2.2 The Array as an Abstract Data Type

Array:

- A set of pairs: <index, value> (correspondence or mapping)

- Two operations: retrieve, store

Now we will use the C++ class to define an ADT.
class GeneralArray {

// a set of pairs <index, value> where for each value of
// index in IndexSet there is a value of type float. IndexSet is
// a finite ordered set of one or more dimensions.

public:

    GeneralArray(int j, RangeList list, float initValue =
                  defaultValue);

    // The constructor GeneralArray creates a j
    // dimensional array of floats; the range of the kth
    // dimension is given by the kth element of list.
    // For all i∈IndexSet, insert <i, initValue> into the array.
float Retrieve(index i);
// if (i∈IndexSet) return the float associated with i in the
// array; else throw an exception.

void Store(index i, float x);
// if (i∈IndexSet) replace the old value associated with i
// by x; else throw an exception.
}
} //end of GeneralArray
Note:

- Not necessarily implemented using consecutive memory
- Index can be coded any way
- GeneralArray is more general than C++ array as it is more flexible about the composition of the index set
- To be simple, we will hereafter use the C++ array
Array can be used to implement other abstract data types. The simplest one might be:

**Ordered or linear list.**

Example:

(Sun, Mon, Tue, Wed, Thu, Fri, Sat)

(2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A)

() // empty list
More generally, **An ordered list** is either empty or
\( (a_0, a_1, ..., a_{n-1}) \). // index important

**Main operations:**

1. Find the length, \( n \), of the list.
2. Read the list from left to right (or right to left)
3. Retrieve the \( i \)th element, \( 0 \leq i < n \).
4. Store a new value into the \( i \)th position, \( 0 \leq i < n \).
(5) Insert a new element at position $i$, $0 \leq i < n$, causing elements numbered $i$, $i+1, \ldots n-1$ to become numbered $i+1$, $i+2, \ldots n$.

(6) Delete the element at position $i$, $0 \leq i < n$, causing elements numbered $i+1$, $i+2, \ldots n-1$ to become numbered $i$, $i+1, \ldots n-2$. 
How to represent ordered list efficiently?

- Sequential mapping
  - Use array: \( a_i \leftrightarrow \text{index } i \)

- Complexity
  - Random access any element in \( O(1) \).
  - Operations (5) and (6)?
    - Data movement
      - \( O(n) \)

Now let us look at a problem requiring ordered list.
Problem:

Build an ADT for the representation and manipulation of symbolic polynomials.

\[ A(x) = 3x^2 + 2x + 4 \]
\[ B(x) = x^4 + 10x^3 + 3x^2 + 1 \]

Degree: the largest exponent
class Polynomial {
    // \( p(x) = a_0x^{e_0} + \ldots + a_nx^{e_n} \); a set of ordered pairs of \(<e_i, a_i>\),
    // where \( a_i \) is a nonzero float coefficient and \( e_i \) is a
    // non-negative exponent

public:
    Polynomial ( );
    // Construct the polynomial \( p(x) = 0 \)
void AddTerm (Exponent e, Coefficient c);
// add the term <e,c> to *this, so that it can be initialized

Polynomial Add (Polynomial poly);
// return the sum of the polynomials *this and poly

Polynomial Mult (Polynomial poly);
// return the product of the polynomials *this and poly

float Eval ( float f);
// evaluate polynomial *this at f and return the result
}
Polynomial Representation

Let a be $A(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$

Representation 1

```cpp
private:
    int degree; // degree ≤ MaxDegree
    float coef[MaxDegree+1];

    a.degree = ?

    n;

    a.coef[i] = ?

    a_{n-i}, 0 ≤ i ≤ n
```

Simple algorithms for many operations.
Representation 2

When a.\texttt{degree} $\ll$ \texttt{MaxDegree}, representation 1 is very poor in memory use. To improve, define variable sized data member as:

\begin{verbatim}
private:
    int degree;
    float *coef;

Polynomial::Polynomial(int d)
{
    int degree=d;
    coef= \texttt{new float}[\texttt{degree+1}];
}
\end{verbatim}
Representation 2 is still not desirable. For instance, \( x^{1000} + 1 \)

makes 999 entries of the coef be zero.

So, we store only the none zero terms:

Representation 3

\[
A(x) = b_m x^{e_m} + b_{m-1} x^{e_{m-1}} + \ldots + b_0 x^{e_0}
\]

Where \( b_i \neq 0, \quad e_m > e_{m-1} > \ldots, \quad e_0 \geq 0 \)
class Polynomial; // forward declaration

class Term {
friend Polynomial;

private:
    float coef; // coefficient
    int exp;    // exponent
};

class Polynomial {
public:
    ......  
private:
    Term *termArray;

    int capacity; // size of termArray
    int terms;    // number of nonzero terms
};
For \( A(x) = 2x^{1000} + 1 \)

\( A \)’s termArray looks like:

<table>
<thead>
<tr>
<th>coef</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>1000</td>
<td>0</td>
</tr>
</tbody>
</table>
Many zero --- good
Few zero --- ?
not very good
may use twice as much space as in presentation 2.
Polynomial Addition

Use presentation 3 to obtain $C = A + B$.

Idea:

Because the exponents are in descending order, we can adds $A(x)$ and $B(x)$ term by term to produce $C(x)$.

The terms of $C$ are entered into its termArray by calling function `NewTerm`.

If the space in termArray is not enough, its capacity is doubled.
Polynomial Polynomial::Add (Polynomial b)  
{   // return the sum of the polynomials *this and b.  
    Polynomial c;  
    int aPos=0, bPos=0;  
    while ((aPos < terms) && (b < b.terms))  
    {  
        if (termArray[aPos].exp==b.termArray[bPos].exp)  
        {  
            float t = termArray[aPos].coef + 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 the remaining terms of *this
for (; aPos < terms; aPos++)
    c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);

// add in the remaining terms of b
for (; bPos < b.terms; bPos++)
    c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
return c;
void Polynomial::NewTerm(const float theCoeff, const int theExp)
{
    // add a new term to the end of termArray.
    if (terms == capacity)
    {
        // double capacity of termArray
        capacity *= 2;
        term *temp = new term[capacity]; // new array
        copy(termArray, termAarray + terms, temp);
        delete [] termArray; // deallocate old memory
        termArray = temp;
    }

termArray[terms].coef = theCoeff;
termArray[terms++].exp = theExp;
}
Analysis of Add:
Let m, n be the number of nonzero terms in a and b respectively.
• line 3 and 4---O(1)
• in each iteration of the while loop, aPos or bPos or both increase by 1, the number of iterations of this loop ≤ m+n-1
• if ignore the time for doubling the capacity, each iteration takes O(1)
• line 20--- O(m), line 23--- O(n)
Asymptotic computing time for Add: O(m+n)
Analysis of doubling capacity:

- The time for doubling is linear in the size of the new array.
- Initially, c.capacity is 1.
- Suppose when Add terminates, c.capacity is $2^k$.
- The total time spent over all array doubling is
  \[ O\left( \sum_{i=1}^{k} 2^i \right) = O\left( 2^{k+1} \right) = O(2^k) \]
- Since c.terms > $2^{k-1}$ and $m + n \geq$ c.terms, the total time for array doubling is
  \[ O(c.terms) = O(m + n) \]
• so, even consider array doubling, the total run time of Add is $O(m + n)$.

• experiments show that array doubling is responsible for very small fraction of the total run time of Add.

Exercises: P93-2,6, P94-9
Sparse Matrices

Introduction

A general matrix consists of m rows and n columns (m × n) of numbers, as:

\[
\begin{array}{ccc}
0 & 1 & 2 \\
0 & -27 & 3 & 4 \\
1 & 6 & 82 & -2 \\
2 & 109 & -64 & 11 \\
3 & 12 & 8 & 9 \\
4 & 48 & 27 & 47 \\
\end{array}
\]

Fig.2.2(a) 5x3
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>-15</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-6</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>0</td>
</tr>
<tr>
<td>4</td>
<td>91</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>28</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 2.2(b)  $6\times6$
A matrix of $m \times m$ is called a **square**.

A matrix with many zero entries is called **sparse**.

**Representation:**

- A natural way ---
  
  - $a[m][n]$
  
  - access element by $a[i][j]$, easy operations. **But**
  
  - for sparse matrix, wasteful of both memory and time.

- Alternative way ---
  
  - store nonzero elements explicitly. 0 as default.
class SparseMatrix
{
   // a set of <row, column, value>, where row, column are
   // non-negative integers and form a unique combination;
   // value is also an integer.

   public:
      SparseMatrix ( int r, int c, int t);
      // creates a r×c SparseMatrix with a capacity of t nonzero
      // terms
      SparseMatrix Transpose ( );
      // return the SparseMatrix obtained by transposing *this
      SparseMatrix Add ( SparseMatrix b);
      SparseMatrix Multiply ( SparseMatrix b);
};
Sparse Matrix Representation

Use triple \(<row, col, value>\), sorted in ascending order by \(<row, col>\).

We need also the number of rows and the number of columns and the number of nonzero elements. Hence,

class SparseMatrix;
class MatrixTerm {
friend class SparseMatrix;
private:
    int row, col, value;
};
And in class `SparseMatrix`:

```cpp
private:
    Int rows, cols, terms, capacity;
    MatrixTerm *smArray;
```

Now we can store the matrix of Fig.2.2 (b) as Fig.2.3 (a).
<table>
<thead>
<tr>
<th>row</th>
<th>col</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>smArray[0]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[1]</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>[2]</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>[3]</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>[4]</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>[5]</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>[6]</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>[7]</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

**Fig.2.3 (a)**
Transposing a Matrix

Transpose:

If an element is at position $[i][j]$ in the original matrix, then it is at position $[j][i]$ in the transposed matrix.

Fig.2.3(b) shows the transpose of Fig2.3(a).

```c
for(col=0;col<n;col++)
    for(row=0;row<m;row++)
        n[col][row]=m[row][col];
T(n)=O(m×n)
```
First try:

For (each row i)

✓ take element (i, j, value)
✓ store it in (j, i, value) of the transpose;

Difficulty:

NOT knowing where to put (j, i, value) until all other elements preceding it have been processed.

<table>
<thead>
<tr>
<th>smArray</th>
<th>row</th>
<th>col</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>0</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>[1]</td>
<td>0</td>
<td>4</td>
<td>91</td>
</tr>
<tr>
<td>[2]</td>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>[3]</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>[4]</td>
<td>2</td>
<td>5</td>
<td>28</td>
</tr>
<tr>
<td>[5]</td>
<td>3</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>[6]</td>
<td>3</td>
<td>2</td>
<td>-6</td>
</tr>
<tr>
<td>[7]</td>
<td>5</td>
<td>0</td>
<td>-15</td>
</tr>
</tbody>
</table>
**Improvement:**

For (all elements in col j)

✓ store (i, j, value) of the original matrix as (j, i, value) of the transpose;

✓ (j, i, value) of the transpose;

Since the rows are in order,
we will locate elements in the correct column order.

<table>
<thead>
<tr>
<th>smArray</th>
<th>row</th>
<th>col</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>0</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>[1]</td>
<td>0</td>
<td>4</td>
<td>91</td>
</tr>
<tr>
<td>[2]</td>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>[3]</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>[4]</td>
<td>2</td>
<td>5</td>
<td>28</td>
</tr>
<tr>
<td>[5]</td>
<td>3</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>[6]</td>
<td>3</td>
<td>2</td>
<td>-6</td>
</tr>
<tr>
<td>[7]</td>
<td>5</td>
<td>0</td>
<td>-15</td>
</tr>
<tr>
<td></td>
<td>i</td>
<td>j</td>
<td>v</td>
</tr>
<tr>
<td>---</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>-3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>4</td>
<td>-7</td>
</tr>
</tbody>
</table>

**ma**

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>j</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>-3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>6</td>
<td>-7</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>3</td>
<td>14</td>
</tr>
</tbody>
</table>

**mb**

**col=1**

**col=2**
SparseMatrix SparseMatrix::Transpose()
{
    // return the transpose of *this
    SparseMatrix b(cols, rows, terms);
    if (terms > 0)
    {
        // nonzero matrix
        int currentB = 0;
    }
for ( int c=0; c<cols; c++ ) // transpose by columns
for ( int i=0; i<terms; i++ )
// find and move terms in column c
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;
Time complexity of Transpose:

- line 7-15 loop--- cols times
- line 10 loop--- terms times
- other line--- O(1)

Total time: O(cols* terms )

Additional space: O(1)

Think:

O(cols* terms) is not good. If terms = O(cols* rows ) then it becomes O(cols^2* rows )--- too bad!
Since with 2-dimensional representation, we can get an easy $O(\text{cols} \times \text{rows})$ algorithm as:

\[
\text{for (int } j=0; j < \text{columns}; j++)
\]
\[
\text{for (int } i=0; i < \text{rows}; i++) \ B[j][i] = A[i][j];
\]

Further improvement:

If we use some more space to store some knowledge about the matrix, we can do much better: doing it in $O(\text{cols} + \text{terms})$. 
<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>-3</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>-7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>-7</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>14</td>
</tr>
</tbody>
</table>
• get the number of elements in each column of *
  *this = the number of elements in each row of b;

• obtain the starting point in b of each of its rows;

• move the elements of *this one by one into their right position in b.

Now the algorithm FastTranspose.
<table>
<thead>
<tr>
<th>col</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>num[col]</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>cpot[col]</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

```
\begin{tabular}{cc}
<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>-3</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>-7</td>
</tr>
</tbody>
</table>
\end{tabular}
```

```
\begin{tabular}{cc}
<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>-3</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>-7</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>14</td>
</tr>
</tbody>
</table>
\end{tabular}
```
SparseMatrix SparseMatrix::FastTranspos() {
    // return the transpose of *this in O(terms+cols) time.
    SparseMatrix b(cols, rows, terms);
    if (terms > 0) {
        // nonzero matrix
        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); // initialize
        for (i=0; i<terms; i++) rowSize[smArray[i].col]++;
    }
}
11    // rowStart[i] = starting position of row i in b
12    rowStart[0] = 0;
13    for (i=1; i<cols; i++) rowStart[i]=rowStart[i-1]+rowSize[i-1];
14    for (i=0; i<terms; i++)
15    {
16        // copy from *this to b
17        int j = rowStart[smArray[i].col];
18        b.smArray[j].row = smArray[i].col;
19        b.smArray[j].col = smArray[i].row;
20        b.smArray[j].value = smArray[i].value;
21        rowStart[smArray[i].col]++;
22    } // end of for
After line 13, we get:

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>RowSize=</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RowStart=</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Note the error in P101 of the text book!
Analysis:
3 loops:
• line 10--- O(terms)
• line 13--- O(cols)
• line 14 – 21--- O(terms)
• line 14 – 21--- O(terms)
and line 9--- O(cols), other lines--- O(1)
Total: O(cols+terms)

This is a typical example for trading space for time.
Exercises: P107-1, 2, 4
The String Abstract data Type

A string \( S = s_0, s_1, \ldots, s_{n-1}, \) where \( s_i \in \text{char}, 0 \leq i < n, \) \( n \) is the length.

ADT 2.5 String

class String

{   // a finite set of zero or more characters;

public:

    String (char *init, int m);
    // initialize *this to string init of length m
bool operator == (String t);
// if *this equals t, return true else false.
bool operator ! ( );
// if *this is empty return true else false.
int Length ( );
// return the number of chars in *this
String Concat (String t);
String Substr (int i, int j);
int Find (String pat);
// return i such that pat matches the substring of *this that
// begins at position i. Return –1 if pat is either empty or not
// a substring of *this.
}
Assume the String class is represented by:

```cpp
private:
    char* str;
```
String Pattern Matching: A Simple Algorithm

S

Pat

Go Back!

S

Pat

i

i+1
int String::Find ( String pat )
{
    // Return -1 if pat does not occur in *this; otherwise
    // return the first position in *this, where pat begins.
    if (pat.Length( ) == 0) return -1; // pat is empty
    for (int start=0; start<=Length( ) - pat.Length( ); start++)
    {
        // check for match beginning at str[start]
        for (int j=0; j<pat.Length( )&&str[start+j]==pat.str[j]; j++)
            if (j== pat.Length( )) return start; // match found
        // no match at position start
    }
    return -1; // pat does not occur in s
}
The complexity of it is $O(\text{LengthP} \times \text{LengthS})$.

Problem:

re-scanning.

Even if we check the last character of pat first, the time complexity can’t be improved!
String Pattern Matching: The Knuth-Morris-Pratt Algorithm

Can we get an algorithm which *avoid rescanning* the strings and works in $O(\text{Length}_P + \text{Length}_S)$?

This is optimal for this problem, as in the worst it is necessary to look at characters in the pattern and string at least once.
**Basic Ideas:**

- Rescanning to avoid missing the target ---
  - too conservative

- If we can go without rescanning, it is likely to do the job in $O(\text{LengthP} + \text{LengthS})$.

- Preprocess the pattern, to get some knowledge of the characters in it and the position in it, so that if a mismatch occurs we can determine where to continue the search and avoid moving backwards in the string.

Now we show details about the idea.
case:  \( j = 0 \)
An concrete example:

\[
s = \ldots a \ b \ d \ a \ b \ \ \ ? \ldots \ \\
pat = \ a \ b \ d \ a \ b \ c \ a \ c \ a \ b
\]
case: \( j \neq 0 \)
To formalize the above idea:

**Definition:** if $p = p_0 p_1 \ldots p_{n-1}$ is a pattern, then its failure function $f$, is defined as:

$$f(j) = \begin{cases} 
\text{largest } k < j, \text{ such that } p_0 p_1 \ldots p_k = p_{j-k} p_{j-k+1} \ldots p_j \\
-1 \quad \text{otherwise}
\end{cases}$$

if such $k \geq 0$ exists
For example, \( \text{pat} = a \ b \ c \ a \ b \ c \ a \ c \ a \ b \), we have

<table>
<thead>
<tr>
<th>( j )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{pat} )</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>( f )</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Note:**
- **largest**: no match be missed
- **\( k < j \)**: avoid dead loop
From the definition of $f$, we have the following rule for pattern matching:

If a partial match is found such that $s_{i-j}...s_{i-1} = p_0p_1...p_{j-1}$ and $s_i \neq p_j$ then matching may be resumed by comparing $s_i$ and $p_{f(j-1)+1}$ if $j \neq 0$.

If $j=0$, then we may continue by comparing $s_{i+1}$ and $p_0$.

The failure function is represented by an array of integers $f$, which is a private data member of String.

Now the algorithm **FastFind**.
```c
1 int String::FastFind (String pat)
2 { // Determine if pat is a substring of s
3    int PosP = 0, PosS = 0;
4    int LengthP= pat.Length( ), LengthS= Length( );
5    while ((PosP < LengthP) && (PosS < LengthS))
6        if ( pat.str[PosP] == str[PosS] ) { // characters match
7            PosP ++; PosS ++;
8        }
9    else
10        if ( PosP==0)
11            PosS++;
12        else PosP= pat.f [PosP-1] + 1;
13 if ((PosP < LengthP) || LengthP==0)) return -1;
14        else return PosS  - LengthP ;
15    }
```
Analysis of FastFind:

- Line 7 and 11 --- at most LengthS times, since PosS is increased but never decreased. So PosP can move right on pat at most LengthS times (line 7).
- Line 12 moves PosP left, it can be done at most LengthS times. Note that f(j-1)+1< j.

Consequently, the computing time is $O(\text{LengthS})$.

How about the computing of the f for the pattern? By similar idea, we can do it in $O(\text{LengthP})$. 
If a=b, then $f(j)=f(j-1)+1$ else
If $c = b$, $f(j) = f(f(j-1)) + 1 = f^2(j-1) + 1$ else ……

In general, we have the following restatement of the failure function:
\[ f(j) = \begin{cases} 
-1 & \text{if } j=0 \\
\text{prime}(j-1)+1 & \text{where } m \text{ is the least } k \text{ for which } \text{prime}_f^k(j-1)+1 = p_j \\
-1 & \text{if there is no } k \text{ satisfying the above} 
\end{cases} \]

Now we get the algorithm to compute \( f \).
void String::Failurefunction()
{
    int LengthP = Length();
    int f[0] = -1;
    for (int j = 1; j < LengthP; j++) // compute f[j]
    {
        int i = f[j-1];
        while (((*(str+j)!=(str+i+1)) && (i>=0)) i = f[i]; // try for m
        if ( *(str+j)==*(str+i+1))
            f[j] = i + 1;
        else f[j] = -1;
    }
}
Analysis of fail:

- In each iteration of the while i decreases (line 8, and f(j)<j)
- i is reset (line 7) to –1 (when the previous iteration went through line 11), or to a value 1 greater than its value on the previous iteration (when through line 10).
- There are only LengthP –1 executions of line 7, the value of i has a total increment of at most LengthP –1.
- i cannot be decremented more than LengthP –1 times, the while is iterated at most LengthP –1 times over the whole algorithm.
Consequently, the computing time is $O(\text{Length}_P)$.

Now we can see, when the failure function is not known in advance, pattern matching can be carried out in $O(\text{Length}_P + \text{Length}_S)$ by first computing the failure function and then using the FastFind.

Exercises: P118-1, P119-7, 9
Experiment 1: P123-8