Chapter 2 Array and Structures

C++ Class
Array
Structures and Unions
Polynomial ADT
Sparse Matrix ADT
String ADT
Representation of Multidimensional Arrays

C++ **Class**

- Class
  - A class name
  - Data members
  - Member functions
- Levels of program access
  - Public: section of a class can be accessed by anyone
  - Private: section of a class can only be accessed by member functions and friends of that class
  - Protected: section of a class can only be accessed by member functions and friends of that class, and by member functions and friends of derived classes

宣告

Declaration of Class **Rectangle**

```
#ifndef RECTANGLE_H
#define RECTANGLE_H
// In the header file
class Rectangle {
public:                                 // The following members are public
        Rectangle();                    // Constructor
        ~Rectangle();                   // Deconstructor
        int GetHeight();                // return the height of the rectangle
        int GetWidth();                 // return the width of the rectangle
private:                                // The following members are private
        int x1, y1, h, w;
        // (x1, y1) are the coordinates of the bottom left corner of the rectangle
        // w is the width of the rectangle; h is the height of the rectangle
};
#endif
```

Implementation of Rectangle Operations

```
// In the source file Rectangle.C
#include "Rectangle.h"

/*
   The prefix "Rectangle::" identifies GetHeight() and GetWidth() as member
   functions belong to class Rectangle. It is required because the member
   functions are implemented outside their class definition
*/
int Rectangle::GetHeight() { return h;}
int Rectangle::GetWidth() { return w;}
```

Constructor and Destructor

- **Constructor:** is a member function which initializes data members of an object.
  - Adv: all class objects are well-defined as soon as they are created.
  - Must have the same name of the class
  - Must <u>not</u> specify a return type or a return value
- **Destructor**: is a member function which deletes data members immediately before the object disappears.
  - Must be named identical to the name of the class prefixed with a tilde ~.
  - It is <u>invoked automatically</u> when a class object goes out of scope or when a class object is deleted.

Examples of Rectangle Constructor

```
Rectangle::Rectangle(int x, int y, int height, int width)
{
    x1 = x;  y1 = y;
    h = height;      w = width;
}
```

default constructor

```
Rectangle::Rectangle(int x = 0, int y = 0, int height = 0, int
width = 0)
: x1(x), y1(y), h(height), w(width)
{ }
```

```
        Rectangle r(1, 3, 6, 6);
        Rectangle *s = new Rectangle(0, 0, 3, 4);
```

Operator Overloading

- C++ can distinguish the operator == when comparing two floating point numbers and two integers.
- But what if you want to compare two Rectangles?

```
int Rectangle::operator==(const Rectangle &s)
{
    if (this == &s) return 1;
    if ((x1 == s.x1) && (y1 == s.y1) && (h == s.h) && (w == s.w))
        return 1;
    else
        return 0;
}
```

Array

- Arrays
  - Array: a set of pairs, <index, value>
  - data structure
    - For each index, there is a value associated with that index.
  - representation (possible)
    - Implemented by using consecutive memory.
    - In mathematical terms, we call this a *correspondence* or a *mapping*.
  - E.g., int list[5]: list[0], …, list[4] each contains an integer

	0	1	2	3	4
list					

Arrays

- When considering an ADT we are more concerned with the **operations** that can be performed on an array.
  - Aside from **creating** a new array, most languages provide only two standard operations for arrays, one that **retrieves** a value, and a second that **stores** a value.
  - ADT 2.1 shows a definition of the array ADT

ADT of Arrays

structure *Array* is

objects: A set of pairs <*index*, *value*> where for each value of *index* there is a value from the set *item*. *Index* is a finite ordered set of one or more dimensions, for example, $\{0, \cdots, n-1\}$ for one dimension, $\{(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)\}$ for two dimensions, etc.

functions:

for all $A \in Array$, $i \in index$, $x \in item$, j, $size \in$ integer

Array Create(j, *list*)	::=	**return** an array of j dimensions where *list* is a j-tuple whose ith element is the the size of the ith dimension. *Items* are undefined.
Item Retrieve(A, i)	::=	**if** ($i \in index$) **return** the item associated with index value i in array A **else return** error
Array Store(A,i,x)	::=	**if** (i in *index*) **return** an array that is identical to array A except the new pair $<i, x>$ has been inserted **else return** error.

end *Array*

Arrays

- When considering an ADT we are more concerned with the **operations** that can be performed on an array.
  - Aside from **creating** a new array, most languages provide only two standard operations for arrays, one that **retrieves** a value, and a second that **stores** a value.
  - ADT 2.1 shows a definition of the array ADT
  - The advantage of this ADT definition is that

    It clearly points out the fact that the array is a more general structure than “a consecutive set of memory locations.”

The array as an ADT (4/6)

- Arrays in C/C++
  - int list[5], *plist[5];
  - list[5]: (five integers) list[0], list[1], list[2], list[3], list[4]
  - *plist[5]: (five pointers to integers)
    - plist[0], plist[1], plist[2], plist[3], plist[4]
  - implementation of 1-D array
    - E.g.,

list[0]	base address = α
list[1]	α + sizeof(int)
list[2]	α + 2*sizeof(int)
list[3]	α + 3*sizeof(int)
list[4]	α + 4*sizeof(int)

The array as an ADT

- Arrays in C/C++ (cont'd)
  - Compare int *list1 and int list2[5] in C/C++.

    Same: list1 and list2 are pointers.

    Difference: list2 reserves five locations.
  - Notations:

    list2 — a pointer to list2[0]

    (list2 + i) — a pointer to list2[i] (&list2[i])

    *(list2 + i) — list2[i]

The array (6/6)

- Example:
  1-dimension array addressing
  - int one[] = {0, 1, 2, 3, 4};
  - Goal: print out address and value

```
void print1(int *ptr, int rows){
/* print out a one-dimensional array using a pointer */
        int i;
        printf("Address Contents\n");
        for (i=0; i < rows; i++)
                printf("%8u%5d\n", ptr+i, *(ptr+i));
        printf("\n");
}
```

Structures and Unions

- Arrays are collections of data of the same type.
- Structures (records)
  - In C/C++ there is an alternate way of grouping data that permit the data to vary in type.
    - **Struct**
  - A structure is a collection of data items, where each item is identified as to its type and name.

```
struct {
        char name[10];
        int age;
        float salary;
        } person;
```

```
strcpy(person.name,"james");
person.age = 10;
person.salary = 35000;
```

Structures and Unions

- Create structure data type
  - We can create our own structure data types by using the **typedef** statement as below:

```
typedef struct human_being {
        char name[10];
        int age;
        float salary;
        };
```

or

```
typedef struct {
        char name[10];
        int age;
        float salary;
        } human_being;
```

  - Declaration of variables

struct human_being person1, person2;

human_being person1, person2;

Structures and Unions

– We can also embed a structure within a structure.

```
typedef struct {
        int month;
        int day;
        int year;
        } date;

typedef struct human_being {
        char name[10];
        int age;
        float salary;
        date dob;
        };
```

- A person born on February 11, 1994, would have have values for the *date* **struct** set as

```
person1.dob.month = 2;
person1.dob.day = 11;
person1.dob.year = 1944;
```

Structures and Unions

- Unions
  - A **union** declaration is similar to a structure.
  - The fields of a **union** must share their memory space.
  - Only one field of the **union** is “active” at any given time

Example: Add fields for male and female.

```
typedef struct sex_type {
   enum tag_field {female, male} sex;
   union {
      int children;
      int beard ;
      } u;
   };
typedef struct human_being {
          char name[10];
          int age;
          float salary;
          date dob;
          sex_type sex_info;
          };
human_being person1, person2;
```

```
person1.sex_info.sex = male;
person1.sex_info.u.beard = FALSE;
person2.sex_info.sex = female;
person2.sex_info.u.children = 4;
```

Structures and Unions

- Internal implementation of structures
  - The fields of a structure in memory will be stored in the same way using increasing address locations in the order specified in the structure definition.
  - Holes or padding may actually occur
    - Within a structure to permit two consecutive components to be properly aligned within memory
  - The size of an object of a struct or union type is the amount of storage necessary to represent the largest component, including any padding that may be required.

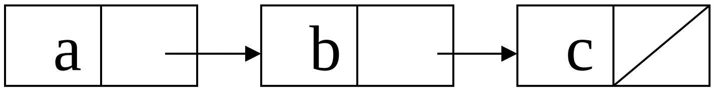
Structures and Unions

- Self-Referential Structures
  - One or more of its components is a pointer to itself.

```
typedef struct list {
   char data;
   list *link;
}
```

  - Construct a list with three nodes

```
list item1, item2, item3;
item1.data=‘a’;
item2.data=‘b’;
item3.data=‘c’;
item1.link=item2.link=item3.link=NULL;
item1.link=&item2;
item2.link=&item3;
```



The Polynomial ADT (1/20)

- Ordered List or Linear List
  - ordered (linear) list: (item1, item2, item3, …, item *n*)
    - (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)
    - (Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King)
    - (basement, lobby, mezzanine, first, second)
    - (1941, 1942, 1943, 1944, 1945)
    - (a_1, a_2, a_3, …, a_{n-1}, a_n)

The Polynomial ADT (2/20)

- **Operation**s on Ordered List
  1) Finding the length, n , of the list.
  2) Reading the items from left to right (or right to left).
  3) Retrieving the i'th element.
  4) Storing a new value into the i'th position.
  5) **Inserting** a new element at the position i , causing elements numbered i, i+1, …, n to become numbered i+1, i+2, …, n+1
  6) **Deleting** the element at position i , causing elements numbered i+1, …, n to become numbered i, i+1, …, n-1
- Implementation
  - sequential mapping (1)~(4)
  - non-sequential mapping (5)~(6)

    Linked list

Performing operations 5 and 6 requires data movement: Costly

The Polynomial ADT(3/20)

- Polynomial examples:
  - Two example polynomials are:
    - degree
    - $A(x) = 3x^{20} + 2x^5 + 4$ and $B(x) = x^4 + 10x^3 + 3x^2 + 1$
  - Assume that we have two polynomials, $A(x) = \sum a_i x^i$ and $B(x) = \sum b_i x^i$ where x is the variable, a_i is the coefficient, and i is the exponent, then:
    - $A(x) + B(x) = \sum(a_i + b_i)x^i$
    - $A(x) \cdot B(x) = \sum(a_i x^i \sum(b_i x^i))$

Similarly, we can define subtraction and division on polynomials, as well as many other operations.

class *polynomial*

{

objects: $p(x) = a_1 x^{e_1} + \ldots + a_n x^{e_n}$ a set of ordered pairs of <e_i, a_i>

where $a_i \in$ *Coefficient and* $e_i \in$ *Exponent*

We assume that Exponent consists of integers ≥ 0

public:

```
Polynomial();
// return the polynomial p(x) = 0
int operator!();
// if *this is the zero polynomial, return 1; else return 0;
Coefficient Coef(Exponent e);
// return the coefficient of e in *this
Exponent LeadExp();
// return the largest exponent in *this
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 the polynomial *this at f and return the result
```

}; //end of Polynomial

The Polynomial ADT(5/20)

- There are two ways to create the type *polynomial* in C++
- Representation I

```
class Polynomial {
    public:
        Polynomial add(Polynomial b);
      …
    private:
        int degree;
        float coef [MAXDegree+1]; /*Max degree of polynomial+1*/
    }
```

drawback: the first representation may waste space.

Polynomial a;
a.degree=n
a.coef[i]=$a_{n-i}, 0 \leq i \leq n$

The Polynomial ADT (6/20)

class *Polynomial* **{**
// $p(x) = a_0 x^{e_0} + \cdots + a_n x^{e_n}$, a ordered pairs of <$e_i$, a_i>,
// where a_i is a nonzero float coefficient and e_i is a nonzero integer exponent.
public:
Polynomial( )**;**
// constructor, create $p(x) = 0$ 。

Polynomial Add(*Polynomial poly*)**;**
// return the polynomial of ***this**+*poly*.

Polynomial Mult(*Polynomial poly*)**;**
// return the polynomial of ***this**˙*poly*.

float *Eval*(**float** *f*)**;**
// return $p(f)$ of ***this**.
};

The Polynomial ADT (7/20)

- Polynomial Addition

  of **Representation I**

  (implemented in C++)

```
Polynomial Polynomial::Add(Polynomial b)
{// return sum of *this and b
    Polynomial c;
    int aPos = degree, bPos = b.degree;
    if (aPos>bPos)
        c. degree = aPos;
    else
        c. degree = bPos;
    while ((aPos>0) || (bPos >0))
        if (aPos==bPos) {
                float t = coef[aPos] + b. coef[bPos];
                If (t) c. coef[aPos]= t;
                 aPos--; bPos--;
        }
        else if (aPos>bPos) {
                c. coef[aPos]= coef[aPos];
                aPos--;
        }
        else {
                c. coef[bPos]= b. coef[bPos];
                bPos--;
        }
    return c;
}
```

advantage: easy implementation
disadvantage: waste space when **sparse**

The Polynomial ADT(8/20)

- Representation II

```
class Polynomial {
  …
private:
  int degree;     // degree ≤ MaxDegree
  float *coef;
}
Polynomial::Polynomial(int d) //constructor
{
   degree=d;
   coef=new float[degree+1]; // from 0~degree
}
```

Assume max degree is known

Waste space when the polynomial is sparse (e.g., $x^{1000}+1$)

The Polynomial ADT (9/20)

- Representation III
  - Store only nonzero terms
    - Class member of ***Term***
      - *coef* and *exp* store the coefficient and exponent of a non-zero term
    - Class member of ***Polynomial***
      - *termArray* is the array of nonzero terms
      - *terms* stores the number of nonzero terms
      - *capacity* stores the size of termArray

$$a(x) = 2x^{1000} + 1$$

a(x) uses only 6 units of space!

terms: 2
capacity: 6
termArray:

	0	1				
coef	2	1				
exp	1000	0				

When all terms are nonzeros, Representation III costs about twice as much space as does Represerntation II!

The Polynomial ADT (10/20)

```
class Polynomial;        // forward declaration

class Term {
  friend Polynomial;
  private:
    float coef;            // coefficient
    int exp;               // exponent
};

class Polynomial{
  …
  private:
    Term *termArray;       // array of nonzero terms
    int capacity;          // size of termArray
    int terms;             // no. of nonzero terms
    …
}
```

```
Polynomial::Polynomial (int c=1, int t = 0): capacity(c), terms(t){
     termArray=new Term[1];
}
```

The Polynomial ADT (11/20)

```
void Polynomial::NewTerm(const float theCoeff, const int theExp)
{// 在 termArray 的末端加入一個新項
    if (terms = = capacity)
    {// 將 termArray 的容量加倍
        capacity *= 2;
        term *temp = new term[capacity];    // 新陣列
        copy(termArray, termArray + terms, temp);
        delete [] termArra ;                // 釋放舊的記憶體
        termArray = temp;
    }
    termArray [terms].coef = theCoeff;
    termArray [terms++].exp = theExp;
}
```

加入一個新項，必要時將陣列大小加倍

Problem: Compaction is required when polynomials that are no longer needed. (data movement takes time.)

Representation III: $c(x) = a(x) + b(x)$

```
Polynomial Polynomial::Add(Polynomial b)
{// return sum of *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++;
        }

                c.NewTerm (termArray [aPos].coef,termArray [aPos].exp);
                aPos++;
        }
    // add remaining items of *this
    for ( ; aPos < terms ; aPos++)
        c.NewTerm (termArray [aPos].coef, termArray [aPos].exp);
    // add the remaining items of b(x)
    for ( ; bPos < b.terms ; bPos++)
        c.NewTerm (b.termArray [bPos].coef, b.termArray [bPos].exp);
    return c;
}
```

worst case: $m + n - 1$

$O(m + n)$?

$\Rightarrow O(m + n + \text{time spent for array doubling})$

The Polynomial ADT (13/19)

- Complexity Analysis of Represerntation III
  - How to estimate doubling time?
    - Assume $c.capacity$ is 2^k
    - Total time spent over all array doublings

    $\Rightarrow O(\sum_{i=1}^{k} 2^k)$

    $\Rightarrow O(2^{k+1})$ = $O(2^k)$
    - Since $c.terms\ > 2^{k-1}$ and $m+n \geq c.terms$

    $\Rightarrow O(c.terms\ ) = O(m+n)$

    $\Rightarrow$Total time spent over all array doublings is $O(m+n)$

The Polynomial ADT (14/20)

- Representation III (Advanced)
  - Use one global array to store all polynomials
  - Class member of ***Polynomial***
    - *start* and *finish* give the loctions of the begin and the last items of the polynomial
    - *free* gives the location of the next free location

$$a(x) = 2x^{1000} + 1$$

specification	representation
polynomial	<start, finish>
a	<0,1>

	a.start ↓	*a.finish* ↓	*free* ↓				
coef	2	1					
exp	1000	0					
	0	1	2	3	4	5	6

The Polynomial ADT (15/20)

- Representation III (Advanced)
  - Use one global array to store all polynomials
  - Class member of ***Ploynomial***
    - *start* and *finish* give the loctions of the begin and the last items of the polynomial
    - *free* gives the location of the next free location

$$a(x) = 2x^{1000} + 1$$
$$b(x) = x^4 + 10x^3 + 3x^2 + 1$$

specification	representation
polynomial	<start, finish>
a	<0,1>
b	<2,5>

	a.start ↓	*a.finish* ↓	*b.start* ↓			*b.finish* ↓	*free* ↓
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

The Polynomial ADT (17/20)

```
class Polynomial{
  ...
  private:
    static term *termArray;
    static int free, capacity;
    int Start, Finish;
    ...
}
```

static: shared by all class instances (objects)

```
Polynomial::Polynomial (){
    Start=Finish=free;
}
```

- ✓ storage requirements: start, finish, 2*(finish-start+1)
- ✓ non-sparse: twice as much as Representation II when all the items are nonzero

```
int Polynomial::capacity=100;
term Polynomial:: termArray=new Term[100];
int Polynomial::free = 0;
// free: location of next free location in temArray
```

The Polynomial ADT (18/20)

- To produce $c(x) = a(x) + b(x)$

$$a(x) = 2x^{1000} + 1$$
$$b(x) = x^4 + 10x^3 + 3x^2 + 1$$

C.Start

	a.start ↓	*a.finish* ↓	*b.start* ↓			*b.finish* ↓	*free* ↓
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

coef							
exp							
	7	8	9	10	11	12	13

Representation III (advanced): $c(x) = a(x) + b(x)$

```
Polynomial Polynomial::Add(Polynomial b)
{// return sum of *this and b
    Polynomial c;
    int aPos = Start, bPos = b.Start;
    while ((aPos <= Finish) && (bPos < =b.Finish))
        if (termArray [aPos].exp = = 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 < termArray [bPos].exp) {
                c.NewTerm (termArray [bPos].coef, termArray [bPos].exp);
                bPos++;
        }
        else {
                c.NewTerm (termArray [aPos].coef,termArray [aPos].exp);
                aPos++;
        }
    // add remaining items of *this
    for ( ; aPos < terms ; aPos++)
        c.NewTerm (termArray [aPos].coef, termArray [aPos].exp);
    // add remaining items of b(x)
    for ( ; bPos < b.terms ; bPos++)
        c.NewTerm (b.termArray [bPos].coef, b.termArray [bPos].exp);
    return c;
}
```

The Polynomial ADT (20/20)

- Adding a New Term (Representation III)

```
void Polynomial::NewTerm(const float theCoeff, const int theExp)
{// add a new item at the end of termArray
    if (free = = capacity)
    {// doubling the capacity of termArray
        capacity *= 2;
        Terms *temp = new Term[capacity];    // create a new array
        copy(termArray, termArray + free-1,temp);
        delete [] termArray ;                // release the mem of the old array
        termArray = temp;
    }
    termArray [free].coef = theCoeff;
    termArray [free].exp = theExp;
    Finish=free++;
}
```

The Sparse Matrix ADT (1/23)

- In mathematics, a matrix contains m rows and n columns of elements, we write $m \times n$ to designate a matrix with m rows and n columns.

5*3

$$
\begin{array}{c|ccc|}
 & \text{col } 0 & \text{col } 1 & \text{col } 2 \\
\hline
\text{row } 0 & -27 & 3 & 4 \\
\text{row } 1 & 6 & 82 & -2 \\
\text{row } 2 & 109 & -64 & 11 \\
\text{row } 3 & 12 & 8 & 9 \\
\text{row } 4 & 48 & 27 & 47 \\
\hline
\end{array}
$$

15/15

6*6

$$
\begin{array}{c|cccccc|}
 & \text{col } 0 & \text{col } 1 & \text{col } 2 & \text{col } 3 & \text{col } 4 & \text{col } 5 \\
\hline
\text{row } 0 & 15 & 0 & 0 & 22 & 0 & -15 \\
\text{row } 1 & 0 & 11 & 3 & 0 & 0 & 0 \\
\text{row } 2 & 0 & 0 & 0 & -6 & 0 & 0 \\
\text{row } 3 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{row } 4 & 91 & 0 & 0 & 0 & 0 & 0 \\
\text{row } 5 & 0 & 0 & 28 & 0 & 0 & 0 \\
\hline
\end{array}
$$

sparse matrix
data structure?

8/36

The Sparse Matrix ADT (2/23)

- The standard representation of a matrix is a two dimensional array defined as $a[MAX\_ROWS][MAX\_COLS]$.
  - We can locate quickly any element by writing $a[i][j]$
- Sparse matrix wastes space
  - We must consider alternate forms of representation.
  - Our representation of sparse matrices should store only nonzero elements.
  - Each element is characterized by <row, col, value>.

The Sparse Matrix ADT(3/18)

class *SparseMatrix*
{// 三元組，<列，行，值>，的集合，其中列與行為非負整數，
// 並且它的組合是唯一的；值也是個整數。
public:
　　SparseMatrix(**int** *r*, **int** *c*,**int** *t*)**;**
　　// 建構子函式，建立一個有 *r* 列 *c* 行並且具有放 *t* 個非零項的容量

　　SparseMatrix Transpose(**);**
　　//回傳將 ***this** 中每個三元組的行與列交換後的 *SparseMatrix*

　　SparseMatrix Add(*SparseMatrix b*)**;**
　　// 如果 ***this** 和 *b* 的維度一樣，那麼就把相對應的項給相加，
　　// 亦即，具有相同列和行的值會被回傳；否則的話丟出例外。

　　SparseMatrix Multiply(*SparseMatrix b*)**;**
　　// 如果***this** 中的行數和 *b* 中的列數一樣多的話，那麼回傳的矩陣 *d* 就是 ***this** 和 *b*
　　//（依據 $d[i][j]=\Sigma(a[i][k] \cdot b[k][j]$，其中 $d[i][j]$是第 (i, j) 個元素）相乘的結果。*k* 的範圍
　　// 從 0 到***this** 的行數減 1；如果不一樣多的話，那麼就丟出例外。
};

The Sparse Matrix ADT(4/23)

- Sparse Matrix Representation
  - Use triple <row, column, value>
  - Store triples row by row
  - For all triples within a row, their column indices are in ascending order.
  - Must know the numbers of rows and columns and the number of nonzero elements

$$
\begin{array}{r c}
 & \begin{array}{ccc} \text{col } 0 & \text{col } 1 & \text{col } 2 \end{array} \\
\begin{array}{r} \text{row } 0 \\ \text{row } 1 \\ \text{row } 2 \\ \text{row } 3 \\ \text{row } 4 \end{array} &
\begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{bmatrix}
\end{array}
$$

The Sparse Matrix ADT(5/23)

- To implement the class SparseMatrix in C++
  - Class *MatrixTerm* stores nonzero terms
  - class SparseMatrix stores information about the spase matrix

```
class SparseMatrix; // forward declaration
class MatrixTerm {
  friend class SparseMatrix;
  private:
     int row, col, value; // row and col are index of the item
};
class SparseMatrix{
…
private:
   int Rows, Cols, Terms; // Rows/Cols: No. of rows/columns in the
                           //  matrix and col are index of the item
   MatrixTerm smArray[MaxTerms]; }
```

The Sparse Matrix ADT (6/23)

- Representing the sparse matrix in the array *a*.
  - Represented by a two-dimensional array.
  - Each element is characterized by <row, col, value>.

$$
\begin{array}{c|cccccc}
 & \text{col } 0 & \text{col } 1 & \text{col } 2 & \text{col } 3 & \text{col } 4 & \text{col } 5 \\
\hline
\text{row } 0 & 15 & 0 & 0 & 22 & 0 & -15 \\
\text{row } 1 & 0 & 11 & 3 & 0 & 0 & 0 \\
\text{row } 2 & 0 & 0 & 0 & -6 & 0 & 0 \\
\text{row } 3 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{row } 4 & 91 & 0 & 0 & 0 & 0 & 0 \\
\text{row } 5 & 0 & 0 & 28 & 0 & 0 & 0
\end{array}
$$

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

transpose →

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

row, column in ascending order

The Sparse Matrix ADT (7/23)

- Transpose a Matrix
  - For each row i
    - take element $< i, j, value >$ and store it in element $< j, i, value >$ of the transpose.
    - Difficulty: where to put $< j, i, value >$

      (0, 0, 15) ====> (0, 0, 15)
      (0, 3, 22) ====> (3, 0, 22)
      (0, 5, -15) ====> (5, 0, -15)
      (1, 1, 11) ====> (1, 1, 11)
      Move elements down very often.

      Order should be maintained!
  - For all elements in column j,
    place element $< i, j, value >$ in element $< j, i, value >$

Iteration i: scan the array and process the entries with $col = i$

The Sparse Matrix ADT (8/23)

CurrentB →

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

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

Iteration 0: scan the array and process the entries with $col = 0$

CurrentB → b

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

Iteration 1: scan the array and process the entries with $col = 1$

The Sparse Matrix ADT (10/23)

```
SparseMatrix SparseMatrix::Transpose( )
{// return transpose of *this
    SparseMatrix b(cols , rows , terms); // capacity of b.smArray is terms
    if (terms > 0)
    {// nonzero matrix
        int currentB = 0;
        for (int c = 0 ; c < cols ; c++)
            for (int i = 0 ; i < terms ; i++)
            // search and move the items 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;
                }
    } // if (terms > 0) terminate
    return b;
}
```

columns

terms

Scan the array “columns” times.
The array has “terms” elements.

==> O(columns*terms)

The Sparse Matrix ADT (11/23)

- Discussion: compared with 2-D array representation
  - O(columns*terms) vs. O(columns*rows)
  - elements --> columns * rows when non-sparse, O($columns^2$*rows)
- Problem: Scan the array "columns" times.
  - We can transpose a matrix represented as a sequence of triples in O(columns + terms) time.
- Solution: Fast Matrix Transposing

1. Determine the number of elements in each column of the original matrix.
2. Determine the starting positions of each row in the transpose matrix.

The Sparse Matrix ADT (12/23)

- **Fast Matrix Transposing**
  - Store some information to avoid scanning all terms back and forth
  - **FastTranspose** requires more space than **Transpose**
    - RowSize
    - RowStart

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

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

index	[0]	[1]	[2]	[3]	[4]	[5]
RowSize =	3	2	1	0	1	1
RowStart =	0	3	5	6	6	7

- Calculate RowSize by scanning array *a*
- Calculate RowStart by scanning RowSize

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

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

index	[0]	[1]	[2]	[3]	[4]	[5]
RowSize =	3	2	1	0	1	1
RowStart =	0	3	5	6	6	7

RowStart[0]++

The Sparse Matrix ADT (15/23)

	row	col	value
b[0]	0	0	15
[1]			
[2]			
[3]			
[4]			
[5]			
[6]	4	0	91
[7]			

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

index	[0]	[1]	[2]	[3]	[4]	[5]
RowSize =	3	2	1	0	1	1
RowStart =	1	3	5	6	6	7

RowStart[4]++

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

index	[0]	[1]	[2]	[3]	[4]	[5]
RowSize =	3	2	1	0	1	1
RowStart =	0	3	5	6	6	7

```
       row col value            row col value

b[0]   0   0    15       a[0]   0   0    15
 [1]   0   3    22        [1]   0   4    91
 [2]   0   5   -15        [2]   1   1    11
 [3]   1   1    11        [3]   2   1     3
 [4]   1   2     3        [4]   2   5    28
 [5]   2   3    -6        [5]   3   0    22
 [6]   4   0    91        [6]   3   2    -6
 [7]   5   2    28        [7]   5   0   -15

  index      [0][1][2][3][4][5]
 RowSize   =  3  2  1  0  1  1
 RowStart  =  0  3  5  6  6  7
```

The Sparse Matrix ADT (19/23)

```
SparseMatrix SparseMatrix::FastTranspose()
// The transpose of a(*this) is placed in b and is found in O(terms + columns) time.
{
 int *Rows = new int[Cols];
 int *RowStart = new int[Cols];
 SparseMatrix b;
 b.Rows = Cols; b.Cols = Rows; b.Terms = Terms;
 if (Terms > 0)          // nonzero matrix
 {
  // compute RowSize[i] = number of terms in row i of b
     for (int i = 0; i < Cols; i++) RowSize[i] = 0;
  // Initialize
  for ( i = 0; i < Terms; i++)
   RowSize[smArray[i].col]++;
  // RowStart[i] = starting position of row i in b
  RowStart[0] = 0;
  for (i = 1; i < Cols; i++)
   RowStart[i] = RowStart[i-1] + RowSize[i-1];
```

O(columns)

O(terms)

O(columns-1)

The Sparse Matrix ADT (20/23)

```
    for (i =0; i < Terms; i++)   // move from a 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
  } // end of if
  delete [] RowSize;
  delete [] RowStart;
  return b;
} // end of FastTranspose
```

O(terms)

O(columns+terms)

Recap…

addition

subtraction

transposing

multiplication

- Sparse Matrix Representation
  - Use triple <row, column, value>
  - Store triples row by row
  - For all triples within a row, their column indices are in ascending order.
  - Must know the numbers of rows and columns and the number of nonzero elements

$$
\begin{array}{c c}
 & \begin{array}{cccccc} \text{col } 0 & \text{col } 1 & \text{col } 2 & \text{col } 3 & \text{col } 4 & \text{col } 5 \end{array} \\
\begin{array}{c} \text{row } 0 \\ \text{row } 1 \\ \text{row } 2 \\ \text{row } 3 \\ \text{row } 4 \\ \text{row } 5 \end{array} &
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

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

The Sparse Matrix ADT(21/23)

- **Matrix multiplication**
  - Definition:

    Given *A* and *B* where *A* is $m \times n$ and *B* is $n \times p$, the product matrix *D* has dimension $m \times p$. Its $< i, j >$ element is

    $$\boxed{d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}}$$

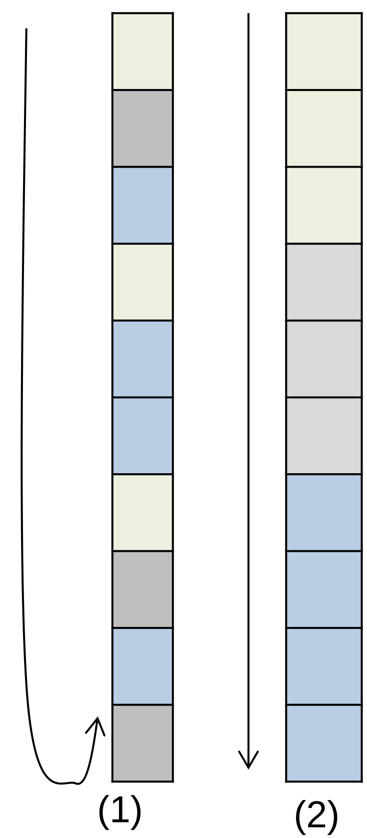
    for $0 \leq i < m$ and $0 \leq j < p$.
  - Example:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

The Sparse Matrix ADT (22/23)

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

- **Sparse Matrix Multiplication**
  - **Definition**: $D_{m \times p} = A_{m \times n} * B_{n \times p}$
  - **Procedure**: Fix a row *j* of A and find all terms in column *j* of *B* for $j = 0, 1, \ldots, p-1$.
  - **Method 1.**
    Scan all of *B* to find all elements in *j*.
  - **Method 2.**
    Compute the transpose of *B first*.
    (Put all column elements consecutively)
    - Once we have located the elements of row *i* of *A* and column *j* of *B* we just do a **merge operation** similar to that used in the polynomial addition of 2.2



2D Arrays

The elements of a 2-dimensional array a declared as:

```
int [][]a = new int[3][4];
```

may be shown as a table

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Rows Of A 2D Array

a[0][0]	a[0][1]	a[0][2]	a[0][3]	row 0
a[1][0]	a[1][1]	a[1][2]	a[1][3]	row 1
a[2][0]	a[2][1]	a[2][2]	a[2][3]	row 2

Columns Of A 2D Array

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]
column 0	column 1	column 2	column 3

2D Array Representation In C++

2-dimensional array x

a, b, c, d

e, f, g, h

i, j, k, l

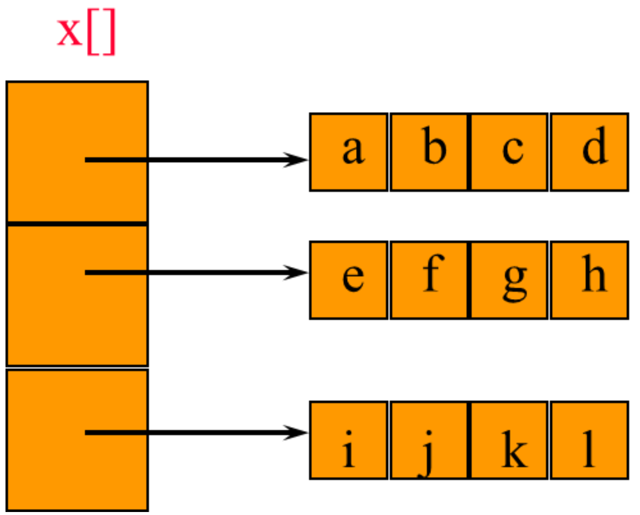
view 2D array as a 1D array of rows

x = [row0, row1, row 2]

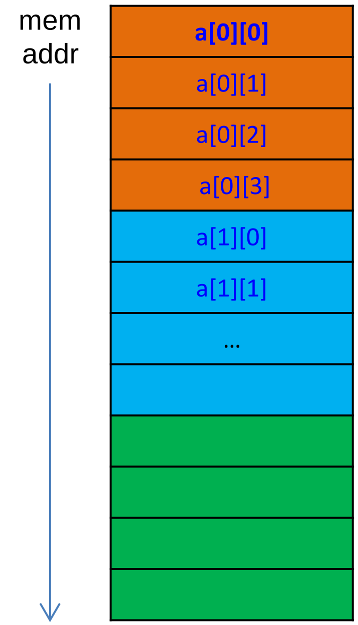
row 0 = [a,b, c, d]

row 1 = [e, f, g, h]

row 2 = [i, j, k, l]



int [][]a = new int[3][4];



Row-Major Mapping

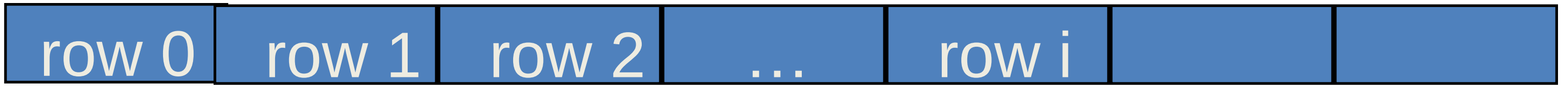
- Example 3 x 4 array:

a b c d

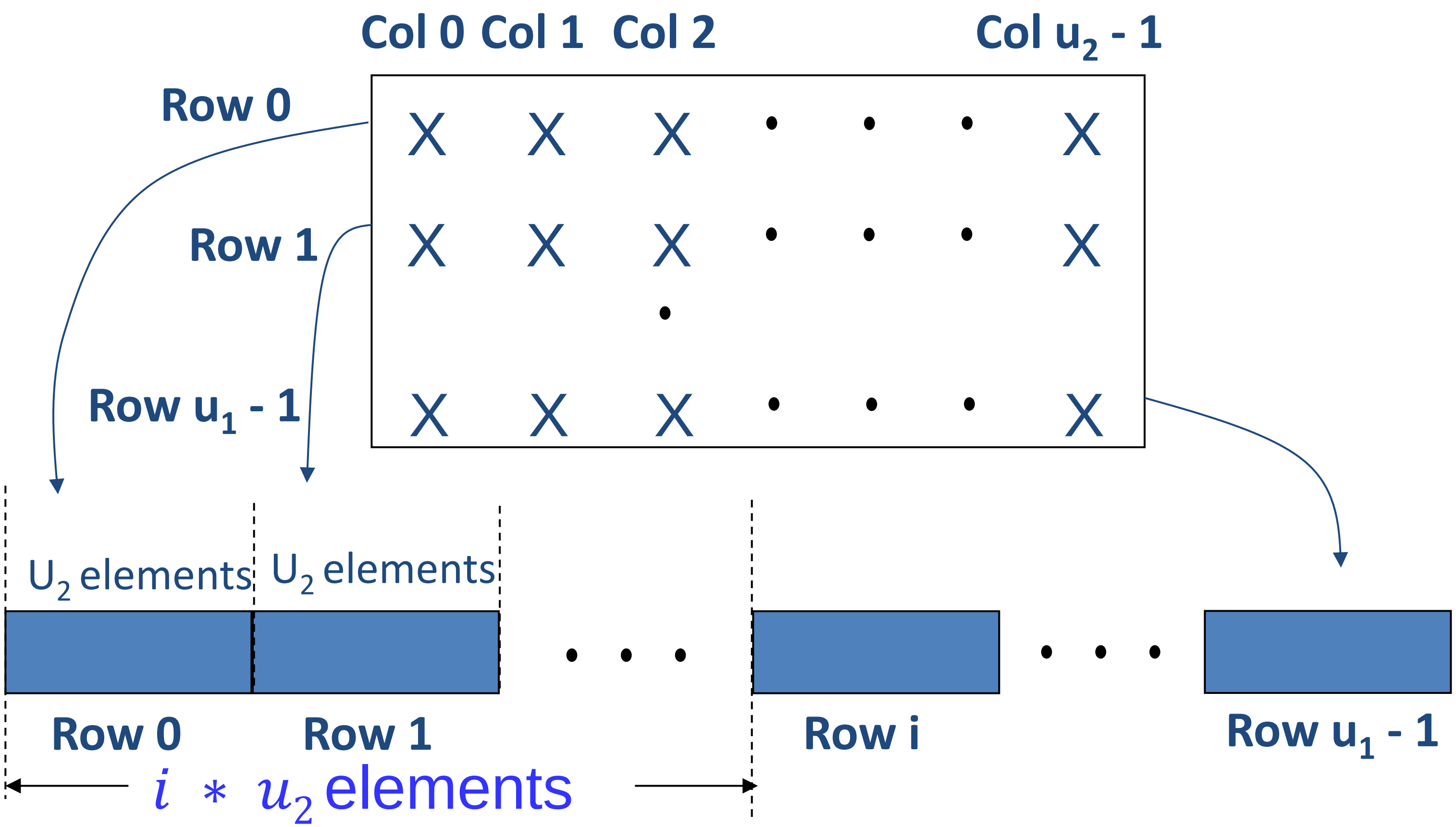
e f g h

i j k l

- Convert into 1D array y by collecting elements by rows.
- Within a row elements are collected from left to right.
- Rows are collected from top to bottom.
- We get y[] = {a, b, c, d, e, f, g, h, i, j, k, l}



Two Dimensional Array Row Major Order



Representation of Arrays (3/5)

- Row major order: $A[i][j]:\ \alpha + i * u_1 + j$

	col_0	col_1		col_{u1-1}
row_0	A[0][0]	A[0][1]	. . .	A[0][u1-1]
	α			
row_1	A[1][0]	A[1][1]	. . .	A[1][u1-1]
	$\alpha + u_1$			
		. . .		
row_{u0-1}	A[u0-1][0]	A[u0-1][1]	. . .	A[u0-1][u1-1]
	$\alpha+(u_0-1)*u_1$			

Locating Element x[i][j]

- assume x has u_0 rows and u_1 columns
- each row has u_1 elements
- i rows to the left of row i
- so $i \times u_1$ elements to the left of x[i][0]
- so $x[i][j]$ is mapped to position

  $i * u_1 + j$ of the 1D array

Column-Major Mapping

a b c d

e f g h

i j k l

- Convert into 1D array y by collecting elements by columns.
- Within a column elements are collected from top to bottom.
- Columns are collected from left to right.
- We get y = {a, e, i, b, f, j, c, g, k, d, h, l}

Representation of Multidimensional Arrays

- The internal representation of multidimensional arrays requires more complex addressing formula.
  - If an array is declared $a[u_0][u_1] \dots [u_n]$, then it is easy to see that the number of elements in the array is:

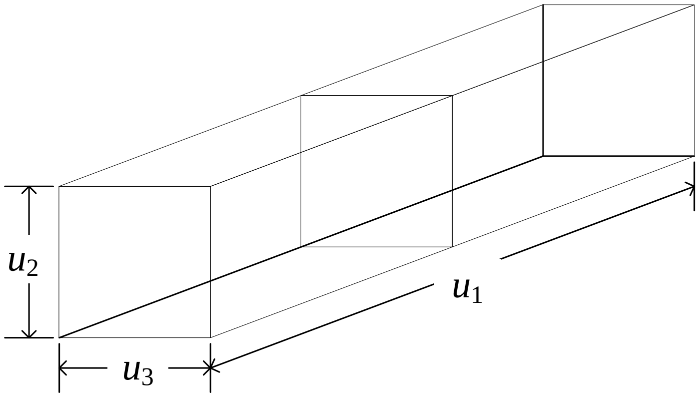
    $$\prod_{i=0}^{n-1} u_i$$

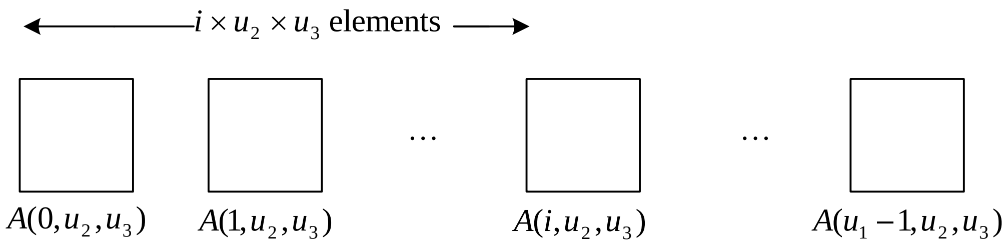
    Where Π is the product of the u_i's.

  - Example:
    - If we declare a as $a[10][10][10]$, then we require $10 * 10 * 10 = 1000$ units of storage to hold the array.

Representation of Multidimensional Arrays

- To represent a three-dimensional array, $A[u_0][u_1][u_2]$, we interpret the array as u_0 two-dimensional arrays of dimension $u_1 \times u_2$.
  - To locate $a[i][j][k]$, we first obtain $\alpha + i*u_1*u_2$ as the address of $a[i][0][0]$ because there are i two dimensional arrays of size u_1*u_2 preceding this element.
  - $\alpha + i*u_1*u_2+j*u_2+k$ as the address of $a[i][j][k]$.





Representation of Multidimensional Arrays

- Generalizing on the preceding discussion, we can obtain the addressing formula for any element $A[i_0][i_1]\ldots[i_{n-1}]$ in an n-dimensional array declared as: $A[u_0][u_1]\ldots[u_{n-1}]$
  - The address for $A[i_0][i_1]\ldots[i_{n-1}]$ is:

$$
\begin{aligned}
&\alpha + i_0 u_1 u_2 \ldots u_{n-1} \\
&\quad + i_1 u_2 u_3 \ldots u_{n-1} \\
&\quad + i_2 u_3 u_4 \ldots u_n \\
&\quad : \\
&\quad : \\
&\quad + i_{n-2} u_{n-1} \\
&\quad + i_{n-1}
\end{aligned}
\qquad = \alpha + \sum_{j=0}^{n-1} i_j a_j \quad \text{where} \quad
\begin{cases}
a_j = \prod_{k=j+1}^{n-1} u_k & 0 \le j \le n-1 \\
a_{n-1} = 1
\end{cases}
$$

The String ADT

- The String: component elements are characters.
  - A string to have the form, $S = s_0, \ldots, s_{n-1}$, where s_i are characters taken from the character set of the programming language.
  - If $n = 0$, then S is an empty or null string.
  - Operations in ADT String

The String Abstract data type(2/19)

- ADT *String*:

```
class String
{
public:
    String(char *init, int m);
    // 建構子：將 *this 初始化為長度為 m 的字串 init。

    bool operator = = (String t);
    // 如果 *this 所表示的字串等於 t，回傳 true；否則回傳 false。

    bool operator!( );
    // 如果 *this 是空字串，回傳 true； 否則回傳 false。

    int Length( );
    // 回傳 *this 裡的字元數。

    String Concat(String t);
    // 回傳一個字串，它的內容是字串 *this 後接著字串 t。

    String Substr(int i, int j);
    // 如果這些位置在 *this 裡是有效的，那麼回傳 *this 裡的第 i, i＋1, ..., i
＋j－1
    // 共 j 個字元的子字串；否則丟出一個例外。

    int Find(String pat);
    // 回傳 pat 在 *this 裡的開始位置 i；
    // 如果 pat 是空字串或者 pat 不是 *this 的子字串則回傳－1。
```

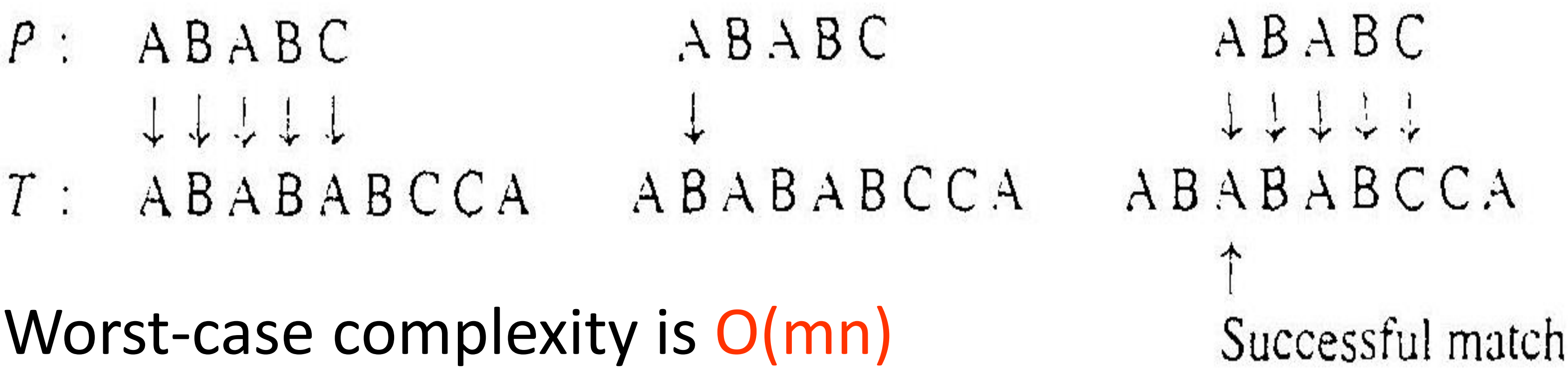
String

- Usually string is represented as a character array.
- General string operations include comparison, string concatenation, copy, insertion, string matching, printing, etc.

H	e	l	l	o		W	o	r	l	d	\0

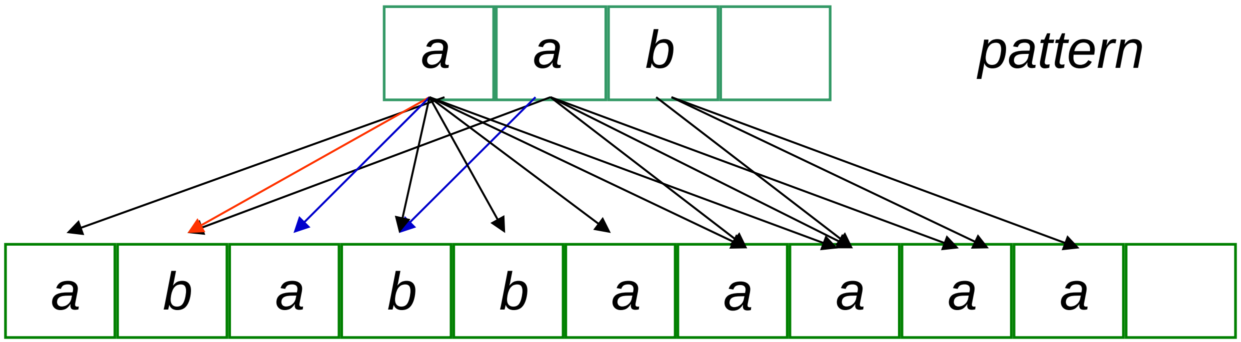
String Pattern Matching

- **Algorithm: Simple string matching**
- **Input**: *P* and *T*, the pattern and text strings; *m*, the length of *P*. The pattern is assumed to be nonempty.
- **Output**: The return value is the index in *T* where a copy of *P* begins, or -1 if no match for *P* is found.



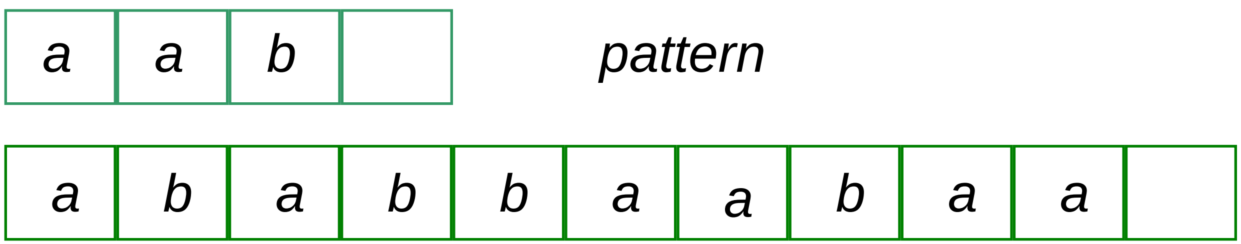
- Worst-case complexity is O(mn)

A simple algorithm



pattern

no match



pattern

match

O(n*m)

Exhaustive Pattern Matching

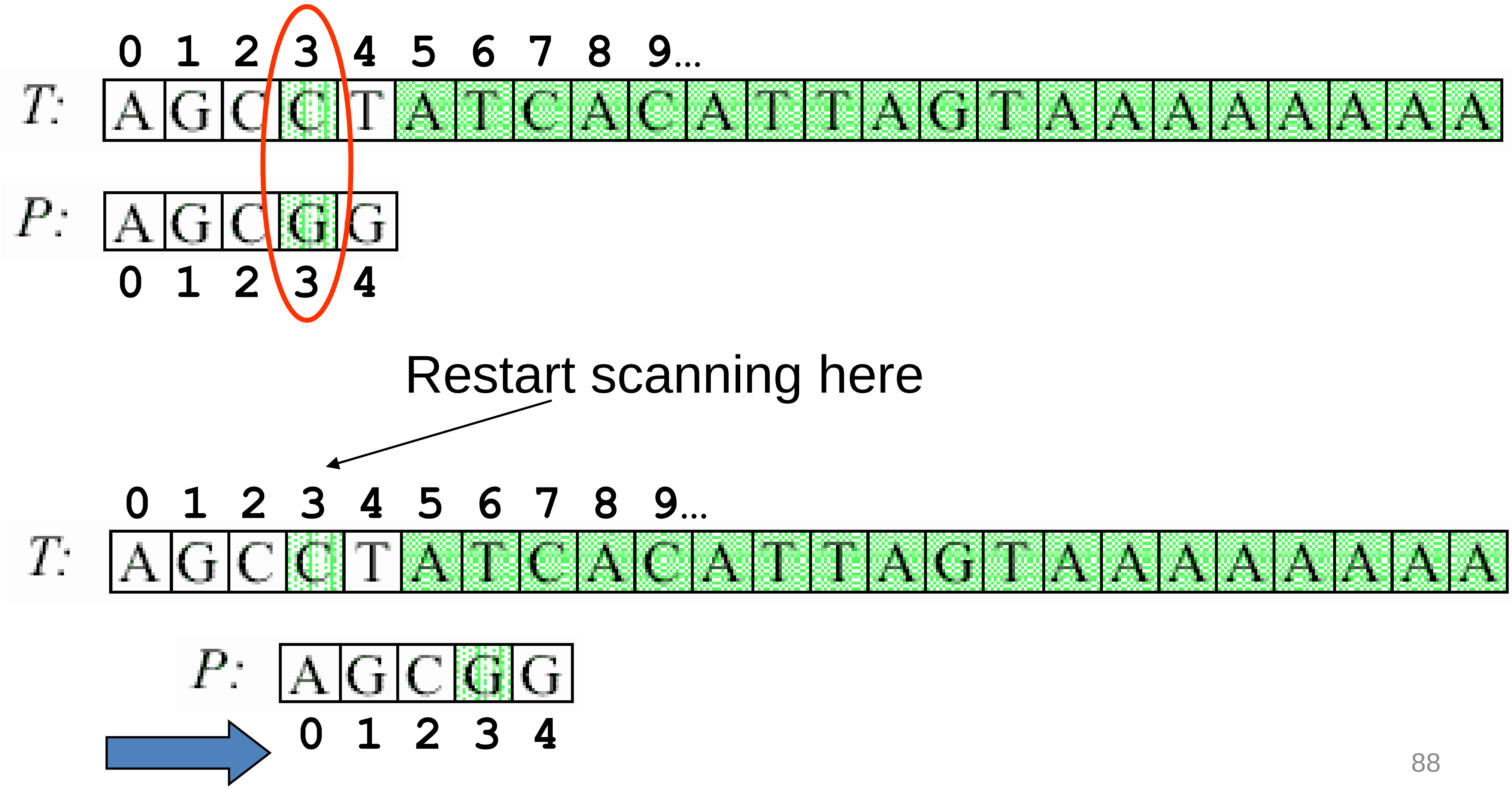
```
int String::Find(String pat)
{// if pat cannot be found in *this, return -1; otherwise, return the starting address of patin *this
        for (int start = 0; start <= Length( )－pat.Length(); star ++)
        { // start from str [start] to find matched character
                int j;
                for (j = 0; j < pat.Length( ) && str [start＋j] = = pat.str[j]; j++)
                if (j = = pat.Length( )) return start; //  找到相同的字串
                // not match at start
        }
        return  －1 ; // pat is an empty string or pat doesn't exist in s
}
```

O(lengthP × lentghS)

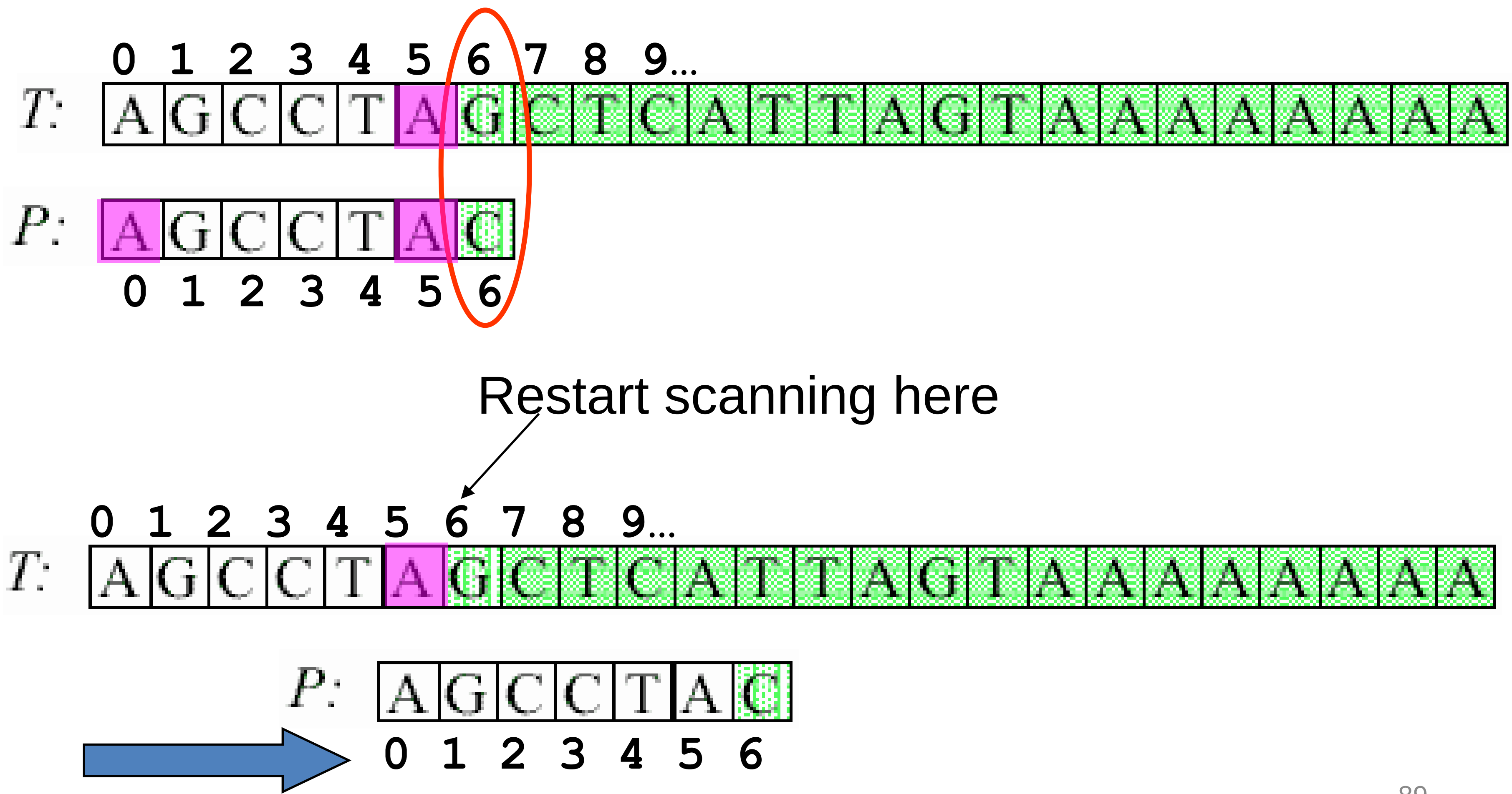
KMP Algorithm

- KMP algorithm
  - Proposed by **Knuth, Morris and Pratt**
- Concept
  - Use the characteristic of the pattern string
- Phase 1:
  - Generate an array to indicate the moving direction.
- Phase 2:
  - Use the array to move and match string

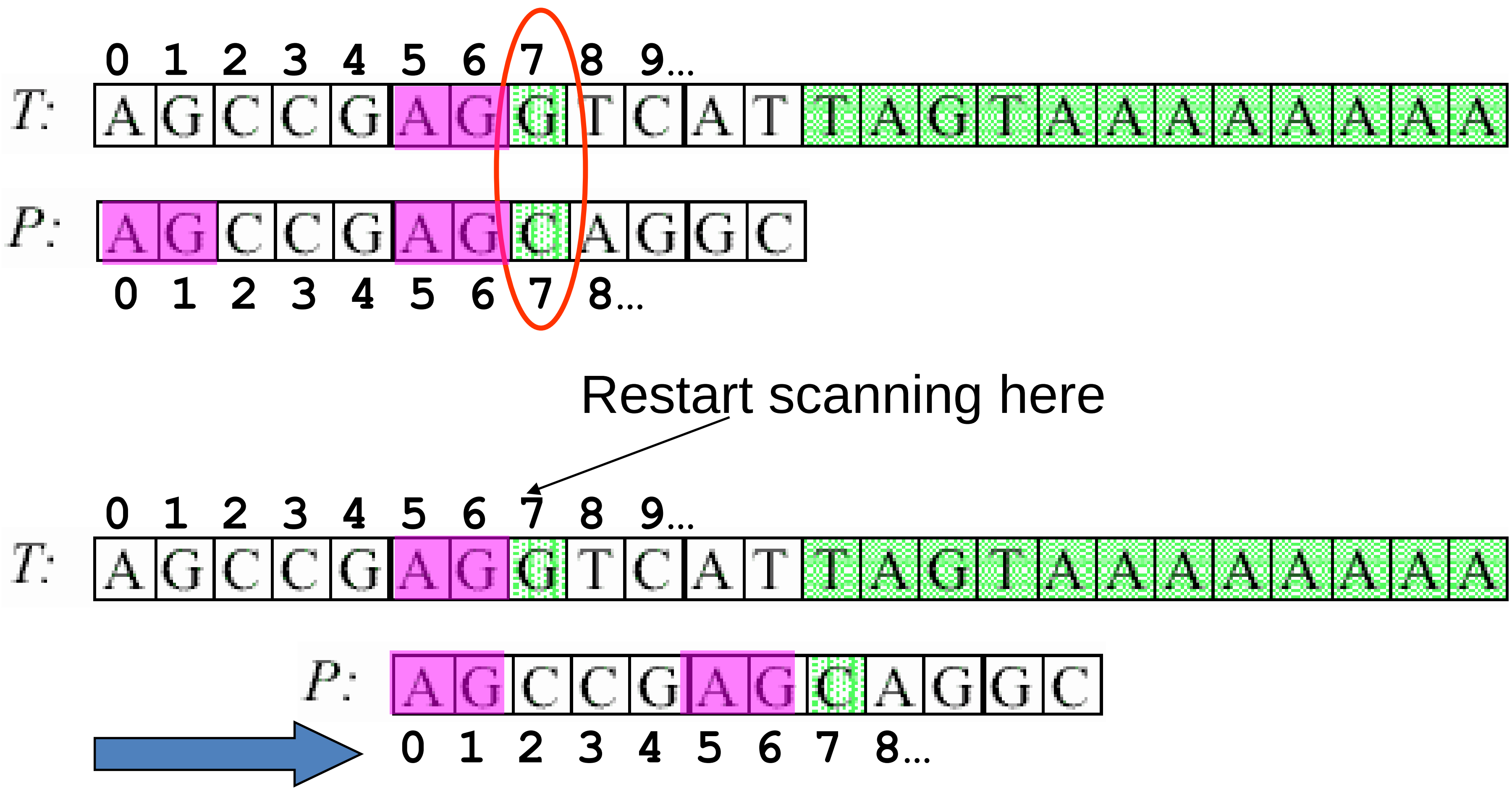
The First Case for the KMP Algorithm



The Second Case for the KMP Algorithm



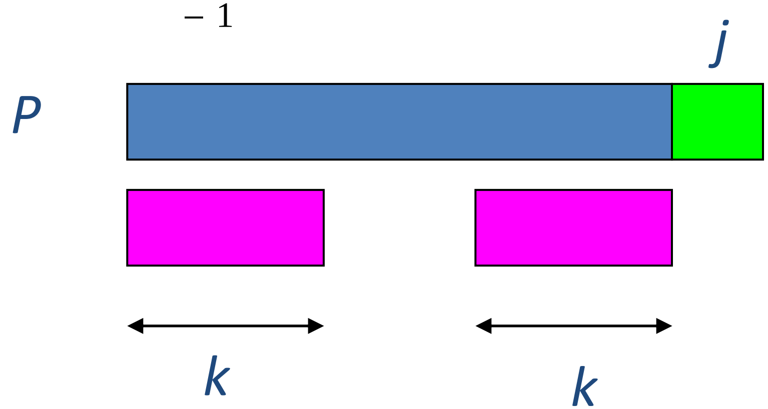
The Third Case for the KMP Algorithm



KMP Algorithm (cont'd)

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

$$f(j) = \begin{cases} \text{largest } k < j \text{ such that } p_0 p_1 \cdots p_k = p_{j-k} p_{j-k+1} \cdots p_j & \text{if such a } k \geq 0 \text{ exists} \\ -1 & \text{otherwise} \end{cases}$$



- 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 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.

Fast Matching Example: Failure Function Calculation

The largest k such that
1. $k < j$
2. $k \geq 0$
3. $p_0 p_1 \ldots p_k = p_{j-k} p_{j-k+1} \ldots p_j$

- $j = 0$
  - Since $k < 0$ and $k \geq 0$, no such k exists
  - $f(0) = -1$
- $j = 1$
  - Since $k < 1$ and $k \geq 0$, k may be 0
  - When $k = 0$ $p_0 = a$ and $p_1 = b$ $\neq$
  - $f(1) = -1$

j	0	1	2	3	4	5	6	7	8	9
p	a	b	c	a	b	c	a	c	a	b
f	-1	-1								

Fast Matching Example: Failure Function Calculation (contd.)

The largest k such that
1. $k < j$
2. $k \geq 0$
3. $p_0 p_1 \ldots p_k = p_{j-k} p_{j-k+1} \ldots p_j$

- $j = 2$
  - Since $k < 2$ and $k \geq 0$, k may be 0, 1
  - When $k = 1$ $p_0 p_1 = ab$ and $p_1 p_2 = bc$ $\neq$
  - When $k = 0$ $p_0 = a$ and $p_2 = c$ $\neq$
  - $f(2) = -1$

j	0	1	2	3	4	5	6	7	8	9
p	a	b	c	a	b	c	a	c	a	b
f	-1	-1	-1							

k=0

k=1

Fast Matching Example: Failure Function Calculation (contd.)

- $j = 4$
  - Since $k < 4$ and $k \geq 0$, k may be 0, 1, 2, 3
  - When $k = 3$ $p_0 p_1 p_2 p_3 = abca$ and $p_1 p_2 p_3 p_4 = bcab$ $\neq$
  - When $k = 2$ $p_0 p_1 p_2 = abc$ and $p_2 p_3 p_4 = cab$ $\neq$
  - When $k = 1$ $p_0 p_1 = ab$ and $p_3 p_4 = ab$ $=$
  - When $k = 0$ $p_0 = a$ and $p_4 = b$ $\neq$
  - $f(4) = 1$

j	0	1	2	3	4	5	6	7	8	9
p	a	b	c	a	b	c	a	c	a	b
f	-1	-1	-1	0	1					

Fast Matching Example: Failure Function Calculation (contd.)

- A restatement of failure function
- $f(j) =$
  - -1 if $j = 0$
  - $f^m(j-1)+1$ where *m* is the least integer *k* for which

    $$p_{f^k(j-1)+1} = p_j$$

  - *-1* if there is no k satisfying the above

$f^1(j) = f(j)$ and $f^m(j) = f(f^{m-1}(j))$

Failure Function

```
void String::FailureFunction()
{// 為字串樣本 *this 計算失敗函數。
    int lengthP = Length( );
    f[0] = -1;
    for (int j = 1; j < lengthP; j++) // 計算 f[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;
    }
}
```

$O(lengthP)$

Pattern-matching with a Failure Function

```
int String::FastFind(String pat)
{// 決定 pat 是否為 s 的子字串。
    int posP = 0, posS = 0;
    int lengthP = pat.Length(), lengthS = Length();
    while((posP < lengthP) && (posS < lengthS))
        if (pat.str[posP] = = str[posS]) {// 匹配到相同的字元
            posP++; posS++;
        }
        else
            if (posP = = 0)
                posS++;
            else posP = pat . f [posP－1] + 1;
    if (posP < lengthP) return －1;
    else return posS－lengthP;
}
```

$O(lengthS)$

Fast Matching Example: String Matching

j	0	1	2	3	4	5	6	7	8	9
p	a	b	c	a	b	c	a	c	a	b
f	-1	-1	-1	0	1	2	3	-1	0	1

2: check failure function ($f(3)$)

```
s =   a b c a ? ? . . . ? ? ? ?
p =   a b c a b c a c a b
```

3: move pattern accordingly

1: fail at $PosP = 4$

```
            a b c a b c a c a b
```

line 12: $PosP = pat.f[PosP - 1] + 1$

Fast Matching Example: String Matching (contd.)

j	0	1	2	3	4	5	6	7	8	9
p	a	b	c	a	b	c	a	c	a	b
f	-1	-1	-1	0	1	2	3	-1	0	1

s =   a b c a | ? ? . . . ? ? ? ?

p =   a b c a | b c a c a b

$\neq$

~~a b c | a b c a c a b~~

~~a b | c a b c a c a b~~

a | b c a b c a c a b

The Analysis of the KMP Algorithm

- O(m+n)
  - O(lengthP) for computing function f
    - Program 2.17
  - O(lengthS) for searching P
    - Program 2.16