CSIEB0100 Data Structures

Lecture 03 Arrays

Shiow-yang Wu 吳秀陽

Department of Computer Science and Information Engineering

National Dong Hwa University

Lecture material is partly home-grown, partly taken from slides came with the textbook originally prepared by Professor Jiun-Long Huang of NCTU.

Arrays

- **Array**: a set of values accessible by indexes
- Data structure
  - For each index, there is a value associated with that index.
- Representation (possible)
  - Implemented by using consecutive memory.
- Example: int list[5]: list[0], …, list[4] each contains an integer

	0	1	2	3	4
list					

```
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 initValue =
   defatultValue);
   /* 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> present in the array and insert the new pair <i,
   x>; else signal an error. */
}; // end of GeneralArray
```


Ordered List

- Ordered (linear) list
  - (item$_1$, item$_2$, item$_3$, …, item$_n$)
- Examples:
  - (MONDAY, TUEDSAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAYY, SUNDAY)
  - (2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace)
  - (2021, 2022, 2023, 2024, 2025)
  - (a_1, a_2, a_3, …, a_{n-1}, a_n)
- How to represent and implement ordered lists?

Operations on Ordered List

1. Find the length, n, of the list.
2. Read the items from left to right (or right to left).
3. Retrieve the i-th element.
4. Store a new value into the i-th position.
5. Insert a new element at the position i , causing elements numbered i, i+1, …, n to become numbered i+1, i+2, …, n+1 (**shift right**)
6. Delete the element at position i , causing elements numbered i+1, …, n to become numbered i, i+1, …, n-1 (**shift left**)

Implementation of Ordered List

- Implementing ordered lists by arrays
  - Sequential mapping
  - Operations (1)~(4) O
  - Operations (5)~(6) X
- Performing operations 5 and 6 requires data movement which is costly
- This overhead motivates us to consider nonsequential mapping of order lists in Chapter 4
  - Linked list

Array Application: 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 its degree
- A polynomial is called sparse when it has many zero terms
- Implement polynomials by arrays


```
class polynomial
{
objects: p(x) = a_1 x^{e_1} + ... + a_n x^{e_n}   a set of ordered pairs of <e_i, a_i>
where a_i ∈ Coefficient and e_i ∈ 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
```

Polynomial Representation #1

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

- $x^4+10x^3+3x^2+1$

	0	1	2	3	4
CoeffArray	1	0	3	10	1

- Waste space if the polynomial is much smaller than *MaxDegree*

Polynomial Representation #2

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

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

Polynomial Representation #3a

- Use one global array to store all polynomials
- Class member `free` gives the location of the next free location
  - $A(X)=2X^{1000}+1$ // highest exp first
  - $B(X)=X^4+10X^3+3X^2+1$

	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>

Polynomial Representation #3b

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

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

Polynomial Representation #3c

```
class Polynomial {
...
private:
  static Term termArray[MaxTerms];
  static int free;
  int Start, Finish;
}
Term Polynomial::termArray[MaxTerms];
// location of next free location
int Polynomial::free = 0;
```


Adding a New Term

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

Adding Two Polynomials

Polynomial #1
$x^4 - 3x^2 + x + 1$

Polynomial #2
$x^3 - x^2 + 5x - 2$

x^4	$+0$	$-3x^2$	$+x$	$+1$
	x^3	$-x^2$	$+5x$	-2
x^4	x^3	$-4x^2$	$+6x$	-1

Adding Two Polynomials on Array

	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 (points to index 0) b (points to index 2)

$A(X)=2X^{1000}+1$
$B(X)=X^4+10X^3+3X^2+1$

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 (m) is the number of non-zeros in A(B).

Disadvantages of Array Representation

- 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.

Sparse Matrix

- 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 Example

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

5×3

(a)

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

6×6

(b)

Sparse matrix

Sparse Matrix ADT

class SparseMatrix

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

public:

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

/ the constructor function creates a SparseMatrix that can hold up to MaxInterms = MaxRow x 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*** **/*

Sparxe Matrix ADT

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);*

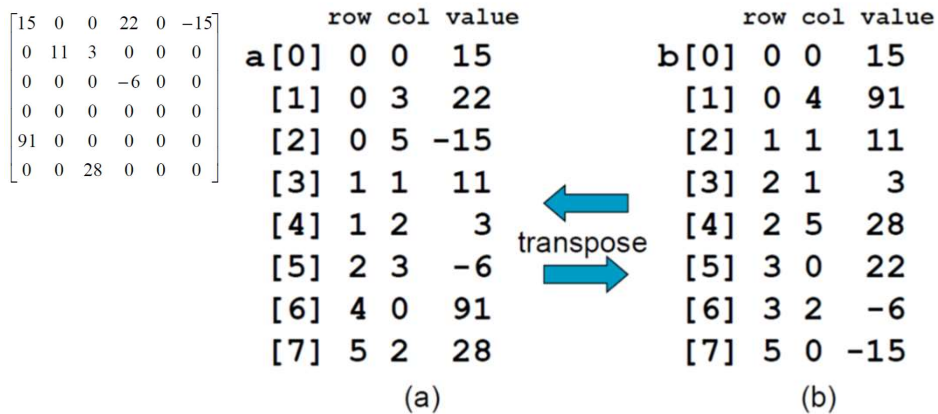
/ **if** number of columns in a (***this**) equals number of rows in b then the matrix d produced by multiplying a by b according to the formula d[i][j] = Σ(a[i][k]*b[k][j]), where d[i][j] is the (i, j)th element, is returned. k ranges from 0 to the number of columns in a – 1*

***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



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

	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)

	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


```
class SparseMatrix; // forward decl.
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)

    Move elements down and up very often.
- (2) For all elements in column j,
  - place element <i, j, value> in element <**j**, i, value>

Transpose a Matrix

	row	col	value
a[0]	0	0	15
[1]	0	3	22
[2]	0	5	-15
[3]	1	1	13
[4]	1	2	3
[5]	2	3	-6
[6]	4	0	91
[7]	5	2	28

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

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

Transpose a Matrix

	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	13 ← CurrentB
[3]	1	1	13	→	[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

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

Transpose a Matrix

		row	col	value			row	col	value
CurrentA →	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 b and process all entries with col = 0

Transpose a Matrix

	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
CurrentA → [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 b and process all entries with col = 1

Questions Worth Thinking

- Iteration 0: Scan all entries with col=0 to form row 0 of the transpose matrix.
- Iteration 1: Scan all entries with col=1 to form row 1 of the transpose matrix.
- …
- Rows are in ascending order.
- What about all col's within the same row? Will they be in ascending order?
- Entries are repeatedly scanned. Can we do better?

```
SparseMatrix SparseMatrix::Transpose()
// return the transpose of a (*this)
{
  SparseMatrix b;
  b.Rows = Cols; // rows in b = cols in a
  b.Cols = Rows; // cols in b = rows in a
  b.Terms = Terms; // terms in b = 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)

Comparison of Sparse and 2D

- 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
- Move elements to the correct positions directly.
- **`FastTranspose`** requires more space than **`Transpose`**
  - RowSize
  - RowStart
- An example of space-time tradeoff in algorithm design.

```
      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


```
      row col value          row col value
a[0]  0   0   15      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 =  0   0   0   0   0   0
        RowSize[0]++
```

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

index	[0]	[1]	[2]	[3]	[4]	[5]
RowSize =	1	0	0	0	0	0

RowSize[4]++

CSIEB0100 Data Structures Arrays 39

	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

RowStart[i] = RowStart[i-1]+RowSize[i-1]

CSIEB0100 Data Structures Arrays 40

```
SparseMatrix SparseMatrix::FastTranspose()
// The transpose of a(*this) is placed in b
// and is found in O(terms + columns) time.
{
  int i;
  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
  {
```



```
    for (int i = 0; i < Cols; i++) RowSize[i] = 0;
    // Initialize
    for (int i = 0; i < Terms; i++)
      RowSize[smArray[i].col]++;
    // RowStart[i] = start of row i in b
    RowStart[0] = 0;
    for (i = 1; i < Cols; i++)  // O(Cols – 1)
      RowStart[i] = RowStart[i-1] + RowSize[i-1];
    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(Cols + 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

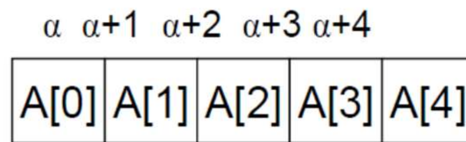
$$result_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

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

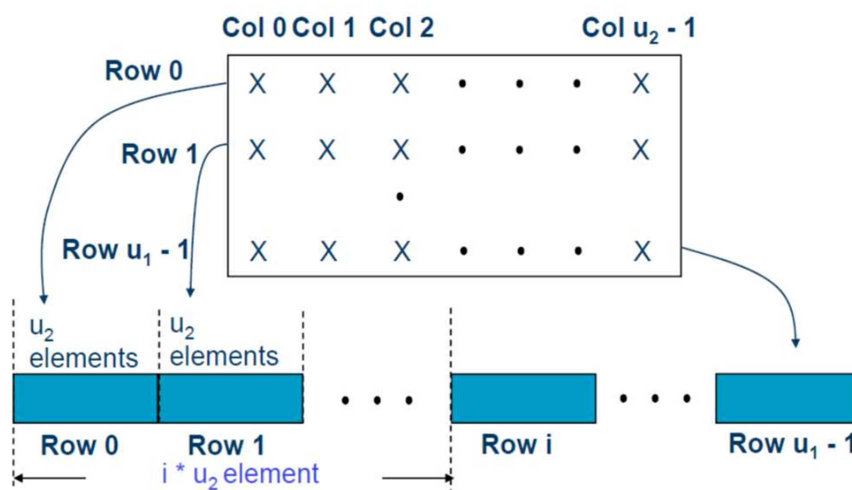
- Please study Textbook Section 2.4.4

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



2D Array in Row Major Order



Generalizing Array Representation

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

$$
\begin{aligned}
&\alpha + i_1 u_2 u_3 \ldots u_n \\
&\quad + i_2 u_3 u_4 \ldots u_n \\
&\quad + i_3 u_4 u_5 \ldots u_n \\
&\quad \vdots \\
&\quad + i_{n-1} u_n \\
&\quad + i_n
\end{aligned}
$$

$$= \alpha + \sum_{j=1}^{n} i_j a_j \quad where \begin{cases} a_j = \prod_{k=j+1}^{n} u_k & 1 \le j \le n \\ a_n = 1 \end{cases}$$

Strings

- 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

```
class String
{
public:
    String(char *init, int m);
    // Constructor to initialize *this to string init of length m.
    bool operator == (String t);
    // If *this string equals t, return true, else return false.
    bool operator!( );
    // If *this string is empty, return true, else return false.
    int Length( );
    // Return the length of *this string.
    String Concat(String t);
    // Return a string of *this string followed by t.
    String Substr(int i, int j);
    // Return the substring of *this starting from position i to i+j-1.
    int Find(String pat);
    // Return the index i where pat resides in *this, -1 if not found.
```



```
. . .

private:
    int length;
    char *str;
    int f[100];  // The failure function (more details later)
};
```

String Matching: Simple Solution

- **Algorithm: Simple string matching**
- **Input**: P and T, the pattern and text strings with length m and n. P 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 is found.

```
P:  ABABC          ABABC          ABABC
    ↓↓↓↓↓          ↓              ↓↓↓↓↓
T:  ABABABCCA   ABABABCCA   ABABABCCA
                                  ↑
                            Successful match
```

- Worst-case complexity is $O(m \times n)$

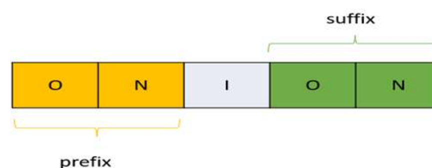

```
int String::Find(String pat)
{// Return the index where pat resides,
 // -1 if not found.
  for (int start=0; start <= Length()-
pat.Length(); star++) {
  // Check for match beginning at str[start]
    int j;
    for (j=0; j < pat.Length() &&
          str[start+j] == pat.str[j]; j++);
    if (j == pat.Length()) return start;
    // no match at position start
  }
  return −1 ; // pat is empty or not found
}
```

KMP Algorithm

- KMP algorithm (by Knuth, Morris and Pratt)
  - Runs in linear time
- Main idea:
  - Use the characteristic of the pattern string
- Phase 1:
  - Generate an LPS(*Longest Proper Prefix which is also a Suffix*) array to indicate partial match and the moving offset.
- Phase 2:
  - Use the LPS array to match and move the pattern string (skipping partial match).

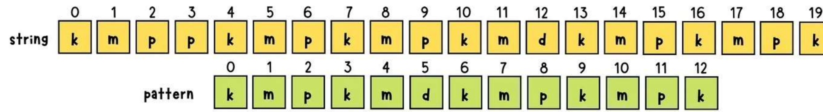
KMP Concepts

- Prefix: All the characters in a string, with one or more, cut off at the end.
  - For example, “a”, “ab”, “abc”, “abcd”, … are prefixes of “abcdabd”. (other prefixes?)
- Suffix: All the characters in a string, with one or more, cut off at the beginning.
  - For example, “dabd”, “abd”, “bd”, “d”, … are suffixes of “abcdabd”. (other suffixes?)

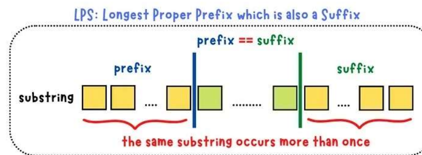


KMP Phase 1

- Compute the LPS array with the pattern string.



1. build a LPS array



-1: no LPS is identified at the given index

	0	1	2	3	4	5	6	7	8	9	10	11	12
pattern	k	m	p	k	m	d	k	m	p	k	m	p	k
LPS array	-1	-1	-1	0	1	-1	0	1	2	3	4	2	3

LPS[i] represents the ending index of early occurring prefix for each substring from index 0 to i in the string

KMP Phase 2

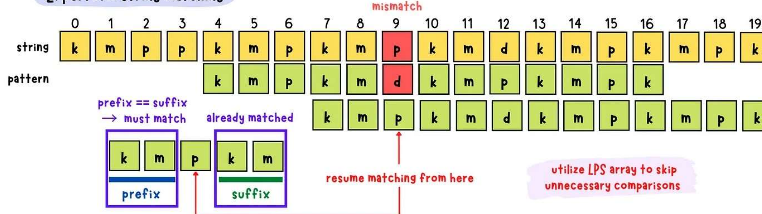
- Perform the matching. Upon a mismatch, move the pattern string with right offset to skip the partial match (already done).

-1: no LPS is identified at the given index

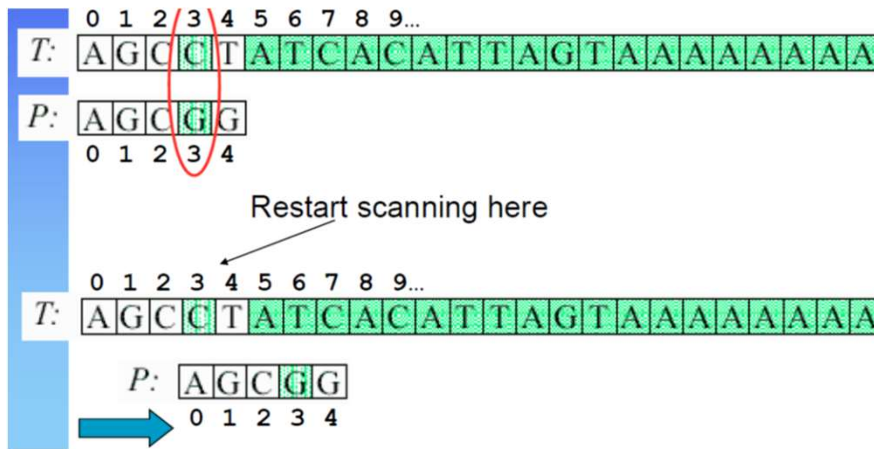
	0	1	2	3	4	5	6	7	8	9	10	11	12
pattern	k	m	p	k	m	d	k	m	p	k	m	p	k
LPS array	-1	-1	-1	0	1	-1	0	1	2	3	4	2	3

LPS[i] represents the ending index of early occurring prefix for each substring from index 0 to i in the string

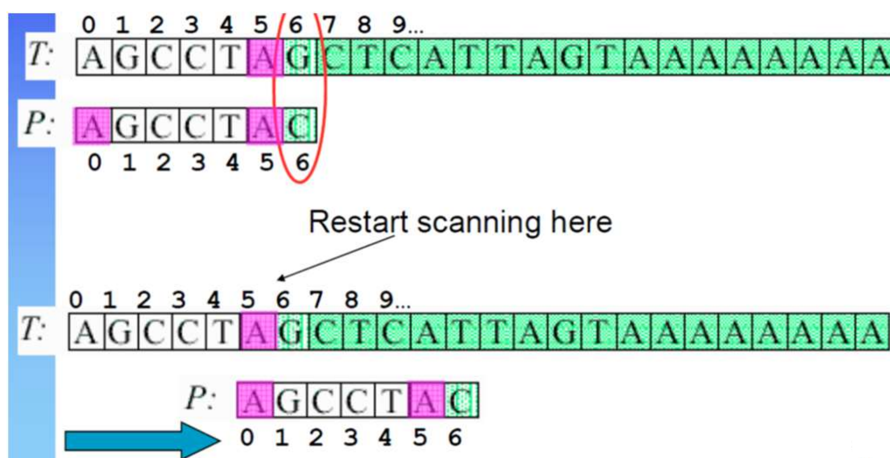
2. perform string matching



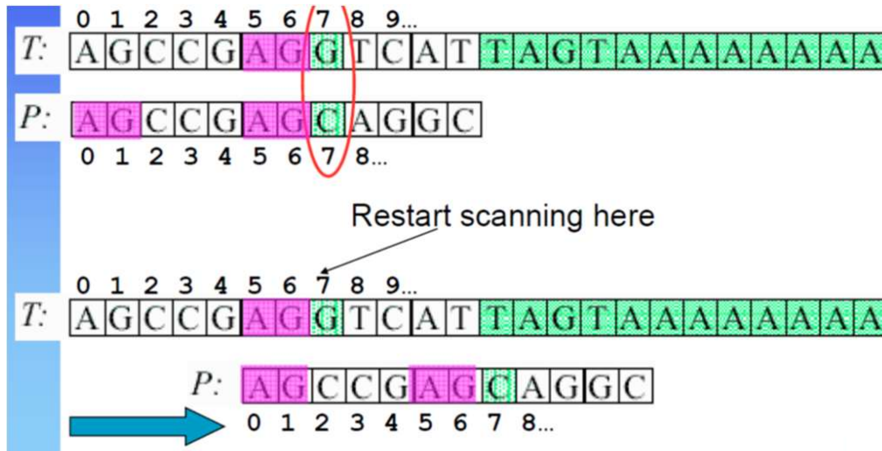
Case 1 of KMP Algorithm



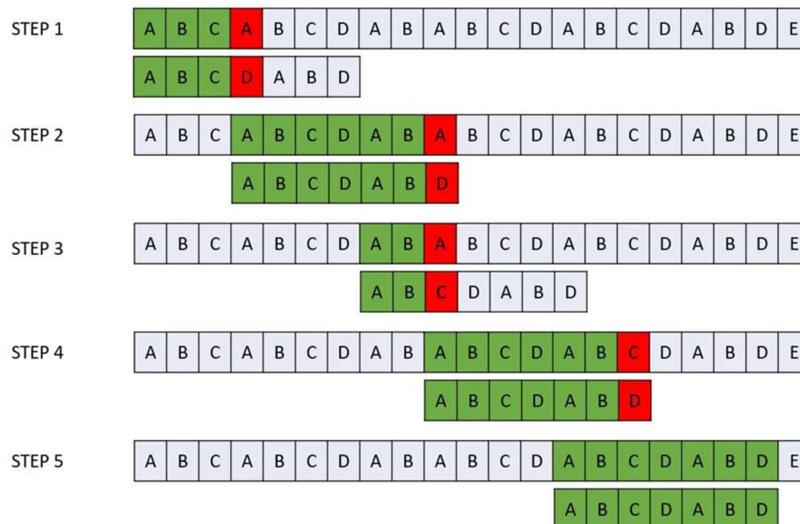
Case 2 of KMP Algorithm



Case 3 of KMP Algorithm



KMP: A Complete Example



The Failure Function

Failure Function

P j k k

Action

S i

P j-1 j

The KMP Algoriothm

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

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

- If a partial match is found such that $s_{i-j} \ldots s_{i-1} = p_0 p_1 \ldots p_{j-1}$ 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 continue by comparing s_{i+1} and p_0.

Failure Function Calculation

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

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

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

Failure Function Calculation

- j=2
  - Since k<2 and k≧0, k may be 0,1
  - When k=1 p_0p_1=ab and p_1p_2=bc → **x**
  - When k=0 p_0=a and p_2=c → **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

Failure Function Calculation

- 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

Failure Function Calculation

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

```
void String::FailureFunction()
{// failure function for string *this
  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;
  }
}
```



```
int String::FastFind(String pat)
{// determine whether pat is a substring of *this。
    int posP = 0, posS = 0;
    int lengthP = pat.Length(), lengthS = Length();
    while((posP < lengthP) && (posS < lengthS))
      if (pat.str[posP] == str[posS]) {// match
        posP++; posS++;
      }
      else if (posP == 0)
        posS++;
      else
        posP = pat.f[posP-1] + 1;
    if (posP < lengthP)
      return -1;
    else
      return posS-lengthP;
}
```

Fast String Matching Example

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

2: check failure function (f(3))

1: fail at PosP=4

3: move pattern accordingly

Program 2.15 line 12
PosP=pat.f[PosP-1]+1

Fast String Matching Example

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

Analysis of KMP Algorithm

- O(m+n)
  - O(m) for computing function f
    - Program 2.16
  - O(n) for searching P
    - Program 2.15
- Is it always better to use the KMP algorithm than the exhaustive search algorithm?

Analysis of KMP Algorithm

- KMP needs O(m) extra space allocation which is not good for large pattern.
- KMP also needs initial processing to compute the failure function.
- KMP has more complex inner loop than the simple search algorithm.
- These factors may make KMP several times slower than the basic O(mn) algorithm.
- A good example of string processing using array and the practical considerations.

Summary of Array Data Structure

- An array represents ordered list of elements with consecutive memory and accessed by indexes.
- Used in many different problems because:
  - Simple and easy to use
  - Provide O(1) random access to any element
  - Provide natural sequential, ordered access by following the indexes
- One of the most important and widely used data structures.
- Almost all PL’s support arrays by default.