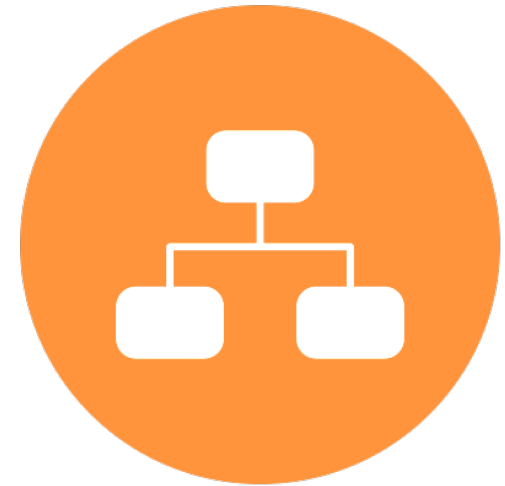


# Data Structure

## 2<sup>nd</sup> Study: Array

- Polynomial
- Sparse Matrix
- String Pattern Match: KMP Algorithm



C++ Korea 옥찬호 (utilForever@gmail.com)

# Polynomial

# What is polynomial?

- $a(x) = 3x^3 + 2x^2 - 4x$ ,  $b(x) = x^8 - 10x^5 - 3x^3 + 1$
- 다항식  $a(x)$ 는 3개의 항  $3x^2$ ,  $2x$ ,  $-4$ 를 가짐
  - 이 항들의 계수(Coefficient)는 3, 2, -4
  - 이 항들의 지수(Exponent)는 2, 1, 0
  - 다항식의 항은 (계수, 지수) 쌍으로 표현할 수 있음
    - 예를 들어, (3, 2)는 항  $3x^2$ 를 나타냄
- 다항식의 덧셈과 곱셈에 대한 표준 수학적 정의  
 $a(x) = \sum a_i x^i$ ,  $b(x) = \sum b_i x^i$ 로 된 2개의 다항식이 있을 때,
  - 덧셈 :  $a(x) + b(x) = \sum (a_i + b_i) x^i$
  - 곱셈 :  $a(x) \cdot b(x) = \sum (a_i x^i \cdot \sum (b_j x^j))$

# Polynomial ADT

```
// Polynomial: A set of pairs, <e_i, a_i>
// a_i: coefficient (not 0, float type)
// e_i: exponent (not negative, int type)
class Polynomial
{
public:
    // Create polynomial  $p(x) = 0$ 
    Polynomial();

    // Add polynomial *this and poly
    const Polynomial Add(const Polynomial& poly);
    // Multiply polynomial *this and poly
    const Polynomial Multiply(const Polynomial& poly);

    // After assign f into x, evaluate polynomial and return result
    const float Eval(const float& f);
};
```

# Polynomial Representation

- 데이터 멤버 정의

```
private:  
    static const int MaxDegree = 100;  
  
    int degree; // degree <= MaxDegree  
    float coef[MaxDegree + 1]; // Coefficient array
```

- 문제점

- 다항식의 최대 차수를 알아야 함
- 메모리가 낭비되는 경우가 많음
- 예를 들어, degree가 3인 경우 98개의 항은 사용하지 않음
- 문제를 해결하기 위해선, 필요한 항만큼 메모리 공간을 할당하면 됨

# Polynomial Representation

- 개선 1 : 동적 메모리 할당

```
private:
    int degree;
    float* coef; // Coefficient array

Polynomial::Polynomial(int d)
    : degree(d), coef(new float[degree + 1]) { }
```

- 문제점

- 다항식에서 대부분의 항이 0인 경우
- 예를 들어, 다항식  $x^{1000} + 1$ 은 999개의 항이 0임
- 0인 항이 대부분인 경우, 메모리 낭비 → 희소(Sparse) 다항식
- 문제를 해결하기 위해선, 0이 아닌 항만 저장하면 됨

# Polynomial Representation

- 개선 2 : 클래스 Term 정의

```
class Polynomial; // Forward Declaration
class Term
{
    friend Polynomial;
private:
    float coef; // Coefficient
    int exp; // Exponent
};

class Polynomial
{
private:
    Term* termArray; // Term array (not 0)
    int capacity; // Capacity of termArray
    int terms; // Number of terms (not 0)
    ...
};
```

$$a(x) = 3x^3 + 2x^2 - 4x$$

coef	3	2	-4
exp	3	2	1

$$b(x) = x^8 - 10x^5 - 3x^3 + 1$$

coef	1	-10	-3	1
exp	8	5	3	0

# Polynomial Addition

```
const Polynomial Polynomial::Add(const Polynomial& b)
{ // Return *this + b
    Polynomial c;
    int aPos = 0, bPos = 0;

    while ((aPos < terms) && (bPos < b.terms)) {
        if (termArray[aPos].exp == b.termArray[bPos].exp) {
            float t = termArray[aPos].coef + b.termArray[bPos].coef;
            if (t) c.NewTerm(t, termArray[aPos].exp);
            aPos++; bPos++;
        }
        else if (termArray[aPos].exp < b.termArray[bPos].exp) {
            c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
            bPos++;
        }
        else {
            c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);
            aPos++;
        }
    }
}
```



# Polynomial Addition

```
// Add rest terms of *this
for (; aPos < terms; aPos++)
    c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);
// Add rest terms of b(x)
for (; bPos < b.terms; bPos++)
    c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);

return c;
}
```

# Polynomial Addition

```
void Polynomial::NewTerm(const float& coef, const int& exp)
{ // Add new term into into last element of termArray
    if (terms == capacity)
    { // Expand the size of termArray twice
        capacity *= 2;
        Term* temp = new Term[capacity]; // New array
        std::copy(termArray, termArray + terms, temp);
        delete[] termArray; // Return original array
        termArray = temp;
    }

    termArray[terms].coef = coef;
    termArray[terms++].exp = exp;
}
```

# Polynomial Addition : Analysis

```
Polynomial c;  
int aPos = 0, bPos = 0;
```

$$a(x) = 3x^3 + 2x^2 - 4x$$

coef	3	2	-4
exp	3	2	1

↑  
aPos

$$b(x) = x^8 - 10x^5 - 3x^3 + 1$$

coef	1	-10	-3	1
exp	8	5	3	0

↑  
bPos

$$c(x) =$$

# Polynomial Addition : Analysis

```
while ((aPos < terms) && (bPos < b.terms)) {  
    if (termArray[aPos].exp == b.termArray[bPos].exp) {  
        float t = termArray[aPos].coef + b.termArray[bPos].coef;  
        if (t) c.NewTerm(t, termArray[aPos].exp);  
        aPos++; bPos++;  
    }  
    else if (termArray[aPos].exp < b.termArray[bPos].exp) {  
        c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);  
        bPos++;  
    }  
    else {  
        c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);  
        aPos++;  
    }  
}
```

a.termArray[aPos].exp = a.termArray[0].exp = 3  
b.termArray[bPos].exp = b.termArray[0].exp = 8  
∴ a.termArray[aPos].exp < b.termArray[bPos].exp

$$a(x) = 3x^3 + 2x^2 - 4x$$

coef	3	2	-4
exp	3	2	1

↑  
aPos

$$b(x) = x^8 - 10x^5 - 3x^3 + 1$$

coef	1	-10	-3	1
exp	8	5	3	0

↑  
bPos

$$c(x) = x^8$$

coef	1
exp	8

# Polynomial Addition : Analysis

```

while ((aPos < terms) && (bPos < b.terms)) {
    if (termArray[aPos].exp == b.termArray[bPos].exp) {
        float t = termArray[aPos].coef + b.termArray[bPos].coef;
        if (t) c.NewTerm(t, termArray[aPos].exp);
        aPos++; bPos++;
    }
    else if (termArray[aPos].exp < b.termArray[bPos].exp) {
        c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
        bPos++;
    }
    else {
        c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);
        aPos++;
    }
}

```

$a.\text{termArray}[aPos].\text{exp} = a.\text{termArray}[0].\text{exp} = 3$   
 $b.\text{termArray}[bPos].\text{exp} = b.\text{termArray}[1].\text{exp} = 5$   
 $\therefore a.\text{termArray}[aPos].\text{exp} < b.\text{termArray}[bPos].\text{exp}$

$$a(x) = 3x^3 + 2x^2 - 4x$$

coef	3	2	-4
exp	3	2	1

↑  
aPos

$$b(x) = x^8 - 10x^5 - 3x^3 + 1$$

coef	1	-10	-3	1
exp	8	5	3	0

↑  
bPos

$$c(x) = x^8 - 10x^5$$

coef	1	-10
exp	8	5

# Polynomial Addition : Analysis

```
while ((aPos < terms) && (bPos < b.terms)) {
    if (termArray[aPos].exp == b.termArray[bPos].exp) {
        float t = termArray[aPos].coef + b.termArray[bPos].coef;
        if (t) c.NewTerm(t, termArray[aPos].exp);
        aPos++; bPos++;      T = 3 - 3 = 0이므로 새 항은 만들어지지 않음
    }
    else if (termArray[aPos].exp < b.termArray[bPos].exp) {
        c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
        bPos++;
    }
    else {
        c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);
        aPos++;
    }
}
```

$a.\text{termArray}[aPos].\text{exp} = a.\text{termArray}[0].\text{exp} = 3$   
 $b.\text{termArray}[bPos].\text{exp} = b.\text{termArray}[2].\text{exp} = 3$   
 $\therefore a.\text{termArray}[aPos].\text{exp} == b.\text{termArray}[bPos].\text{exp}$

$$a(x) = 3x^3 + 2x^2 - 4x$$

coef	3	2	-4
exp	3	2	1

↑  
aPos

$$b(x) = x^8 - 10x^5 - 3x^3 + 1$$

coef	1	-10	-3	1
exp	8	5	3	0

↑  
bPos

$$c(x) = x^8 - 10x^5$$

coef	1	-10
exp	8	5

# Polynomial Addition : Analysis

```
while ((aPos < terms) && (bPos < b.terms)) {  
    if (termArray[aPos].exp == b.termArray[bPos].exp) {  
        float t = termArray[aPos].coef + b.termArray[bPos].coef;  
        if (t) c.NewTerm(t, termArray[aPos].exp);  
        aPos++; bPos++;  
    }  
    else if (termArray[aPos].exp < b.termArray[bPos].exp) {  
        c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);  
        bPos++;  
    }  
    else {  
        c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);  
        aPos++;  
    }  
}
```

a.termArray[aPos].exp = a.termArray[1].exp = 2  
b.termArray[bPos].exp = b.termArray[3].exp = 0  
 $\therefore$  a.termArray[aPos].exp > b.termArray[bPos].exp

$$a(x) = 3x^3 + 2x^2 - 4x$$

coef	3	2	-4
exp	3	2	1

↑  
aPos

$$b(x) = x^8 - 10x^5 - 3x^3 + 1$$

coef	1	-10	-3	1
exp	8	5	3	0

↑  
bPos

$$c(x) = x^8 - 10x^5 + 2x^2$$

coef	1	-10	2
exp	8	5	2

# Polynomial Addition : Analysis

```
while ((aPos < terms) && (bPos < b.terms)) {  
    if (termArray[aPos].exp == b.termArray[bPos].exp) {  
        float t = termArray[aPos].coef + b.termArray[bPos].coef;  
        if (t) c.NewTerm(t, termArray[aPos].exp);  
        aPos++; bPos++;  
    }  
    else if (termArray[aPos].exp < b.termArray[bPos].exp) {  
        c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);  
        bPos++;  
    }  
    else {  
        c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);  
        aPos++;  
    }  
}
```

a.termArray[aPos].exp = a.termArray[2].exp = 1  
b.termArray[bPos].exp = b.termArray[3].exp = 0  
∴ a.termArray[aPos].exp > b.termArray[bPos].exp

$$a(x) = 3x^3 + 2x^2 - 4x$$

coef	3	2	-4	
exp	3	2	1	

↑  
aPos

$$b(x) = x^8 - 10x^5 - 3x^3 + 1$$

coef	1	-10	-3	1
exp	8	5	3	0

↑  
bPos

$$c(x) = x^8 - 10x^5 + 2x^2 - 4x$$

coef	1	-10	2	-4
exp	8	5	2	1



# Polynomial Addition : Analysis

```
// Add rest terms of *this
for (; aPos < terms; aPos++)
    c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);
// Add rest terms of b(x)
for (; bPos < b.terms; bPos++)
    c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
```

$$a(x) = 3x^3 + 2x^2 - 4x$$

coef	3	2	-4	
exp	3	2	1	

↑  
aPos

$$b(x) = x^8 - 10x^5 - 3x^3 + 1$$

coef	1	-10	-3	1	
exp	8	5	3	0	

↑  
bPos

$$c(x) = x^8 - 10x^5 + 2x^2 - 4x + 1$$

coef	1	-10	2	-4	1
exp	8	5	2	1	0

# Polynomial Addition : Analysis

- add 함수의 분석 ( $m : a(x)$  항의 수,  $n : b(x)$  항의 수)
  - while 문을 반복할 때마다 aPos나 bPos 또는 둘 다 값이 1씩 증가함
  - while 문은 aPos가 a.terms와 같거나 bPos가 b.terms와 같을 때 종료됨
  - 따라서 반복문의 반복 횟수는  $m + n - 1$ 이 됨
  - $a(x) = \sum_{i=0}^n x^{2i}$ ,  $b(x) = \sum_{i=0}^n x^{2i+1}$ 인 경우, 가장 오랜 시간이 걸림 (Worst-Case)
  - 이 때, 시간 복잡도는  $O(m + n)$
  - 2개의 for 문도 전체적으로  $O(m + n)$ 의 시간이 걸림
  - 따라서 이 알고리즘의 점근적 연산 시간은  $O(m + n + \text{배열 확장에 걸리는 시간})$

# Sparse Matrix

# What is matrix?

- 행렬은 숫자가  $m$ 개의 행과  $n$ 개의 열로 구성됨

$$\cdot \begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \end{bmatrix} \quad \begin{bmatrix} 0 & 2 & 0 \\ 0 & 0 & 3 \\ 0 & 0 & 0 \end{bmatrix}$$

각각 3개의 행과 3개의 열을 가지는 행렬임

- 이와 같이  $m$ 과  $n$ 이 같은 행렬을 정방 행렬(Square Matrix)이라고 함
- 행렬은 보통 2차원 배열, 예를 들어  $a[m][n]$ 으로 저장
- 첫 번째 행렬과 달리 두 번째 행렬은 많은 항들이 0임  
이러한 행렬을 희소 행렬(Sparse Matrix)라고 함
- 만약  $5000 \times 5000$  행렬에서 0이 아닌 원소가 5000개 뿐인 경우, 0이 아닌 원소만 저장하는 다른 표현을 사용한다면 상당한 시간과 공간을 절약할 수 있음

# Sparse Matrix ADT

```
// Three elements: <row, column, value>
// row, column: int (not negative)
// <row, column> is a unique pair
class SparseMatrix
{
public:
    // Create SparseMatrix with r rows, c columns, t terms (not 0)
    SparseMatrix(int r, int c, int t);
    // Transpose all elements of *this and return it
    SparseMatrix Transpose();
    // If the dimension of *this equals to b, add/subtract *this and b and return result
    // Otherwise, throw exception
    SparseMatrix Add(const SparseMatrix& b);
    SparseMatrix Subtract(const SparseMatrix& b);
    // If the column of *this equals to the row of b,
    // multiply *this and b and return result
    // Otherwise, throw exception
    SparseMatrix Multiply(const SparseMatrix& b);
};
```

# Sparse Matrix Representation

```
class SparseMatrix; // Forward Declaration
```

```
class MatrixTerm
{
    friend class SparseMatrix;
```

```
private:
    int row, col, value;
};
```

```
class SparseMatrix
{
private:
    int rows, cols, terms, capacity;
    MatrixTerm* smArray;
    ...
};
```

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28

# Matrix : Add / Subtract / Multiply

- 행렬의 덧셈, 뺄셈 : 두 행렬의 행과 열의 크기가 같다면, 두 행렬을 서로 더하거나 뺄 수 있음
  - $\begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix} + \begin{bmatrix} 2 & 0 \\ 0 & -4 \end{bmatrix} = \begin{bmatrix} 5 & 0 \\ 0 & -2 \end{bmatrix}, \begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix} - \begin{bmatrix} 2 & 0 \\ 0 & -4 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 6 \end{bmatrix}$
- 행렬의 곱셈 : A가  $a \times b$  행렬이고 B가  $c \times d$  행렬이라면,  $b = c$ 라면 두 행렬을 곱할 수 있고 결과는  $a \times d$  행렬이 됨  
AB의  $i$ 행  $j$ 열의 성분은 (A의  $i$ 행의 성분)  $\times$  (B의  $j$ 열의 성분) 합과 같음
  - $\begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix} \times \begin{bmatrix} 2 & 0 \\ 0 & -4 \end{bmatrix} = \begin{bmatrix} 3 \times 2 + 0 \times 0 & 3 \times 0 + 0 \times (-4) \\ 0 \times 2 + 2 \times 0 & 0 \times 0 + 2 \times (-4) \end{bmatrix} = \begin{bmatrix} 6 & 0 \\ 0 & -8 \end{bmatrix}$

# Matrix : Transpose

- 행렬의 전치 : 원래 행렬에서 열은 행으로, 행은 열로 바꾼 행렬

- $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$

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



# Matrix Transpose

```
SparseMatrix SparseMatrix::Transpose()
{ // Return the transpose matrix of *this
    SparseMatrix b(cols, rows, terms); // the size of b.smArray is terms
    if (terms > 0)
    { // This is not 0-matrix
        int currentB = 0;
        for (int c = 0; c < cols; c++)
        { // Transpose each column
            for (int i = 0; i < terms; i++)
            { // Find element from column c and move it
                if (smArray[i].col == c)
                {
                    b.smArray[currentB].row = c;
                    b.smArray[currentB].col = smArray[i].row;
                    b.smArray[currentB++].value = smArray[i].value;
                }
            }
        }
    }
    // End of if (terms > 0)
    return b;
}
```

# Matrix Transpose : Analysis

- Transpose 함수의 분석 (cols : 열 수, rows : 행 수, terms : 원소 수)
  - 첫 번째 for 문은 cols번 수행
  - 두 번째 for 문은 terms번 수행
  - 따라서 Transpose 함수의 총 실행 시간은  $O(\text{terms} \cdot \text{cols})$
  - 원소 수가  $\text{rows} \cdot \text{cols}$ 개 있는 경우, 가장 오랜 시간이 걸림 (Worst-Case)
  - 이 때, 시간 복잡도는  $O(\text{rows} \cdot \text{cols}^2)$
  - 공간을 절약을 위해 시간을 희생한 결과
  - 메모리를 조금 더 사용해서 개선된 알고리즘을 만들 수는 없을까?  
 $O(\text{terms} + \text{cols})$ 에 가능!

# Faster Matrix Transpose

```
SparseMatrix SparseMatrix::FastTranspose()
{ // Return the transpose matrix of *this in O(terms + cols)
  SparseMatrix b(cols, rows, terms);
  if (terms > 0)
  { // This is not 0-matrix
    int* rowSize = new int[cols];
    int* rowStart = new int[cols];

    std::fill(rowSize, rowSize + cols, 0); // Initialization
    for (int i = 0; i < terms; i++) rowSize[smArray[i].col]++;

    // Start point of row i of rowStart[i] = b
    rowStart[0] = 0;
    for (int i = 1; i < cols; i++)
      rowStart[i] = rowStart[i - 1] + rowSize[i - 1];
```

# Faster Matrix Transpose

```
    for (int i = 0; i < terms; i++)
    { // Copy *this to b
        int j = rowStart[smArray[i].col];
        b.smArray[j].row = smArray[i].col;
        b.smArray[j].col = smArray[i].row;
        b.smArray[j].value = smArray[i].value;
        rowStart[smArray[i].col]++;
    } // End of for

    delete[] rowSize;
    delete[] rowStart;
} // End of if

return b;
}
```

# Faster Matrix Transpose : Analysis

- rowSize = 전치된 후 각 행에 존재하는 원소의 수
- rowStart = 전치된 후 각 행에 다음 원소가 삽입될 위치

```
int* rowSize = new int[cols];  
int* rowStart = new int[cols];
```

```
std::fill(rowSize, rowSize + cols, 0); // Initialization  
for (int i = 0; i < terms; i++) rowSize[smArray[i].col]++;
```

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	0	0	0	0	0	0
rowStart =						

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28

# Faster Matrix Transpose : Analysis

- rowSize = 전치된 후 각 행에 존재하는 원소의 수
- rowStart = 전치된 후 각 행에 다음 원소가 삽입될 위치

```
int* rowSize = new int[cols];  
int* rowStart = new int[cols];
```

```
std::fill(rowSize, rowSize + cols, 0); // Initialization  
for (int i = 0; i < terms; i++) rowSize[smArray[i].col]++;
```

$i = 0 \rightarrow \text{smArray}[0].\text{col} = 0 \rightarrow \text{rowSize}[0]++;$

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	1	0	0	0	0	0
rowStart =						

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28

# Faster Matrix Transpose : Analysis

- rowSize = 전치된 후 각 행에 존재하는 원소의 수
- rowStart = 전치된 후 각 행에 다음 원소가 삽입될 위치

```
int* rowSize = new int[cols];  
int* rowStart = new int[cols];
```

```
std::fill(rowSize, rowSize + cols, 0); // Initialization  
for (int i = 0; i < terms; i++) rowSize[smArray[i].col]++;
```

i = 1 → smArray[1].col = 3 → rowSize[3]++;

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	1	0	0	1	0	0
rowStart =						

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28

# Faster Matrix Transpose : Analysis

- rowSize = 전치된 후 각 행에 존재하는 원소의 수
- rowStart = 전치된 후 각 행에 다음 원소가 삽입될 위치

```
int* rowSize = new int[cols];  
int* rowStart = new int[cols];
```

```
std::fill(rowSize, rowSize + cols, 0); // Initialization  
for (int i = 0; i < terms; i++) rowSize[smArray[i].col]++;
```

i = 2 → smArray[2].col = 5 → rowSize[5]++;

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	1	0	0	1	0	1
rowStart =						

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28



# Faster Matrix Transpose : Analysis

- rowSize = 전치된 후 각 행에 존재하는 원소의 수
- rowStart = 전치된 후 각 행에 다음 원소가 삽입될 위치

```
int* rowSize = new int[cols];  
int* rowStart = new int[cols];
```

```
std::fill(rowSize, rowSize + cols, 0); // Initialization  
for (int i = 0; i < terms; i++) rowSize[smArray[i].col]++;
```

i = 3 → smArray[3].col = 1 → rowSize[1]++;

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	1	1	0	1	0	1
rowStart =						

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28

# Faster Matrix Transpose : Analysis

- rowSize = 전치된 후 각 행에 존재하는 원소의 수
- rowStart = 전치된 후 각 행에 다음 원소가 삽입될 위치

```
int* rowSize = new int[cols];  
int* rowStart = new int[cols];
```

```
std::fill(rowSize, rowSize + cols, 0); // Initialization  
for (int i = 0; i < terms; i++) rowSize[smArray[i].col]++;
```

i = 4 → smArray[4].col = 2 → rowSize[2]++;

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	1	1	1	1	0	1
rowStart =						

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28

# Faster Matrix Transpose : Analysis

- rowSize = 전치된 후 각 행에 존재하는 원소의 수
- rowStart = 전치된 후 각 행에 다음 원소가 삽입될 위치

```
int* rowSize = new int[cols];  
int* rowStart = new int[cols];
```

```
std::fill(rowSize, rowSize + cols, 0); // Initialization  
for (int i = 0; i < terms; i++) rowSize[smArray[i].col]++;
```

i = 5 → smArray[5].col = 3 → rowSize[3]++;

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	1	1	1	2	0	1
rowStart =						

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28

# Faster Matrix Transpose : Analysis

- rowSize = 전치된 후 각 행에 존재하는 원소의 수
- rowStart = 전치된 후 각 행에 다음 원소가 삽입될 위치

```
int* rowSize = new int[cols];  
int* rowStart = new int[cols];
```

```
std::fill(rowSize, rowSize + cols, 0); // Initialization  
for (int i = 0; i < terms; i++) rowSize[smArray[i].col]++;
```

i = 6 → smArray[6].col = 0 → rowSize[0]++;

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	2	1	1	2	0	1
rowStart =						

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28

# Faster Matrix Transpose : Analysis

- rowSize = 전치된 후 각 행에 존재하는 원소의 수
- rowStart = 전치된 후 각 행에 다음 원소가 삽입될 위치

```
int* rowSize = new int[cols];  
int* rowStart = new int[cols];
```

```
std::fill(rowSize, rowSize + cols, 0); // Initialization  
for (int i = 0; i < terms; i++) rowSize[smArray[i].col]++;
```

i = 7 → smArray[7].col = 2 → rowSize[2]++;

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	2	1	2	2	0	1
rowStart =						

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28

# Faster Matrix Transpose : Analysis

- rowSize = 전치된 후 각 행에 존재하는 원소의 수
- rowStart = 전치된 후 각 행에 다음 원소가 삽입될 위치

```
// Start point of row i of rowStart[i] = b
rowStart[0] = 0;
for (int i = 1; i < cols; i++)
    rowStart[i] = rowStart[i - 1] + rowSize[i - 1];
```

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	2	1	2	2	0	1
rowStart =	0					

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28

# Faster Matrix Transpose : Analysis

- rowSize = 전치된 후 각 행에 존재하는 원소의 수
- rowStart = 전치된 후 각 행에 다음 원소가 삽입될 위치

```
// Start point of row i of rowStart[i] = b
rowStart[0] = 0;
for (int i = 1; i < cols; i++)
    rowStart[i] = rowStart[i - 1] + rowSize[i - 1];
```

i = 1 → rowStart[1] = rowStart[0] + rowSize[0];

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	2	1	2	2	0	1
rowStart =	0	2				

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28

# Faster Matrix Transpose : Analysis

- rowSize = 전치된 후 각 행에 존재하는 원소의 수
- rowStart = 전치된 후 각 행에 다음 원소가 삽입될 위치

```
// Start point of row i of rowStart[i] = b
rowStart[0] = 0;
for (int i = 1; i < cols; i++)
    rowStart[i] = rowStart[i - 1] + rowSize[i - 1];
```

$i = 2 \rightarrow \text{rowStart}[2] = \text{rowStart}[1] + \text{rowSize}[1];$

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	2	1	2	2	0	1
rowStart =	0	2	3			

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28



# Faster Matrix Transpose : Analysis

- rowSize = 전치된 후 각 행에 존재하는 원소의 수
- rowStart = 전치된 후 각 행에 다음 원소가 삽입될 위치

```
// Start point of row i of rowStart[i] = b
rowStart[0] = 0;
for (int i = 1; i < cols; i++)
    rowStart[i] = rowStart[i - 1] + rowSize[i - 1];
```

$i = 3 \rightarrow \text{rowStart}[3] = \text{rowStart}[2] + \text{rowSize}[2];$

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	2	1	2	2	0	1
rowStart =	0	2	3	5		

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28

# Faster Matrix Transpose : Analysis

- rowSize = 전치된 후 각 행에 존재하는 원소의 수
- rowStart = 전치된 후 각 행에 다음 원소가 삽입될 위치

```
// Start point of row i of rowStart[i] = b
rowStart[0] = 0;
for (int i = 1; i < cols; i++)
    rowStart[i] = rowStart[i - 1] + rowSize[i - 1];
```

$i = 4 \rightarrow \text{rowStart}[4] = \text{rowStart}[3] + \text{rowSize}[3];$

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	2	1	2	2	0	1
rowStart =	0	2	3	5	7	

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28

# Faster Matrix Transpose : Analysis

- rowSize = 전치된 후 각 행에 존재하는 원소의 수
- rowStart = 전치된 후 각 행에 다음 원소가 삽입될 위치

```
// Start point of row i of rowStart[i] = b
rowStart[0] = 0;
for (int i = 1; i < cols; i++)
    rowStart[i] = rowStart[i - 1] + rowSize[i - 1];
```

$i = 5 \rightarrow \text{rowStart}[5] = \text{rowStart}[4] + \text{rowSize}[4];$

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	2	1	2	2	0	1
rowStart =	0	2	3	5	7	7

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28

# Faster Matrix Transpose : Analysis

```
for (int i = 0; i < terms; i++)  
{ // Copy *this to b  
    int j = rowStart[smArray[i].col];  
    b.smArray[j].row = smArray[i].col;  
    b.smArray[j].col = smArray[i].row;  
    b.smArray[j].value = smArray[i].value;  
    rowStart[smArray[i].col]++;  
} // End of for
```

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	2	1	2	2	0	1
rowStart =	1	2	3	5	7	7

$i = 0 \rightarrow \text{smArray}[0].\text{col} = 0 \rightarrow j = \text{rowStart}[0] = 0$   
 $\text{rowStart}[0]++;$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28



	row	col	value
smArray[0]	0	0	15
smArray[1]			
smArray[2]			
smArray[3]			
smArray[4]			
smArray[5]			
smArray[6]			
smArray[7]			

# Faster Matrix Transpose : Analysis

```
for (int i = 0; i < terms; i++)  
{ // Copy *this to b  
    int j = rowStart[smArray[i].col];  
    b.smArray[j].row = smArray[i].col;  
    b.smArray[j].col = smArray[i].row;  
    b.smArray[j].value = smArray[i].value;  
    rowStart[smArray[i].col]++;  
} // End of for
```

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	2	1	2	2	0	1
rowStart =	1	2	3	6	7	7

$i = 1 \rightarrow \text{smArray}[1].\text{col} = 3 \rightarrow j = \text{rowStart}[3] = 5$   
 $\text{rowStart}[3]++;$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28



	row	col	value
smArray[0]	0	0	15
smArray[1]			
smArray[2]			
smArray[3]			
smArray[4]			
smArray[5]	3	0	22
smArray[6]			
smArray[7]			

# Faster Matrix Transpose : Analysis

```
for (int i = 0; i < terms; i++)  
{ // Copy *this to b  
    int j = rowStart[smArray[i].col];  
    b.smArray[j].row = smArray[i].col;  
    b.smArray[j].col = smArray[i].row;  
    b.smArray[j].value = smArray[i].value;  
    rowStart[smArray[i].col]++;  
} // End of for
```

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	2	1	2	2	0	1
rowStart =	1	2	3	6	7	8

$i = 2 \rightarrow \text{smArray}[2].\text{col} = 5 \rightarrow j = \text{rowStart}[5] = 7$   
 $\text{rowStart}[5]++;$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28



	row	col	value
smArray[0]	0	0	15
smArray[1]			
smArray[2]			
smArray[3]			
smArray[4]			
smArray[5]	3	0	22
smArray[6]			
smArray[7]	5	0	-15

# Faster Matrix Transpose : Analysis

```
for (int i = 0; i < terms; i++)
{ // Copy *this to b
    int j = rowStart[smArray[i].col];
    b.smArray[j].row = smArray[i].col;
    b.smArray[j].col = smArray[i].row;
    b.smArray[j].value = smArray[i].value;
    rowStart[smArray[i].col]++;
} // End of for
```

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	2	1	2	2	0	1
rowStart =	1	3	3	6	7	8

$i = 3 \rightarrow \text{smArray}[3].\text{col} = 1 \rightarrow j = \text{rowStart}[1] = 2$   
 $\text{rowStart}[1]++;$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28



	row	col	value
smArray[0]	0	0	15
smArray[1]			
smArray[2]	1	1	11
smArray[3]			
smArray[4]			
smArray[5]	3	0	22
smArray[6]			
smArray[7]	5	0	-15

# Faster Matrix Transpose : Analysis

```
for (int i = 0; i < terms; i++)
{ // Copy *this to b
    int j = rowStart[smArray[i].col];
    b.smArray[j].row = smArray[i].col;
    b.smArray[j].col = smArray[i].row;
    b.smArray[j].value = smArray[i].value;
    rowStart[smArray[i].col]++;
} // End of for
```

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	2	1	2	2	0	1
rowStart =	1	3	4	6	7	8

$i = 4 \rightarrow \text{smArray}[4].\text{col} = 2 \rightarrow j = \text{rowStart}[2] = 3$   
 $\text{rowStart}[2]++;$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28



	row	col	value
smArray[0]	0	0	15
smArray[1]			
smArray[2]	1	1	11
smArray[3]	2	1	3
smArray[4]			
smArray[5]	3	0	22
smArray[6]			
smArray[7]	5	0	-15



# Faster Matrix Transpose : Analysis

```
for (int i = 0; i < terms; i++)  
{ // Copy *this to b  
    int j = rowStart[smArray[i].col];  
    b.smArray[j].row = smArray[i].col;  
    b.smArray[j].col = smArray[i].row;  
    b.smArray[j].value = smArray[i].value;  
    rowStart[smArray[i].col]++;  
} // End of for
```

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	2	1	2	2	0	1
rowStart =	1	3	4	7	7	8

$i = 5 \rightarrow \text{smArray}[5].\text{col} = 3 \rightarrow j = \text{rowStart}[3] = 6$   
 $\text{rowStart}[3]++;$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28



	row	col	value
smArray[0]	0	0	15
smArray[1]			
smArray[2]	1	1	11
smArray[3]	2	1	3
smArray[4]			
smArray[5]	3	0	22
smArray[6]	3	2	-6
smArray[7]	5	0	-15

# Faster Matrix Transpose : Analysis

```
for (int i = 0; i < terms; i++)
{ // Copy *this to b
    int j = rowStart[smArray[i].col];
    b.smArray[j].row = smArray[i].col;
    b.smArray[j].col = smArray[i].row;
    b.smArray[j].value = smArray[i].value;
    rowStart[smArray[i].col]++;
} // End of for
```

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	2	1	2	2	0	1
rowStart =	2	3	4	7	7	8

$i = 6 \rightarrow \text{smArray}[6].\text{col} = 0 \rightarrow j = \text{rowStart}[0] = 1$   
 $\text{rowStart}[0]++;$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28



	row	col	value
smArray[0]	0	0	15
smArray[1]	0	4	91
smArray[2]	1	1	11
smArray[3]	2	1	3
smArray[4]			
smArray[5]	3	0	22
smArray[6]	3	2	-6
smArray[7]	5	0	-15

# Faster Matrix Transpose : Analysis

```
for (int i = 0; i < terms; i++)  
{ // Copy *this to b  
    int j = rowStart[smArray[i].col];  
    b.smArray[j].row = smArray[i].col;  
    b.smArray[j].col = smArray[i].row;  
    b.smArray[j].value = smArray[i].value;  
    rowStart[smArray[i].col]++;  
} // End of for
```

	[0]	[1]	[2]	[3]	[4]	[5]
rowSize =	2	1	2	2	0	1
rowStart =	2	3	5	7	7	8

$i = 7 \rightarrow \text{smArray}[7].\text{col} = 2 \rightarrow j = \text{rowStart}[2] = 4$   
 $\text{rowStart}[2]++;$

	row	col	value
smArray[0]	0	0	15
smArray[1]	0	3	22
smArray[2]	0	5	-15
smArray[3]	1	1	11
smArray[4]	1	2	3
smArray[5]	2	3	-6
smArray[6]	4	0	91
smArray[7]	5	2	28



	row	col	value
smArray[0]	0	0	15
smArray[1]	0	4	91
smArray[2]	1	1	11
smArray[3]	2	1	3
smArray[4]	2	5	28
smArray[5]	3	0	22
smArray[6]	3	2	-6
smArray[7]	5	0	-15

# Faster Matrix Transpose : Analysis

- FastTranspose 함수의 분석 (cols : 열 수, rows : 행 수, terms : 원소 수)
  - terms, cols - 1, terms번 실행되는 3개의 반복문이 있음
  - 각 반복문을 한 번 반복하는 데 상수 시간이 걸리므로, 3개의 반복문 전체에 대한 시간은  $O(cols + terms)$
  - 원소 수가  $rows \cdot cols$ 개 있는 경우, 가장 오랜 시간이 걸림 (Worst-Case)
  - 이 때, 시간 복잡도는  $O(rows \cdot cols)$

# String Pattern Match: KMP Algorithm

# String ADT

```
class String {
public:
    // Constructor with content *init, length m
    String(char* init, int m);

    // Return true if *this's string is equal to t's. Otherwise, return false
    bool operator==(String t);
    // Return true if *this's string is empty. Otherwise, return false
    bool operator!();

    // Return the number of *this's characters
    int Length();
    // Return *this's string + t's
    String Concat(String t);
    // Return substring with index (i, i+1, ..., i+j-1) of *this's string
    String Substr(int i, int j);

    // Return index i if *this's substring matches pat's string
    // Return -1 if pat is empty or not *this's substring
    int Find(String pat);
};
```

# String Pattern Match

- 두 문자열 `s`와 `pat`가 주어지고, `pat`가 `s`에서 탐색할 패턴이라고 가정
- `s`에 `pat`가 있는지를 알아내기 위해 문자열 패턴 매칭 함수를 사용
- 문자열 패턴 매칭 함수를 호출하면 `pat`와 `i`번째 위치에서 시작하는 `s`의 문자열 일부가 매칭될 때, 위치 `i`를 반환
- `pat`가 공백이거나 `s`의 문자열 일부가 아닌 경우에는 `-1`을 반환

# Simple Algorithm

```
int String::Find(String pat)
{ // If pat is found in *this, then return start position of pat in *this
  // Otherwise, return -1
  for (int start = 0; start <= Length() - pat.Length(); start++)
  { // Check pattern match at str[start]
    int j;
    for (j = 0; j < pat.Length() && str[start + j] == pat.str[j]; j++)
      if (j == pat.Length()) return start; // Found match
  }

  return -1; // pat is empty or not exist in *this
}
```



# Simple Algorithm : Analysis

- 가장 간단하지만 효율적이지 못한 방법 :  
s의 각 위치를 순차적으로 조사해 그 위치가 매칭의 시작점인지를 결정함
- lengthP : 패턴 pat의 길이, lengthS : 문자열 s의 길이
- s에서 위치 lengthS - lengthP 보다 오른쪽에 있는 경우에는 pat와 매치가 될 문자가 충분하지 않으므로 고려하지 않음
- Find 함수의 시간 복잡도는  $O(\text{lengthP} \cdot \text{lengthS})$

# KMP Algorithm

- 문자열 패턴 매칭을 푸는 최적의 시간 복잡도는  $O(\text{length}P + \text{length}S)$ 
  - 최악의 경우에는 패턴과 문자열의 문자들을 전부 적어도 한 번씩 검색해야 하기 때문
- 패턴에 대한 문자열의 탐색은 문자열 뒤로 이동하는 일이 없어야 함
  - 즉, 매치되지 않은 경우 매치되지 않은 패턴 내의 문자와 패턴 내의 위치 정보를 이용해 어디에서 탐색을 계속해야 할 것인가를 결정해야 함
- KMP(Knuth, Morris, Pratt)는 이러한 방식으로 동작하면서 선형 시간 복잡도를 갖는 패턴 매치 알고리즘을 개발함

# KMP Algorithm

- 예를 들어, 문자열  $pat = a b c a b c a c a b$ 이 있을 때,  
문자열  $s = s_0 s_1 \cdots s_{m-1}$ 이라 하고,  
 $s_i$ 에서부터 매치되는지 여부를 결정해야 된다고 가정
  - $s_i \neq a$ 이면,  $s_i + 1$ 과  $a$ 를 비교해야 함
  - 마찬가지로  $s_i = a$ 이고  $s_i + 1 \neq b$ 이면,  $s_i + 1$ 과  $a$ 를 비교해야 함

# KMP Algorithm

- 하지만  $s_i s_{i+1} = ab$ 이고,  $s_{i+2} \neq c$ 이면 다음과 같은 상황이 발생함

$s =$	-	$a$	$b$	$?$	$?$	$?$	.	.	.	.	$?$
$pat =$		$a$	$b$	$c$	$a$	$b$	$c$	$a$	$c$	$a$	$b$


- $s$ 의 첫 번째  $?$ 는  $s_{i+2}$ 에 해당하고  $s_{i+2} \neq c$ 라고 가정
- 이 시점에서  $pat$ 의 첫 문자와  $s_{i+2}$ 를 비교 탐색을 계속 진행
- 이미  $s_{i+1}$ 이  $pat$ 의 두 번째 문자인  $b$ 와 같으므로,  
 $s_{i+1} \neq a$ 라는 것을 알기 때문에  $pat$ 의 첫 문자와  $s_{i+1}$ 을 비교할 필요가 없음

# KMP Algorithm

- 이제  $pat$ 의 처음 네 문자가 매치되고 그 다음이 매치되지 않은 경우, 즉  $s_{i+4} \neq b$ 인 경우를 가정해 보면 다음과 같은 상황이 발생함

$s =$     -     $a$      $b$      $c$      $a$     ?    ?    .    .    .    ?  
 $pat =$      $a$      $b$      $c$      $a$      $b$      $c$      $a$      $c$      $a$      $b$

- $s_{i+4}$ 와  $pat$ 에 있는 두 번째 문자  $b$ 를 비교
- 이 곳은 패턴  $pat$ 를 오른쪽으로 이동시켜서 부분 매치가 일어나는 첫 번째 위치

$s =$     -     $a$      $b$      $c$      $a$     ?    ?    .    .    .    ?  
 $pat =$       $a$      $b$      $c$      $a$      $b$      $c$      $a$      $c$      $a$      $b$

- 패턴 내의 문자들을 알고  $s$  내의 문자와 매치되지 않는 패턴 내의 위치를 알아냄으로써  $s$  안에서 패턴 내의 어느 위치에서부터 탐색을 계속할 것인지를 결정할 수 있음

# Failure Function

- 이를 정형화하기 위해 패턴에 대한 실패 함수를 다음과 같이 정의함
  - 임의의 패턴  $p = p_0p_1 \cdots p_{n-1}$ 이 있을 때 이 패턴의 실패 함수  $f$ 는 아래와 같이 정의됨
    - $f(j) = \begin{cases} \text{제일 큰 } k(< j), \text{ 여기서 } p_0 \cdots p_k = p_{j-k} \cdots p_j \text{인 } k(\geq 0) \text{가 존재하는 경우} \\ -1, \text{ 그 외의 경우} \end{cases}$
  - 예를 들어, 패턴  $pat = abcabcacab$ 에 대해  $f$ 는 다음과 같음

$j$	0	1	2	3	4	5	6	7	8	9
$pat$	$a$	$b$	$c$	$a$	$b$	$c$	$a$	$c$	$a$	$b$
$f$	-1	-1	-1	0	1	2	3	-1	0	1
- 실패 함수의 정의로부터 패턴 매치를 위한 규칙을 구할 수 있음
  - 만약  $s_{i-j} \cdots s_{i-1} = p_0 \cdots p_{j-1}$ 이고  $s_i \neq p_j$ 인 부분 매치가 일어난다면
    - $j \neq 0$ 일 때  $s_i$ 와  $p_{f(j-1)+1}$ 을 비교해 매치를 다시할 수 있음
    - $j = 0$ 인 경우에는  $s_{i+1}$ 과  $p_0$ 를 비교해 매치를 계속할 수 있음

# KMP Algorithm : FastFind Function

```
int String::FastFind(String pat)
{ // Determine whether pat is substring of s or not
    int posP = 0, posS = 0;
    int lengthP = pat.Length(), lengthS = Length();
    while ((posP < lengthP) && (posS < lengthS)) {
        if (pat.str[posP] == pat.str[posS]) { // String match
            posP++; posS++;
        }
        else {
            if (posP == 0) posS++;
            else posP = pat.f[posP - 1] + 1;
        }
    }
    if (posP < lengthP) return -1;
    else return posS - lengthP;
}
```

# FastFind Function : Analysis

- while 문이 한 번 반복될 때마다 posS가 1씩 증가하지만 결코 감소하지 않으므로 posP는 pat에서 최대 lengthS번 이동할 수 있음
- 문자가 매치하지 않고 posP가 0이 아닐 때마다 posP는 pat의 왼쪽으로 이동하는데 최대 lengthS번 수행됨
  - 왜냐하면 그렇지 않은 경우 posP가 0보다 작아지기 때문
- 결과적으로 while 문의 최대 반복 횟수는 lengthS가 되고 FastFind 함수의 시간 복잡도는  $O(\text{lengthS})$



# Failure Function

- FastFind 함수의 분석을 통해, 실패 함수를  $O(\text{length}P)$  시간 내에 계산할 수 있다면 전체 패턴 매치 과정은 문자열과 패턴 길이의 합에 비례하는 시간 내에 이루어질 수 있음을 알 수 있음
- 하지만 실패 함수를 빨리 계산하는 방법이 있음

$$\cdot f(j) = \begin{cases} -1, j = 0 \text{ 일 경우} \\ f^m(j-1) + 1, \text{ 이 때 } p_{f^k(j-1)+1} = p_j \text{ 를 만족시키는 } k \text{ 중 가장 작은 정수} \\ -1, \text{ 위 식을 만족시키는 } k \text{ 가 없을 경우} \end{cases}$$

# KMP Algorithm : Failure Function

```
void String::FailureFunction()
{ // Calculate failure function about *this pattern
    int lengthP = Length();

    f[0] = -1;
    for (int j = 1; j < lengthP; j++)
    {
        int i = f[j - 1];
        while ((*str + j) != *(str + i + 1) && (i >= 0)) i = f[i];
        if ((*str + j) == *(str + i + 1)) f[j] = i + 1;
        else f[j] = -1;
    }
}
```

# Failure Function : Analysis

- while 문을 반복할 때마다  $i$ 의 값은 ( $f$ 의 정의에 의해) 감소되며, 변수  $i$ 는 for 문을 반복하는 시작 위치에서 재설정됨
- 그러나  $i$ 는  $-1$  ( $j = 1$ 이거나 else 문을 수행할 때), 또는 이전 반복의 마지막 값보다 1이 더 큰 값으로 재설정됨 (if 문을 수행할 때)
- $\text{int } i = f[j - 1];$ 는  $\text{lengthP} - 1$ 번만 반복되기 때문에,  $i$ 의 값은 최대  $\text{lengthP} - 1$ 번 증가함
- 결과적으로 while 문은 전체 알고리즘을 통해 최대  $\text{lengthP} - 1$ 번 반복되고 FailureFunction의 시간 복잡도는  $O(\text{lengthP})$

# KMP Algorithm : Analysis

- 실패 함수를 미리 알 수 없는 경우에 함수 FailureFunction을 이용해 실패 함수를 먼저 계산한 다음 함수 FastFind를 이용해 패턴 매치를 수행할 경우 시간 복잡도는  $O(\text{length}P + \text{length}S)$