EECS 204002
Data Structures 資料結構
Prof. REN-SONG TSAY 蔡仁松 教授
NTHU

CH. 2
ARRAYS

2.2

Array Abstract Data Type

2.2

Definition of Array

- A data structure that represents an ***ordered*** or ***linear list***.
- Elements in an array could be the same or different data types.
  - Days of the week:
    - {Sunday, Monday, …, Saturday}
  - Deck of cards:
    - {Ace, 2, 3, …, King}
  - Phone Book:
    - {(James, #1), (Claire, #2), …, (Tony, #n)}

Common Array Operations

- ADT array[n]=$\{a_0, a_1, \ldots, a_{n-1}\}$
  1. Find the length, n, of the array.
  2. Read the array from left to right (or reverse).
  3. Retrieve the i^{th} element, $0 \leq i < n$.
  4. Store a new element into i^{th} position , $0 \leq i < n$.
  5. Insert/delete the element at position i , $0 \leq i < n$.
- It is not necessary to include all operations
- Different representations carry out different subset of operations efficiently.

Array Representations

- **Sequential mapping**
  - Element a_i is stored in the location i of the array
  - The most commonly used
  - Efficient random access (operation 1,2,3)
- **Non-sequential mapping**
  - Perform insertion and deletion efficiently
  - E.g. Linked Lists in chapter 4

2.3

Polynomial

2.3 Polynomial

- $p(x) = a_0 x^{e_0} + a_1 x^{e_1} + \cdots + a_n x^{e_n} = \sum a_i x^{e_i}$
- Each $a_i x^{e_i}$ is called a **term** with coefficient a_i
  - The **degree** of p(x) is the largest exponent from among the non-zero terms.
  - Ex. $p(x) = x^5 + 4x^3 + 2x^2 + 1$
    Has 4 terms with coefficients 1, 4 ,2 and 1.
    The degree of p(x) is 5
- Array representation
  - Store (a_i, e_i) as (array[n-i], i) pair and n is the degree

x^5		$4x^3$	$2x^2$		x^0
1	0	4	2	0	1
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

Polynomial Operation

- If $a(x) = \sum a_i x^i$ and $b(x) = \sum b_i x^i$
- Polynomial addition
  - $a(x) + b(x) = \sum (a_i + b_i) x^i$

  Ex. $a(x) = x^5 + 4x^3 + 2x^2 + 1$ (degree = 5)
  $b(x) = 3x^6 + 4x^3 + x$ (degree = 6)
  $a(x) + b(x) = 3x^6 + x^5 + 8x^3 + 2x^2 + x + 1$ (degree = 6)
- Polynomial multiplication
  - $a(x) \cdot b(x) = \sum (a_i x^i \cdot \sum (b_j x^j))$

Polynomial :ADT

```
class Polynomial {
public:
    // Construct p(x) = 0
    Polynomial(void);
    // Destructor
    ~Polynomial(void);
    // Return the sum of *this and poly
    Polynomial Add(Polynomial poly);
    // Return multiplication of *this and poly
    Polynomial Mult(Polynomial poly);
    // Return the evaluation result
    float Eval(float x );
private:
    // Array representation
     …
};
```

We will ignore destructor in the codes hereafter. It is programmer's responsibility to treat her memory well ☺

Polynomial: 1st Representation

```
// in class Polynomial
public: // for convenience…
    // degree ≤ MaxDegree
    int degree;
    // coefficient array
    float coef[MaxDegree+1];
```

```
Usage:
    Polynomial a;
     a.degree = n;
     a.coef[i] = a_{n-i}
```

- Coefficients are stored in order of decreasing exponents
- Advantages:
  - Simple algorithm of operations
- Disadvantages:
  - Waste memory in a sparse polynomial

Polynomial: 2^{nd} Representation

```
class Term {
 friend Polynomial;
 float coef;
 int exp;
};
```

```
// in class Polynomial
private:
  // array of nonzero terms
  Term* termArray;
  int capacity; // size of termArray
  int terms; // number of nonzero terms
```

- Store only nonzero terms.
- Each nonzero term holds an exponent and its corresponding coefficient.
- If polynomial is sparse, 2^{nd} representation is better. If polynomial is full, 2^{nd} one has double size of 1^{st}.

2.3.2

Polynomial Addition: Code

```
Polynomial Polynomial::Add(Polynomial b)
{ // Return sum of polynomial *this and 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++;
    }
  // add in remaining terms of *this
  for(; aPos < terms; aPos++)
    c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);
  // add in remaining terms of b
  for(; bPos < b.terms; bPos++)
    c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
  return c;
}
```

Example

$$a(x) = x^5 + 9x^4 + 7x^3 + 2x$$

$$b(x) = x^6 + 3x^5 + 6x + 3$$

$$c(x) = x^6 + (1+3)x^5 + 9x^4 + 7x^3 + (2+6)x + 3$$
$$= x^6 + 4x^5 + 9x^4 + 7x^3 + 8x + 3$$

Time Complexity of Analysis

- Inside the while loop: every statement takes $O(1)$ time
- How many times the "while loop" is executed in the worst case ?
  - Let $a(x)$ have m terms, and $b(x)$ have n terms.
  - In each iteration, we access next element in $a(x)$ or $b(x)$, or both.
  - Worst case: $m + n$.
    e.g. It happens when
    $$A(x) = 7x^5 + x^3 + x; \quad B(x) = x^6 + 2x^4 + 6x^2 + 3$$
    Access remaining terms in $A(x)$: $O(m)$
    Access remaining terms in $B(x)$: $O(n)$
- Hence, total run time = $O(m + n)$

2.4

Matrix

2.4

Matrix

- Denote a matrix consists of ***m rows*** and ***n columns*** as $A_{m\star n}$ (read A is a ***m by n*** matrix).
- Usually stored as a two-dimensional array, $\boldsymbol{a}[\boldsymbol{m}][\boldsymbol{n}]$, in which element at i^th row and j^th column is accessed by $\boldsymbol{a}[\boldsymbol{i}][\boldsymbol{j}]$.
- $$A_{5\star 3} = \begin{array}{c c} \begin{matrix} \text{col 0} & \text{col 1} & \text{col 2} \end{matrix} & \\ \begin{pmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{pmatrix} & \begin{matrix} \text{row 0} \\ \text{row 1} \\ \text{row 2} \\ \text{row 3} \\ \text{row 4} \end{matrix} \end{array}$$

Matrix Operations

- Transpose
  - $C_{nxm} = A^T{}_{mxn}$
  - $c[i][j] = a[j][i]$
- Addition
  - $C_{mxn} = A_{mxn} + B_{mxn}$
  - $c[i][j] = a[i][j] + b[i][j]$
- Multiplication
  - $C_{mxp} = A_{mxn} + B_{nxp}$
  - $c[i][j] = \sum_{k=0}^{n-1} a[i][k] \times b[k][j]$

Matrix: ADT

```
class Matrix{
public:
    // Construct
    Matrix(int r, int c);
    // Return the transpose of (*this) matrix
    Matrix Transpose(void);
    // Return sum of *this and b
    Matrix Add(Matrix b);
   // Return the multiplication of *this and b
    Matrix Multiply(Matrix b);
private:
    // Array representation
     int **a, rows, cols;
};
```

Transpose : Code

```
Matrix Matrix::Transpose(void){
   Matrix c(cols, rows);
   for (i=0; i<rows; i++)           // O(rows)
      for (j=0; j<cols; j++)        // O(cols)
          c[j][i]=a[i][j];
   return c;
}
```

- Time complexity: $O(rows \cdot cols)$

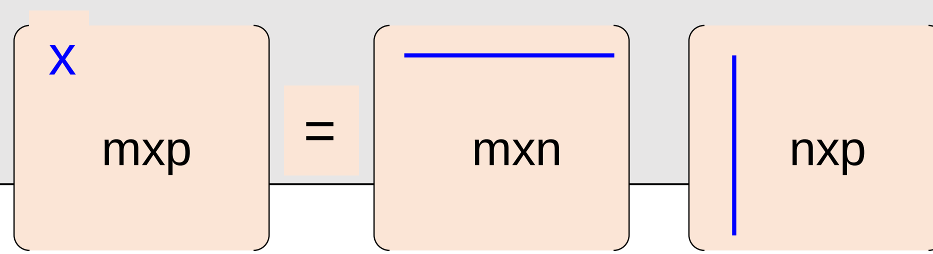
Add: Code

```
Matrix Matrix::Add(Matrix b){
   Matrix c(rows, cols);
   for (i=0; i<rows; i++)          // O(rows)
      for (j=0; j<cols; j++)       // O(cols)
         c[i][j]=a[i][j]+b[i][j];
   return c;
}
```

- Time complexity: $O(rows \cdot cols)$

Multiply: Code

```
Matrix Matrix::Multiply(Matrix b){
   Matrix c(rows, b.cols);
   for (i=0; i<rows; i++) {             // O(rows)
      for (j=0; j<b.cols; j++) {        // O(b.cols)
          sum=0;
          for (k=0; k<cols; k++)        // O(cols)
              sum += a[i][k]*b[k][j];
          c[i][j]=sum;
      }
  }
  return c;
}
```



- Time complexity: O(rows · cols · b.cols)

2.4.2 Sparse Matrix

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

- A matrix has few non-zero elements.
- 2D array representation is inefficient.
  - Wasteful **memory** and **computing time**
  - Consider a matrix $A_{5000X5000}$ with only 100 nonzero elements!

Single Linear List Example

$$\begin{matrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{matrix}$$

list =

$$\begin{array}{l} \text{row} \\ \text{column} \\ \text{value} \end{array} \begin{bmatrix} 1 & 1 & 2 & 2 & 4 & 4 \\ 3 & 5 & 3 & 4 & 2 & 3 \\ 3 & 4 & 5 & 7 & 2 & 6 \end{bmatrix}$$

One Linear List Per Row

0	0	3	0	4	row1 = [(3, 3), (5,4)]
0	0	5	7	0	row2 = [(3,5), (4,7)]
0	0	0	0	0	row3 = []
0	2	6	0	0	row4 = [(2,2), (3,6)]

Sparse Matrix Representation

- We use an array, ***smArray*[]**, of ***triple* <*row, col, value*>** to store those nonzero elements.
- Triples are stored in a ***row-major*** order.

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

	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

2.4.2

ADT2.4

Sparse Matrix: ADT

```
class SparseMatrix{
public:
    // Construct, t is the capacity of nonzero terms
    SparseMatrix(int r, int c, int t);
    // Return the transpose of (*this) matrix
    SparseMatrix Transpose(void);
    // Return sum of *this and b
    SparseMatrix Add(SparseMatrix b);
    // Return the multiplication of *this and b
    SparseMatrix Multiply(SparseMatrix b);
private:
    // Sparse representation
    int rows, cols, terms, capacity;
    MatrixTerm *smArray;
};
```

```
class MatrixTerm {
 friend SparseMatrix;
 int row, col, value;
};
```

Approximate Memory Requirements

- 5000 x 5000 matrix with 100 nonzero elements, 4 bytes per element

- 2D array
  - 5000 x 5000 x 4 = 100 million bytes
- Class SparseMatrix
  - 100 x 4 x 3 + 4 = 1204 bytes

2.4.3

Trivial Transpose

- $c[i][j] = a[j][i]$

	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

Transpose →

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

- Problem: the nonzero terms in A^T are no longer stored in row major order!

Smart Transpose

Because the row and column are swapped, we trace the nonzero terms in a **column-major** order.

```
For(all non-zero elements in column j)
  Store a(i,j,value) as aT(j,i,value)
```

	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]			
smArray[3]			
smArray[4]			
smArray[5]			
smArray[6]			
smArray[7]			

Smart Transpose: Code

```
SparseMatrix SparseMatrix::Transpose()
{ // Return the transpose of (*this) matrix
  // b.smArray has the same number of nonzero terms
  SparseMatrix b(cols, rows, terms);
  if (terms > 0) // has nonzero terms
  {
    int currentB = 0;
    for(int c=0; c<cols; c++)      // O(cols)
      for(int i=0; i<terms; i++)  // O(terms)
        if(smArray[i].col == c)
        {
          b.smArray[currentB].row = c;
          b.smArray[currentB].col = smArray[i].row;
          b.smArray[currentB++].value = smArray[i].value;
        }
  }
  return b;
}
```

Fast Transpose

- Examine all terms only twice!
- Use additional space to store
  - rowSize[i]: # of nonzero terms in i^{th} row of A^T
  - rowStart[i]: location of nonzero term in i^{th} row of A^T
  - For i>0, rowStart[i]=rowStart[i-1]+rowSize[i-1]
- Copy element from A to A^T one by one.
- Time complexity: O(terms + cols)!

Fast Transpose

- Count the # of nonzero terms in each row of A^T
- Calculate the location of 1^{st} nonzero term i^{th} row of A^T

A	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

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

Fast Transpose

- Copy element from A to A^T one by one

A	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

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

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

Fast Transpose

- Copy element from A to A^T one by one

A	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

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

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

Fast Transpose

- Copy element from A to A^T one by one

A	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

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

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

Fast Transpose

- Copy element from A to A^T one by one

A	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

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

A^T	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]			

Fast Transpose

- Copy element from A to A^T one by one

A	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

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

A^T	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

Fast Transpose

- Copy element from A to A^T one by one

A	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

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

A^T	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

Fast Transpose

- Copy element from A to A^T one by one

A	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

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

A^T	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

Fast Transpose: Codes

```
SparseMatrix SparseMatrix::FastTranspose( )
{ // Compute the transpose in O(terms + cols) time
  SparseMatrix b(cols, rows, terms);
  if (terms > 0) {
    int *rowSize = new int[cols];
    int *rowStart = new int[cols];
    // compute rowSize[i]=number of terms in row i of b
    fill(rowSize, rowSize+cols, 0);
    for(int i=0; i<terms; i++) rowSize[smArray[i].col]++;

    // rowStart[i] = starting position of row i in b
    rowStart[0] = 0;
    for(int i=1; i<cols; i++)
      rowStart[i]=rowStart[i-1]+rowSize[i-1];

    for(int i=0; i<terms; i++)
    { // copy terms from *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]++; // Increase the start pos by 1
    }
    delete [] rowSize;
    delete [] rowStart;
  }
  return b;
}
```

Computation Time Comparison

Trivial Transpose	Smart Transpose	Fast Transpose
$\boldsymbol{O(rows \cdot cols)}$	$\boldsymbol{O(cols \cdot terms)}$	$\boldsymbol{O(terms + cols)}$

- For a dense matrix ($\text{terms} = rows \cdot cols$)
  Fast equals to **Trivial**: $O(rows \cdot cols)$
  Smart is the slowest: $O(rows \cdot cols^2)$
- For a sparse matrix ($\text{terms} \ll rows \cdot cols$)
  - **Fast** is faster than **Trivial** and **Smart** ones

2.4.4

Sparse Matrix Multiplication

- Compute the transpose of b

c: m x p   a: m x n   b: n x p

$$\begin{pmatrix} x & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{pmatrix} = \begin{pmatrix} 0 & 5 & 2 & 0 & 0 & 7 \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{pmatrix} \begin{pmatrix} 3 & & & \\ 0 & & & \\ 4 & & & \\ 3 & & & \\ 6 & & & \\ 5 & & & \end{pmatrix}$$

Sparse Matrix Multiplication

- Use approach similar to "**Polynomial Addition**" to compute the X!

c: m x p a: m x n b^T: p x n

$$\begin{pmatrix} x & & \\ & & \\ & & \end{pmatrix} = \begin{array}{c} \begin{pmatrix} \cancel{0} & 5 & 2 & \cancel{0} & \cancel{0} & 7 \\ & & & & & \\ & & & & & \end{pmatrix} \\ \hphantom{\cancel{0}\ 5\ 2\ \cancel{0}\ \cancel{0}\ }p \end{array} \quad \begin{array}{c} \begin{pmatrix} 3 & \cancel{0} & 4 & 3 & 6 & 5 \\ & & & & & \\ & & & & & \end{pmatrix} \\ \hphantom{3\ \cancel{0}\ 4\ 3\ 6\ }q \end{array}$$

ref code in the textbook

$$x = (2)(4) + (7)(5) = 43$$

Time Complexity

```
SparseMatrix SparseMatrix::Multiply(SparseMatrix b)
{ // Compute the transpose of b
  SparseMatrix bT = b.FastTranspose(); // O(b.terms+b.cols)

  for ith row in smArray                       // O(rows)
    for jth row in bT.smArray                  // O(b.cols)
      Perform "Polynomal Addition"             // O(Terms[i]+b.Terms[j])
}
```

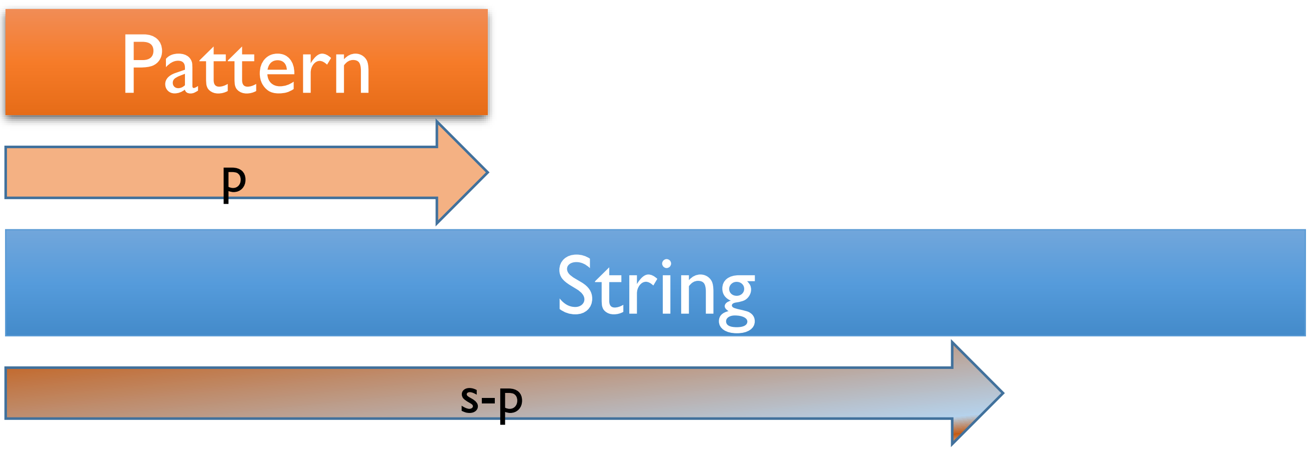
- Complexity:
  - O(rows · b.cols · (Terms[i] + b.Terms[j]))
  - rows · Terms[i] = a.terms and b.cols · b.Terms[j] = b.terms
  - O(rows · b.terms + b.cols · a.terms)

2.6

The String Abstract Data Type

Simple String Pattern Matching

- s= string.length();
- p= pattern.length();



- O(s*p)

The Knuth-Morris-Pratt Alg.

- Complexity: O(p+s)

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

$$f = \begin{cases} \text{largest } k < j, \text{s.t. } p_0 \dots p_k = p_{j-k} \dots p_j & , \exists k \geq 0 \\ -1 & , otherwise \end{cases}$$

- If a partial match is found such that $s_{i-j} \dots s_{i-1} = p_0 \dots p_{j-1}$ and $s_i \neq p_j$ then matching may resume by comparing s_i and $p_{f(j-1)+1}$

Observation

string

….	a	b	c	a	b	c	a	x	….

a	b	c	a	b	c	a	c	a	b

pattern

Pattern Matching

string

....	**a**	**b**	**c**	**a**	**b**	**c**	**a**	**x**	**....**

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

pattern

2.6.2
P2.16

Pattern-matching with a Failure Function

```
int String::FastFind(String pat) {
   // Determine if pat is a substring of s
   int PosP = 0, PosS = 0;  // j=> PosP, i=> PosS
   int LengthP = pat.Length(), LengthS = Length();

   while((PosP < LengthP) && (PosS < LengthS))
   {
      if (pat.str[PosP] == str[PosS]) {
            PosP++; PosS++;
      } else
            if (PosP == 0) PosS++;
            else PosP = pat.f[PosP-1] + 1;
   }
   if (PosP < LengthP) return -1;
   else return PosS-LengthP;
}
```