Arrays

Fan-Hsun Tseng
Department of Computer Science and Information Engineering
National Cheng Kung University

Outline

- The array as an Abstract Data Type
- The polynomial Abstract Data Type
- The sparse matrix Abstract Data Type
- Representation of Arrays
- The string Abstract Data Type

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.

Array as an Abstract Data Type

- Example: int list[5]
  - list[0], …, list[4] each contains an integer

	list[0]	list[1]	list[2]	list[3]	list[4]
Memory address	base address = α	α + sizeof(int)	α + 2*sizeof(int)	α + 3*sizeof(int)	α + 4*sizeof(int)
Integer_Value	Integer_Value_1	Integer_Value_2	Integer_Value_3	Integer_Value_4	Integer_Value_5

class GeneralArray {

/* objects: 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.

For example, {0, …, 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. */

public:

GeneralArray(**int** *j*; *RangeList list*, **float** *init*Value = 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 each index i in the index set, insert <i, initValue> into the array. */

float Retrieve(index i);

/* if (i is in the index set of the array) return the float associated with i in the array; else signal an error */

void Store(index i, **float** x);

/* if (i is in the index set of the array) delete any pair of the form <i, y> the array and insert the new pair <i, y present in x>; else signal an error. */

}; // end of GeneralArray

Ordered List

- Ordered (linear) list
  - ($item_1$, $item_2$, $item_3$, …, $item_n$)
- Examples:
  - (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday)
  - (2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace)
  - (1941, 1942, 1943, 1944, 1945)
  - (a_1, a_2, a_3, …, a_{n-1}, a_n)

Operations on Ordered List

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

Implementation on Ordered List

- Implementing ordered list by array
  - Sequential mapping
  - (1)~(4) O
  - (5)~(6) X

	0	1	2	3	4
list					

- Performing operations 5 and 6 requires data movement
  - Costly
- This overhead motivates us to consider non-sequential mapping of order lists in Chapter 4
  - Linked list

Outline

- The array as an Abstract Data Type
- The polynomial Abstract Data Type
- The sparse matrix Abstract Data Type
- Representation of Arrays
- The string Abstract Data Type

Polynomial

- Example:
  - $A(X)=3X^2+2X+4$, $B(X)=X^4+10X^3+3X^2+1$
- The largest exponent of a polynomial is called is degree
- A polynomial is called sparse when it has many zero terms
- Implement polynomials by arrays

class polynomial

{

// objects: $p(x) = a_1x^{e1} + \ldots + a_nx^{en}$ 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 integer exponent

public:

Polynomial();

// Construct the polynomial $p(x) = 0$

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

Polynomial Representation #1

private:

int *degree*; // degree ≤ MaxDegree

float coef [MaxDegree + 1]; // coefficient array

$$X^4+10X^3+3X^2+1$$

	0	1	2	3	4
CoeffArray	1	0	3	10	1

- Need to know the maximum degree of the polynominals (*MaxDegree*)
- Waste space when the degree of the polynomial is much smaller than *MaxDegree*
  - Most of the positions in the array (*coef*[]) are unused

Polynomial Representation #2

```
private:
        int degree;
        float *coef;
   // and adding the following constructor to Polynomial
Polynomial::Polynomial(int d)
{
        degree = d;
        coef = new float[degree+1];
}
```

- By defining *coef* so that its size is degree+1
- Waste space when the polynomial is sparse (e.g., $x^{1000}+1$)

Polynomial Representation #3

- Use one global array to store all polynomials
  - $A(X)=2X^{1000}+1$
  - $B(X)=X^4+10X^3+3X^2+1$

	A.Start	*A.Finish*	*B.Start*			*B.Finish*	*free*
Coef	2	1	1	10	3	1	
Exp	1000	0	4	3	2	0	
Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*

Specification: polynomial	Representation: <start, finish>
A	<0,1>
B	<2,5>

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

class Polynomial; // forward declaration
  private:
    static term termArray[MaxTerms];
    static int free;
    int Start, Finish;

term Polynomial:: termArray[MaxTerms];
// location of next free location
// in termArray
int Polynomial::free = 0;
```

- Storage requirements: start, finish, 2*(finish-start+1)
- Non sparse: twice as much as representation 2 when all the items are nonzero

Adding Two Polynomials

	A.Start	*A.Finish*	*B.Start*			*B.Finish*	*free*
Coef	2	1	1	10	3	1	
Exp	1000	0	4	3	2	0	
Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*

a b

Coef							
Exp							
Index	*7*	*8*	*9*	*10*	*11*	*12*	*13*

```
Polynomial Polynomial:: Add(Polynomial B)
// return the sum of A(x) ( in *this) and B(x)
{
  Polynomial C; int a = Start; int b = B.Start; C.Start = free; float c;
    while ((a <= Finish) && (b <= B.Finish))
      switch (compare(termArray[a].exp, termArray[b].exp)) {
        case '=':
          c = termArray[a].coef +termArray[b].coef;
          if ( c ) NewTerm(c, termArray[a].exp);
            a++; b++;
            break;
         case '<':
            NewTerm(termArray[b].coef, termArray[b].exp);
            b++;
         case '>':
            NewTerm(termArray[a].coef, termArray[a].exp);
            a++;
    } // end of switch and while
    // add in remaining terms of A(x)
    for (; a<= Finish; a++)
    NewTerm(termArray[a].coef, termArray[a].exp);
    // add in remaining terms of B(x)
    for (; b<= B.Finish; b++)
      NewTerm(termArray[b].coef, termArray[b].exp);
    C.Finish = free 1;
    return C;
  } // end of Add
```

Analysis: O(n+m) where n and m is the number of non-zeros in A and B.

Adding a New Term

```
void Polynomial::NewTerm(float c, int e)
// Add a new term to C(x)
{
    if (free >= MaxTerms) {
        cerr << “Too many terms in polynomials"<< endl;
        exit();
    }
    termArray[free].coef = c;
    termArray[free].exp = e;
    free++;
} // end of NewTerm
```

Disadvantages of Representing Polynomials by Arrays

- The value of free is continually incremented until it tries to exceed *MaxTerms*
- What should we do when free is going to exceed *MaxTerms*?
  - Either quit or reuse the space of unused polynomials by compacting the global array
  - It is costly!
- A more elegant solution is proposed in Chapter 4 by employing linked list

Outline

- The array as an Abstract Data Type
- The polynomial Abstract Data Type
- The sparse matrix Abstract Data Type
- Representation of Arrays
- The string Abstract Data Type

Sparse Matrix

- Matrix → table of values

$$
\begin{array}{l}
\begin{matrix}
0 & 0 & 3 & 0 & 4 \\
0 & 0 & 5 & 7 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 2 & 6 & 0 & 0
\end{matrix}
\begin{matrix} \\ \text{Row 2} \\ \\ \\ \end{matrix} \\
\qquad\qquad \text{Column 4}
\end{array}
$$

4 x 5 matrix

4 rows

5 columns

20 elements

6 nonzero elements

Sparse Matrix (Contd.)

- A general matrix consists of *m* rows and *n* columns of numbers
  - An $m \times n$ matrix
  - It is natural to store a matrix in a two-dimensional array, say A[*m*][*n*]
- A matrix is called sparse if it consists of many zero entries
  - Implementing a spare matrix by a two-dimensional array waste a lot of memory
  - Space complexity is $O(m \times n)$

Sparse Matrix (Contd.)

$$\begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{bmatrix}$$

5x3

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

6x6

Sparse matrix

Figure 2.2 Two matrices (P.96)

Sparse Matrix Abastract Data Type

class *SparseMatrix*

/* objects: A set of triples, <row, column, value>, where row and column are integers, value is also an integer, and form a unique combinations */

public:

SparseMatrix(**int** *MaxRow*, **int** *MaxCol*);

/* the constructor function creates a SparseMatrix that can hold up to MaxInterms = MaxRow × MaxCol and whose maximum row size is MaxRow and whose maximum column size is MaxCol */

SparseMatrix Transpose();

/* returns the SparseMatrix obtained by interchanging the row and column value of every triple in ***this** */

Sparse Matrix Abastract Data Type (Contd.)

SparseMatrix Add(SparseMatrix b);

/* **if** the dimensions of a (*this) and b are the same, then the matrix produced by adding corresponding items, namely those with identical row and column values is returned

else error. */

SparseMatrix Multiply(SparseMatrix b);

/* **if** number of columns in a (***this**) equals number of rows in *b* then the matrix *d* produced by multiplying a by b according to the formula $d[i][j] = \Sigma(a[i][k] \cdot b[k][j])$, where $d[i][j]$ is the (i, j)th element, is returned. k ranges from 0 to the number of columns in $a - 1$

else error */

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

Sparse Matrix Representation (Contd.)

- Represented by a two-dimensional array.
  - Sparse matrix wastes space.
- Each element is characterized by <row, col, value>

	row	col	value
a[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

(a)

transpose

	row	col	value
b[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

(b)

row, column in ascending order

Sparse Matrix Representation (Contd.)

```
class SparseMatrix; // forward declaration
class MatrixTerm {
   friend class SparseMatrix
   private:
      int row, col, value;
};
In class SparseMatrix:
private:
   int Rows, Cols, Terms;
   MatrixTerm smArray[MaxTerms];
```

Transpose a Matrix

(1) 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)

(2) For all elements in column j,

place element <i, j, value> in element <j, i, value>

Transpose a Matrix (Contd.)

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

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

Transpose a Matrix (Contd.)

	row	col	value		row	col	value
CurrentB → a[0]	0	0	15	b[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

```
SparseMatrix SparseMatrix::Transpose()  // return the transpose of a (*this)
{
  SparseMatrix b;
  b.Rows = Cols; // rows in b = columns in a
  b.Cols = Rows; // columns in b = rows in a
  b.Terms = Terms; // terms in b = terms in a
  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 elements 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;
          CurrentB++;
        }
  } // end of if (Terms > 0)
  return b;
} // end of transpose
```

Time complexity O(terms*cols)

Compared with 2-Dimensional Array Representation

- Discussion:
  - O(columns $\times$ terms) vs. O(columns $\times$ rows)
  - Terms $\rightarrow$ columns $\times$ rows when non-sparse
  - O(columns^2 $\times$ rows) when non-sparse
- Problem: Scan the array “columns” times.
- Solution:
  - Determine the number of elements in each column of the original matrix.
  - Determine the starting positions of each row in the transpose matrix.

Fast Matrix Transposing

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

Fast Matrix Transposing (Contd.)

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

- Calculate RowSize by scanning array b
- Calculate RowStart by scanning RowSize

Fast Matrix Transposing (Contd.)

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

	row	col	value
b[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]++

Fast Matrix Transposing (Contd.)

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

	row	col	value
b[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]++

Fast Matrix Transposing (Contd.)

	row	col	value			row	col	value
a[0]	0	0	15		b[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

Fast Matrix Transposing (Contd.)

```
SparseMatrix SparseMatrix::Transpose()
// The transpose of a(*this) is placed in b and is found in Q(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;    O(columns)
        // Initialize
        for ( i = 0; i < Terms; i++)    O(terms)
            RowSize[smArray[i].col]++;
        // RowStart[i] = starting position of row i in b
        RowStart[0] = 0;
        for (i = 1; i < Cols; i++)    O(columns-1)
            RowStart[i] = RowStart[i-1] + RowSize[i-1];
```

Fast Matrix Transposing (Contd.)

```
    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)

Matrix Multiplication

- Definition: Given A and B, where A is $m \times n$ and B is $n \times p$, the product matrix Result has dimension $m \times p$. Its [i][j] element is

$$result_{i,j} = \sum_{k=0}^{n-1} a_{i,j} b_{i,j}$$

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

- Please study Section 2.4.4 by yourself

Outline

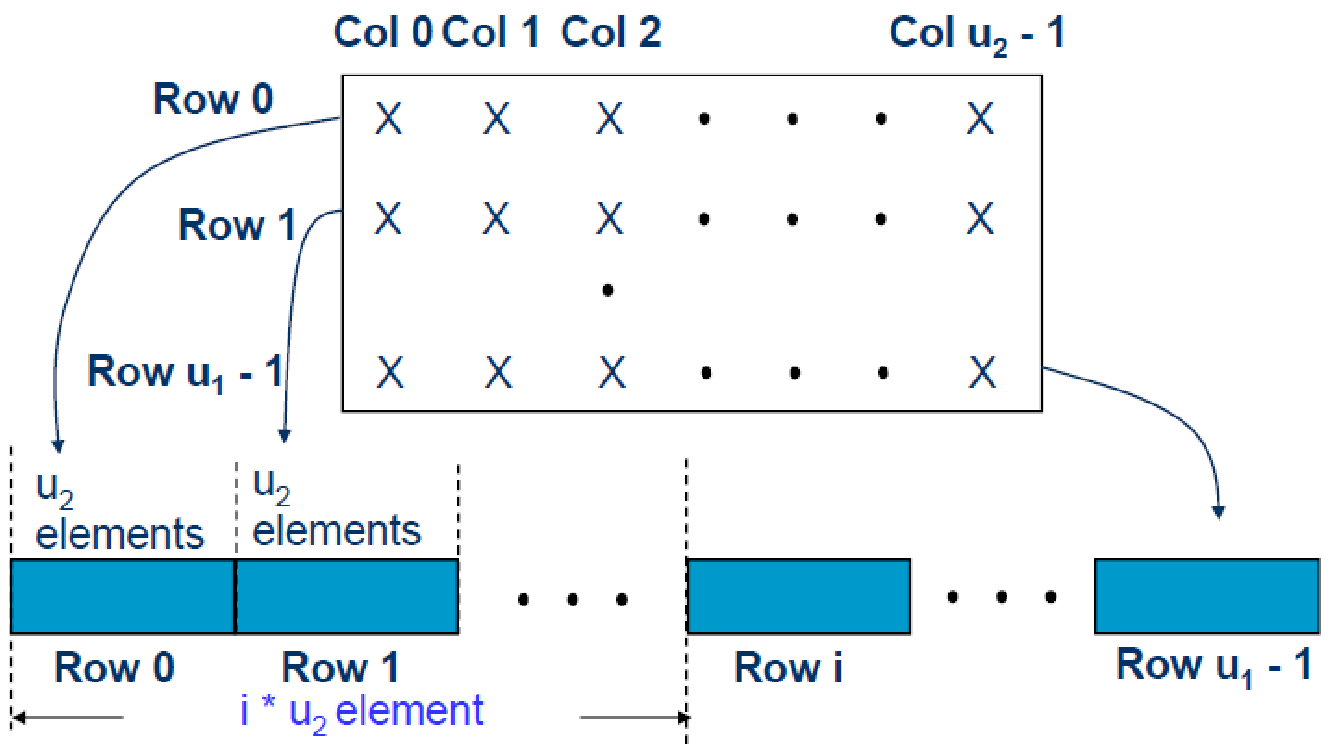
- The array as an Abstract Data Type
- The polynomial Abstract Data Type
- The sparse matrix Abstract Data Type
- Representation of Arrays
- The string Abstract Data Type

Representation of Arrays

- Multidimensional arrays are usually implemented by one dimensional array via either row major order or column major order.
- Example: One dimensional array

α	$\alpha+1$	$\alpha+2$	$\alpha+3$	$\alpha+4$
A[0]	A[1]	A[2]	A[3]	A[4]

Two Dimensional Array Row Major Order



Generalizing Array Representation

- The address indexing of Array $A[i_1][i_2],\ldots,[i_n]$ is

$$
\begin{aligned}
&\alpha + i_1 u_2 u_3 \ldots u_n \\
&\quad + i_2 u_3 u_4 \ldots u_n \\
&\quad + i_3 u_4 u_5 \ldots u_n \\
&\quad \vdots \\
&\quad + i_{n-1} u_n \\
&\quad + i_n \\
&= \alpha + \sum_{j=1}^{n} i_j a_j \quad where \begin{cases} a_j = \prod_{k=j+1}^{n} u_k & 1 \le j \le n \\ a_n = 1 \end{cases}
\end{aligned}
$$

Outline

- The array as an Abstract Data Type
- The polynomial Abstract Data Type
- The sparse matrix Abstract Data Type
- Representation of Arrays
- The string Abstract Data Type

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

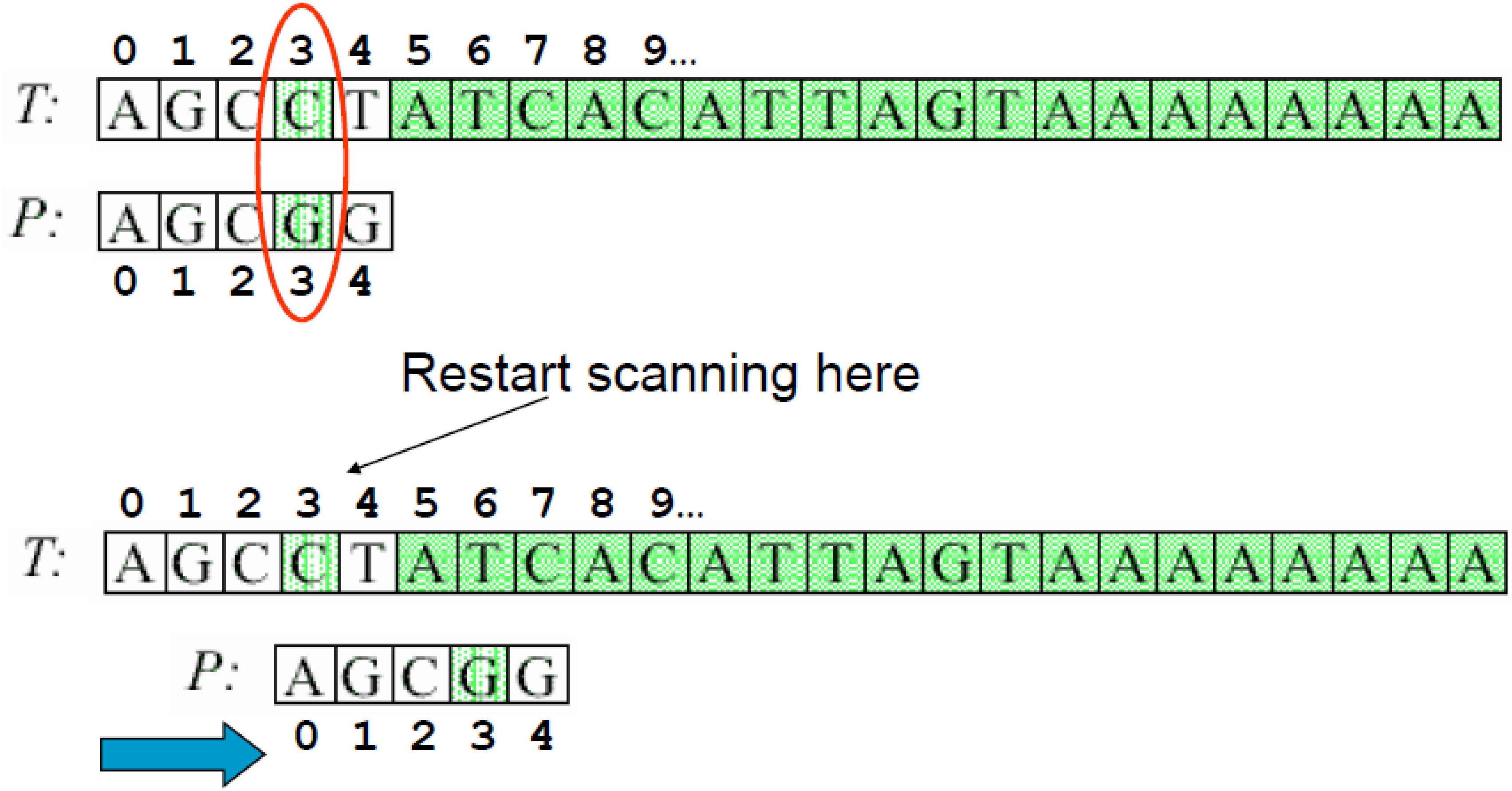
String Matching: Straightforward Solution

- **Algorithm: Simple string matching**
- **Input**: the pattern (*P*) and text strings (*T*), the length of P (m). 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 $\theta(mn)$

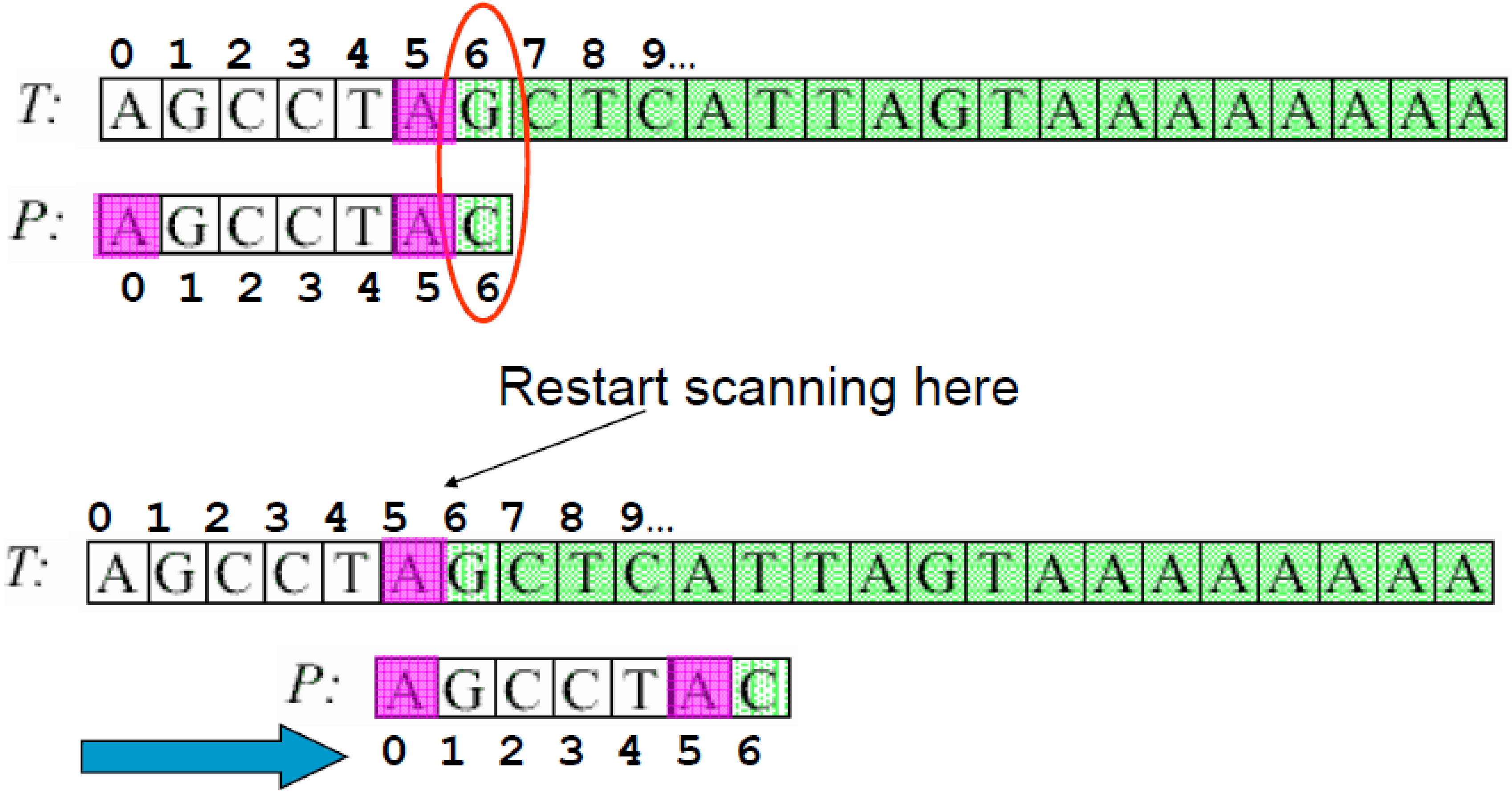
KMP Algorithm

- KMP (Knuth-Morris-Pratt) 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

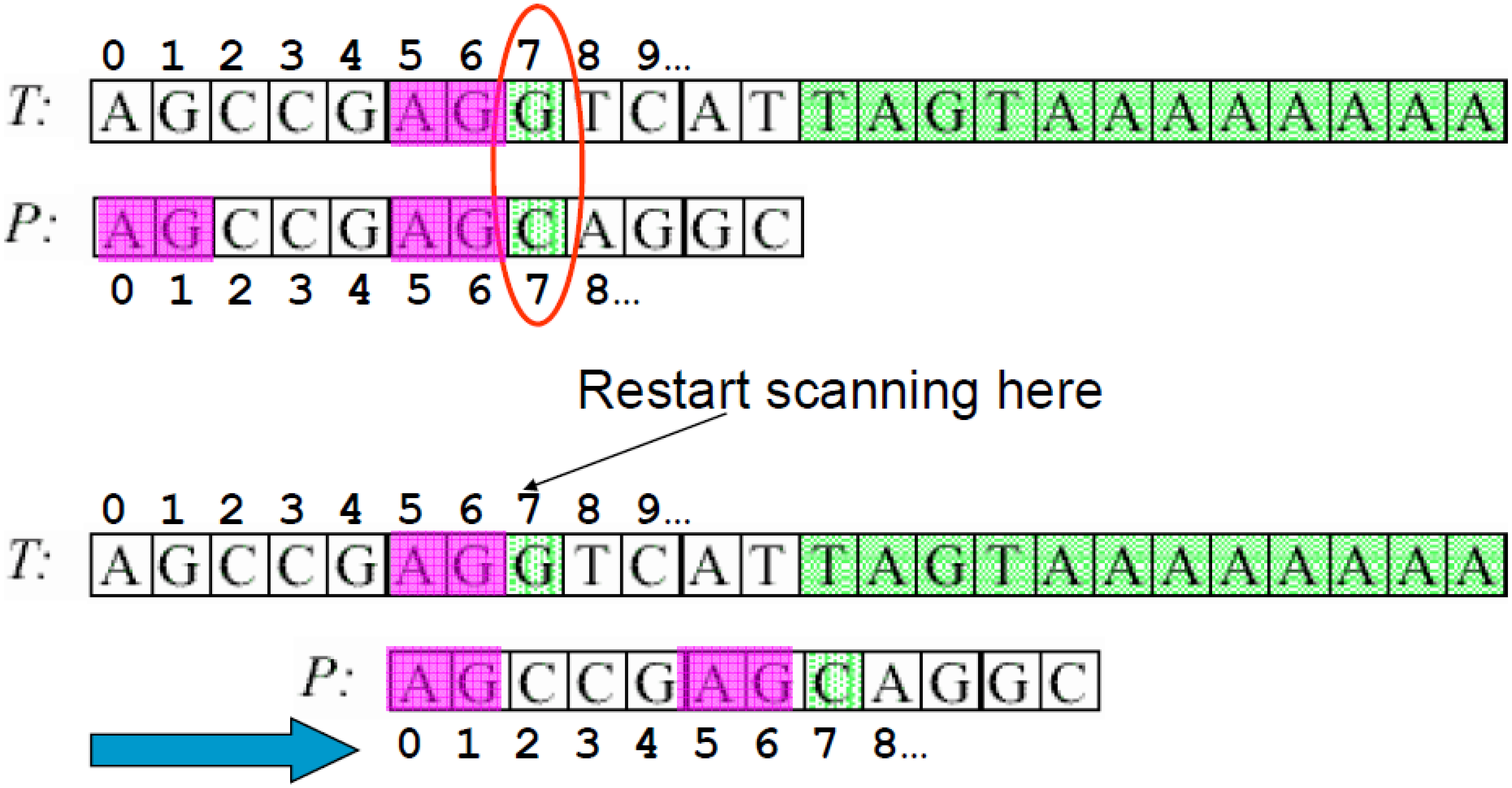
The First Case for the KMP Algorithm



The Second Case for the KMP Algorithm

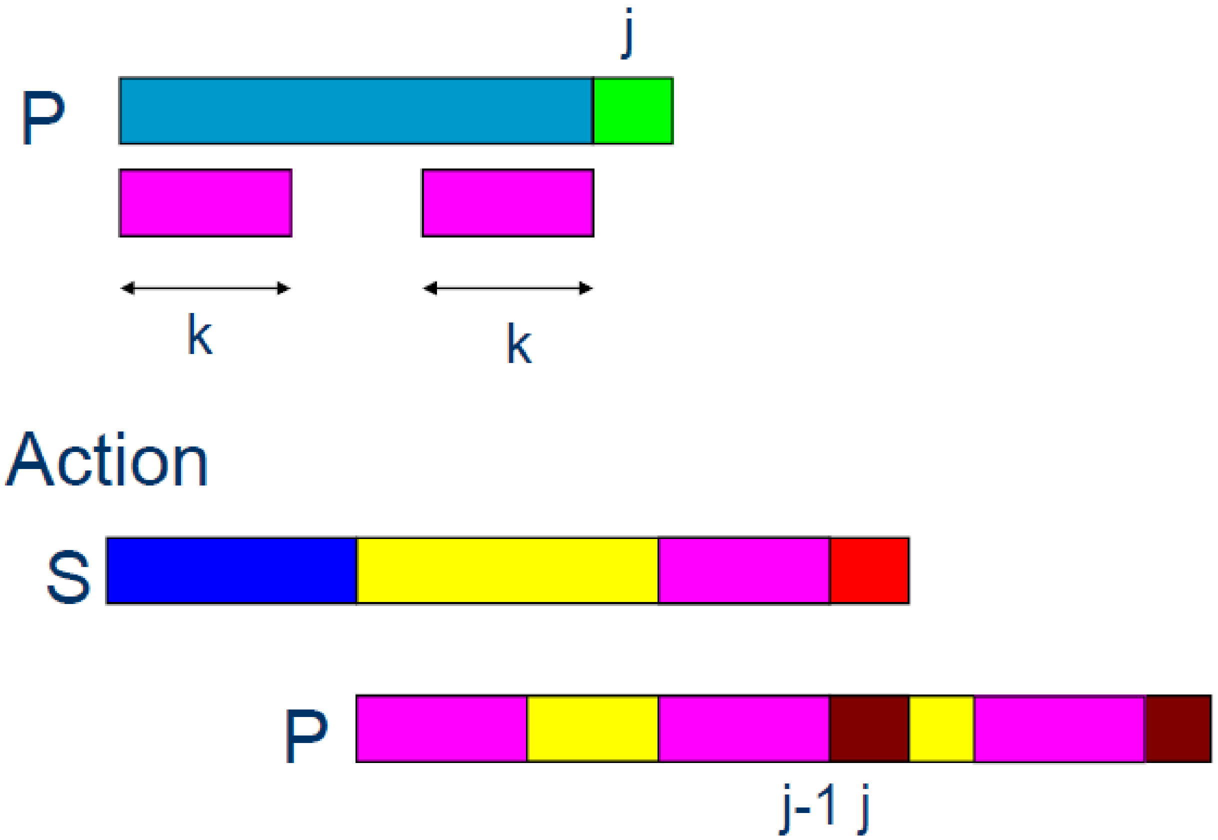


The Third Case for the KMP Algorithm



KMP Algorithm (Contd.)

- Failure Function



KMP Algorithm (Contd.)

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

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

- If a partial match is found such that $s_{i\text{-}j} \ldots s_{i\text{-}1} = p_0 p_1 \ldots p_{n\text{-}1}$ and $s_i \neq p_j$ then matching may be resumed by comparing s_i and $p_{f(j\text{-}1)+1}$, if $j \neq 0$.
- If $j = 0$ we may continue s, then by comparing s_{i+1} and p_0.

Fast Matching Example: Failure Function Calculation

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

The largest k such that
1. $k<j$
2. $k \geq 0$
3. $p_0p_1 \ldots p_k = p_{k-1}p_{j-k+1} \ldots p_j$

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

Fast Matching Example: Failure Function Calculation (Contd.)

- j=2
  - Since k<2 and k$\geqq$0, k may be 0,1
  - When k=1 $\rightarrow p_0p_1$=ab and p_1p_2=bc$\rightarrow$ **x**
  - When k=0 $\rightarrow$ p_0=a and p_2=c$\rightarrow$ **x**
  - 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	0	1	2	3	-1	0	1

k=0

k=1

Fast Matching Example: Failure Function Calculation (Contd.)

- j=4
  - Since k<4 and k≧0, k may be 0, 1, 2, 3
  - When k=3 → $p_0p_1p_2p_3$=abca and $p_1p_2p_3p_4$=bcab → **x**
  - When k=2 → $p_0p_1p_2$=abc and $p_2p_3p_4$=cab → **x**
  - When k=1 → p_0p_1=ab and p_3p_4=ab → **ok**
  - When k=0 → p_0=a and p_4=b → **x**
  - 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	2	3	-1	0	1

Fast Matching Example: String Matching

- A restatement of failure function
- $f(j) = \begin{cases} -1 & \text{if } j = 0 \\ f^m(j-1)+1, & \text{where } m \text{ is the least integer } k \text{ for withich } p_{f^m(j-1)+1} = p_j \\ -1 & \text{if there is no } k \text{ satisfying the above} \end{cases}$

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

2: check failure function (f(3))

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

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

1: fail at PosP=4

3: move pattern accordingly

a b c a b c a c a b

Program 2.15 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
              | x
        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
```

Imply f(3)=2 if this comparison is necessary

The Analysis of the KMP Algorithm

- O(m+n)
  - O(m) for computing function *f*
    - Program 2.16
  - O(n) for searching *P*
    - Program 2.15

- The *strstr* function in Linux kernel 2.4.22 is implemented by exhaustive search
  - Why?