DATA STRUCTURE AND ALGORITHMS

Lecture 03: Arrays, Sparse Matrix, String

ARRAYS

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

list[5]	0	1	2	3	4

- **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, … , 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 ∈ Array, *i* ∈ *index*, x ∈ *item*, *j, size* ∈ integer
  Array Create(j, list) ::= **return** an array of *j* dimensions where list is a
  j-tuple whose *i*th element is the size of the *i*th dimension. *Items* are undefined.
  Item Retrieve(A, *i*) ::= **if** (*i* ∈ *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
****Structure 2.1:** Abstract Data Type *Array* (p.50)*

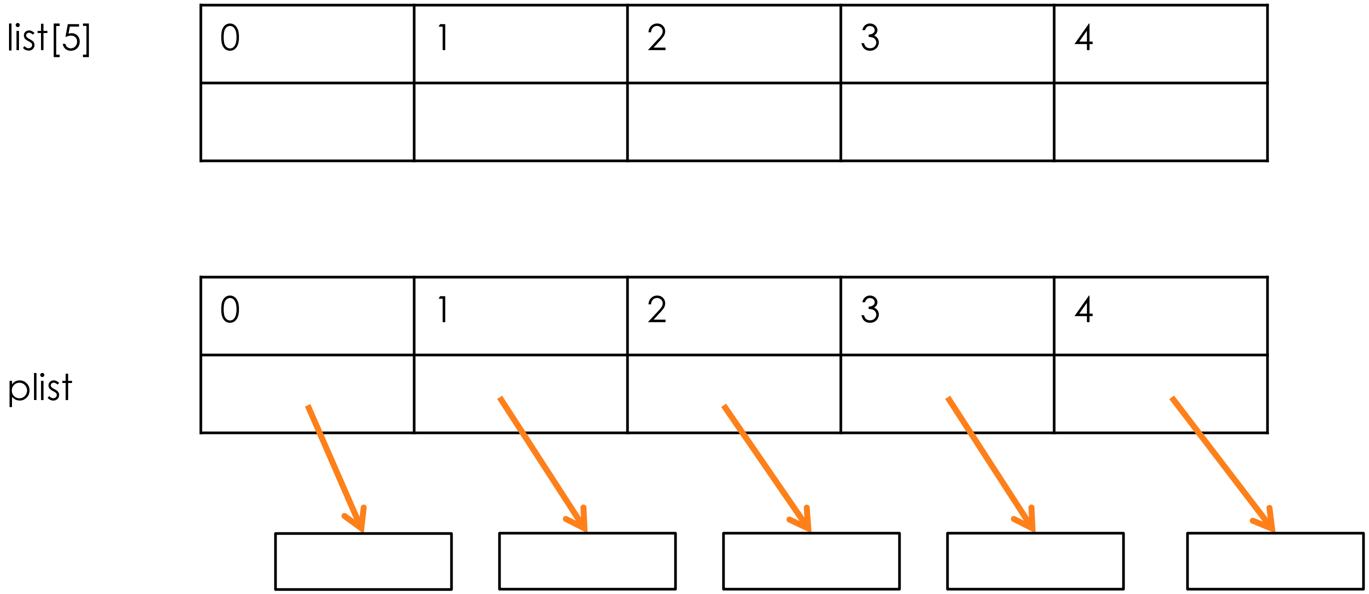
ARRAY IN C

- int list[5], *plist[5]

- list[5]: five integers

  list[0], list[1], list[2], list[3], list[4]

- *plist[5]: five pointers to integer



ARRAY IN C (CONT'D)

- Implementation of 1-D array

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

- Compare int *list1 and int list2 in C

same:	list1 and list2 are pointers
difference:	list2 reserve five locations

- Notations:

  list2 - a pointer to list2[0]

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

  *list2 + I – list2[i]

EXAMPLE: 1-DIMENSION ARRAY ADDRESSINNG

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")
}
```

EXAMPLE: 1-DIMENSION ARRAY ADDRESSINNG (CONT'D)

- Call print1(&one[0],5)

Address	Contents
1228	0
1230	1
1232	2
1234	3
1236	4

*__Figure 2.1:__ One- dimensional array addressing (p.53)

STRUCTURES (RECORDS)

```
struct {
        char name[10];
        int age;
        float salary;
        } person;
strcpy(person.name, “James”)
person.age = 10;
person.salary = 35000;
```

CREATE STRUCTURE DATA TYPE

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

Or

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

Human_being person1, person2;
```

ORDERED LIST

- Ordered (linear) list:
  - (item1, item2, item3,…, item*K*)

- 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

(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

(6) Delete the element at position i, causing elements numbered i+1, …, n to become numbered i, i+1, …, n-1

IMPLEMENTATION ON ORDERED LIST

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

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 **degree**
- A polynomial is called **sparse** when it has many zero terms
- Implement polynomials by arrays

Polynomials $A(X)=3X^{20}+2X^5+4$, $B(X)=X^4+10X^3+3X^2+1$

- **Structure** *Polynomial* is
  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 *Coefficients* and e_i in *Exponents*, e_i are integers >= 0

  functions:
  for all *poly, poly1, poly2* *Polynomial*, *coef* *Coefficients*, *expon* *Exponents*
  Polynomial Zero( ) ::= **return** the polynomial, *p(x)* = 0
  Boolean IsZero(*poly*) ::= **if** (*poly*) **return** *FALSE* **else return** *TRUE*
  Coefficient Coef(*poly, expon*) ::= **if** (*expon* *poly*) **return** its coefficient **else return** Zero
  Exponent Lead_Exp(*poly*) ::= **return** the largest exponent in *poly*
  Polynomial Attach(*poly,coef, expon*) ::= **if** (*expon* *poly*) **return** error **else return** the polynomial poly with the term <*coef, expon*> inserted

Polynomial Remove(*poly, expon*) ::= **if** (*expon* *poly*) **return** the polynomial *poly* with the term whose exponent is *expon deleted*
else return error

Polynomial SingleMult(*poly, coef, expon*) ::= **return** the polynomial *poly* • *coef* • x^{expon}

Polynomial Add(*poly1, poly2*) ::= **return** the polynomial *poly*1 +*poly*2

Polynomial Mult(*poly1, poly2*) ::= **return** the polynomial *poly*1 • *poly*2

End *Polynomial*

***Structure 2.2:**Abstract data type *Polynomial* (p.61)

Polynomial Addition

data structure 1:

```
#define MAX_DEGREE 101
typedef struct {
        int degree;
        float coef[MAX_DEGREE];
        } polynomial;
```

- ```
  /* d =a + b, where a, b, and d are polynomials */
  d = Zero( )
  while (! IsZero(a) && ! IsZero(b)) do {
    switch COMPARE (Lead_Exp(a), Lead_Exp(b))  {
      case -1: d =
        Attach(d, Coef (b, Lead_Exp(b)), Lead_Exp(b));
        b = Remove(b, Lead_Exp(b));
        break;
     case  0: sum = Coef (a, Lead_Exp (a)) + Coef ( b, Lead_Exp(b));
       if (sum) {
          Attach (d, sum, Lead_Exp(a));
          a = Remove(a , Lead_Exp(a));
          b = Remove(b , Lead_Exp(b));
          }
        break;
  ```
  ```

```
case 1: d =
        Attach(d, Coef (a, Lead_Exp(a)), Lead_Exp(a));
        a = Remove(a, Lead_Exp(a));
      }
  }
insert any remaining terms of a or b into d
```

advantage: easy implementation
disadvantage: waste space when sparse

***Program 2.4 :**Initial version of *padd* function(p.62)

DATA STRUCTURE 2: USE ONE GLOBAL ARRAY TO STORE ALL POLYNOMIALS

$A(X)=2X^{1000}+1$

$B(X)=X^4+10X^3+3X^2+1$

***Figure 2.2:** Array representation of two polynomials (p.63)

	starta	*finisha*	*startb*			*finishb*	*avail*
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

specification	representation
poly	<start, finish>
A	<0,1>
B	<2,5>

- storage requirements: start, finish, 2*(finish-start+1)
- nonparse: twice as much as (1)
- when all the items are nonzero

```
MAX_TERMS 100 /* size of terms array */
typedef struct {
        float coef;
        int expon;
        } polynomial;
polynomial terms[MAX_TERMS];
int avail = 0;
```

*(p.62)

Add two polynomials: D = A + B

```
void padd (int starta, int finisha, int startb, int finishb, int * startd,
                int *finishd)
{
/* add A(x) and B(x) to obtain D(x) */
   float coefficient;
  *startd = avail;

  while (starta <= finisha && startb <= finishb)
   switch (COMPARE(terms[starta].expon,
                              terms[startb].expon)) {

    case -1: /* a expon < b expon */
          attach(terms[startb].coef, terms[startb].expon);
          startb++
          break;
```

```
case  0: /* equal exponents */
            coefficient = terms[starta].coef +
                          terms[startb].coef;
            if (coefficient)
              attach (coefficient, terms[starta].expon);
            starta++;
            startb++;
            break;

case 1: /* a expon > b expon */
        attach(terms[starta].coef, terms[starta].expon);
        starta++;
}
```

```
/* add in remaining terms of  A(x) */
for( ; starta <= finisha; starta++)
     attach(terms[starta].coef, terms[starta].expon);

/* add in remaining terms of B(x) */
for( ; startb <= finishb; startb++)
    attach(terms[startb].coef, terms[startb].expon);
*finishd =avail -1;
}
```

Analysis: O(n+m)
where n, m are the number of nonzeros in A, B, respectively.

***Program 2.5:** Function to add two polynomial (p.64)

- void attach(float coefficient, int exponent)

```
{
/* add a new term to the polynomial */
    if (avail >= MAX_TERMS) {
       fprintf(stderr, "Too many terms in the polynomial\n");
       exit(1);
      }
      terms[avail].coef  = coefficient;
      terms[avail++].expon = exponent;
}
```

***Program 2.6:**Function to add anew term (p.65)

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

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

SPARSE MATRIX

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

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

SPARSE MATRIX (CONT'D)

- 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 $O(m \times n)$

ADT OF SPARSE MATRIX

Structure *Sparse_Matrix* is

objects: a set of triples, <*row, column, value*>, where *row* and *column* are integers and form a unique combination, and *value* comes from the set *item*.

functions:

for all *a, b* ∈ *Sparse_Matrix*, x   *item, i, j, max_col, max_row*   *index*

Sparse_Marix Create(*max_row, max_col*) ::=

return a *Sparse_matrix* that can hold up to *max_items* = *max _row*   *max_col* and whose maximum row size is *max_row* and whose maximum column size is *max_col.*

ADT OF SPARSE MATRIX (CONT'D)

Sparse_Matrix Transpose(*a*) ::=
return the matrix produced by interchanging the row and column value of every triple.

Sparse_Matrix Add(*a*, *b*) ::=
if the dimensions of a and b are the same
return the matrix produced by adding corresponding items, namely those with identical *row* and *column* values.
else return error

Sparse_Matrix Multiply(*a*, *b*) ::=
if number of columns in a equals number of rows in **b**
return the matrix *d* produced by multiplying a by *b* according to the formula: *d* [*i*] [*j*] = (a[i][k]•b[k][j]) where *d (i, j)* is the *(i,j)*th element
else return error.

SPARSE MATRIX REPRESENTATION

- Represented by a two-dimensional array.
  - Sparse matrix wastes space.

- 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

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

(a)

# of rows (columns)

# of nonzero terms

transpose

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

(b)

row, column in ascending order

***Figure 2.4:**Sparse matrix and its transpose stored as triples (p.69)

Sparse_matrix Create(max_row, max_col) ::=

```
#define MAX_TERMS 101 /* maximum number of terms +1*/
   typedef struct {
            int col;
            int row;
            int value;
            } term;
   term a[MAX_TERMS]
```

# of rows (columns)
# of nonzero terms

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 very often.

(2) For all elements in column j,

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

```
void transpose (term a[], term b[])
/* b is set to the transpose of a */
{
    int n, i, j, currentb;
    n = a[0].value;  /* total number of elements */
    b[0].row = a[0].col;  /* rows in b = columns in a */
    b[0].col = a[0].row;  /*columns in b = rows in a */
    b[0].value = n;
    if (n > 0) {                /*non zero matrix */
        currentb = 1;
        for (i = 0; i < a[0].col; i++)
        /* transpose by columns in a */
            for( j = 1; j <=  n; j++)
            /*  find elements from the current column */
            if (a[j].col == i) {
           /* element is in current column, add it to b */
```

columns

elements

```
b[currentb].row = a[j].col;
b[currentb].col  = a[j].row;
b[currentb].value = a[j].value;
currentb++
}
}
}
```

* **Program 2.7:** Transpose of a sparse matrix (p.71)

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

==> O(columns*elements)

COMPARE WITH 2-DIMENSIONAL ARRAY REPRESENTATION

- Discussion: compared with 2-D array representation
  - O(columns×elements) versus O(columns×rows)
  - elements → columns×rows when non-sparse

  ➔ O(columns^2×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.

a[0]	6	6	8
a[1]	0	0	15
a[2]	0	3	22
a[3]	0	5	-15
a[4]	1	1	11
a[5]	1	2	3
a[6]	2	3	-6
a[7]	4	0	91
a[8]	5	2	28

INDEX	[0]	[1]	[2]	[3]	[4]	[5]
ROW_TERMS =	2	1	2	2	0	1
STARTING_POS =	1	3	4	6	8	8

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 (CONT’D)

```
void fast_transpose(term a[ ], term b[ ])
{
/* the transpose of a is placed in b */
  int row_terms[MAX_COL], starting_pos[MAX_COL];
  int i, j, num_cols = a[0].col, num_terms = a[0].value;
  b[0].row = num_cols; b[0].col = a[0].row;
  b[0].value = num_terms;
  if (num_terms > 0){ /*nonzero matrix*/
```

columns

```
    for (i = 0; i < num_cols; i++)
        row_terms[i] = 0;
```

elements

```
    for (i = 1; i  <= num_terms; i++)
        row_term [a[i].col]++
    starting_pos[0] = 1;
```

columns

```
    for (i =1; i < num_cols; i++)
        starting_pos[i]=starting_pos[i-1] +row_terms [i-1];
```

elements

```
        for (i=1; i <= num_terms, i++) {
                j = starting_pos[a[i].col]++;
                b[j].row = a[i].col;
                b[j].col = a[i].row;
                b[j].value = a[i].value;
        }
    }
}
```

***Program 2.8:**Fast transpose of a sparse matrix

Compared with 2-D array representation

O(columns+elements) vs. O(columns*rows)

elements --> columns * rows

O(columns+elements) --> O(columns*rows)

Cost: Additional row_terms and starting_pos arrays are required.

Let the two arrays row_terms and starting_pos be shared.

MATRIX MULTIPLICATION

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

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

$$\text{for } 0 \le \mathrm{i} < \mathrm{m} \text{ and } 0 \le \mathrm{j} < \mathrm{p}$$

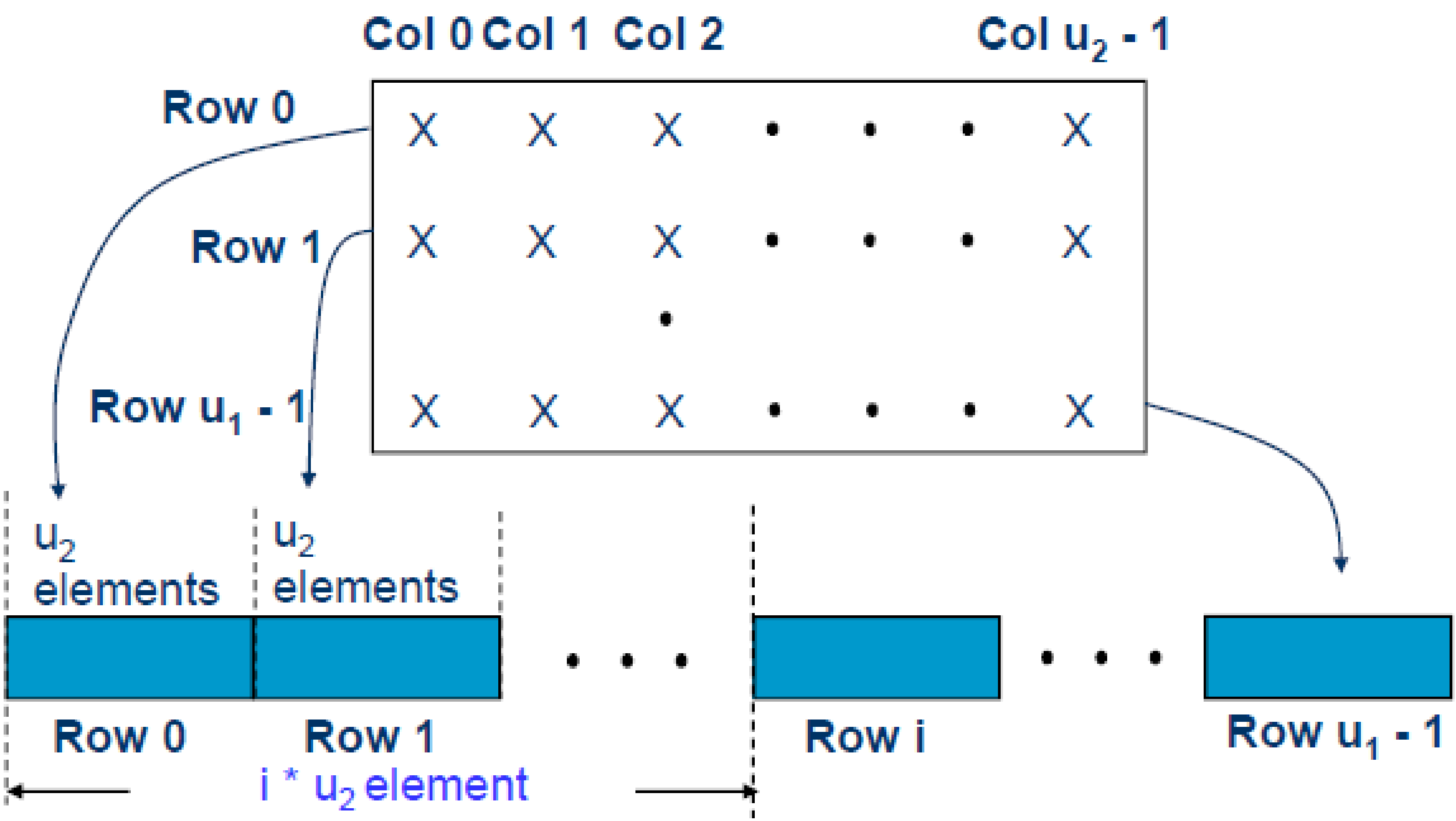
REPRESENTATION OF ARRAYS

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

- Example: One dimensional array

a	a+1	a+2	a+3	a+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

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

$$=\alpha + \sum_{j=1}^{n} i_j a_j \text{, where } \begin{cases} a_j = \prod_{k=j+1}^{n} u_k \text{, } 1 \leq j \leq n \\ a_n = 1 \end{cases}$$

STRING

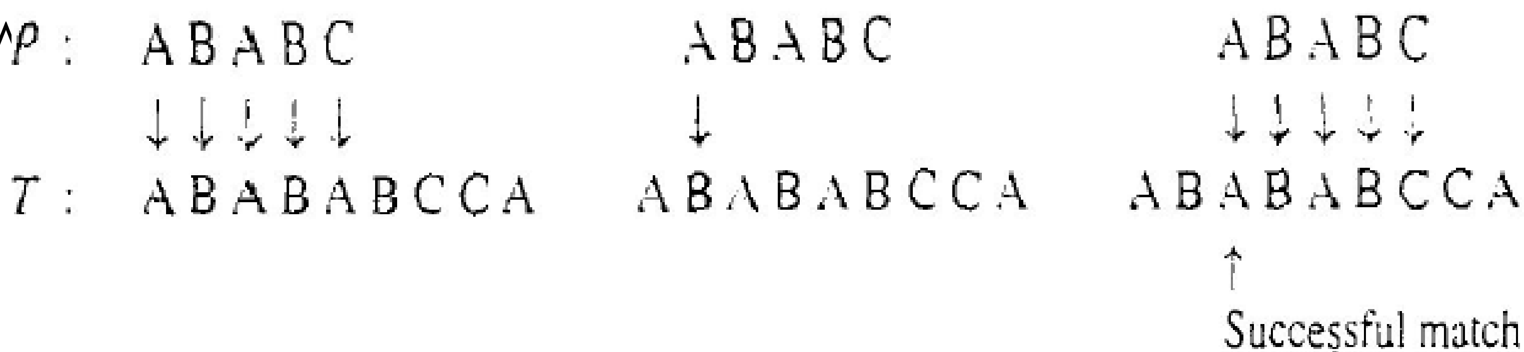
- Usually string is represented as a character array.
- General string operations include comparison, string concatenation, copy, insertion, string matching, printing, etc.

Note: ‘\0’ is a null character, which is used to represent the end of a string.

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

STRING MATCHING: STRAIGHTFORWARD SOLUTION

- **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.
- W



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

	0	1	2	3	4	5	6	7	8	9	…									
T:	A	G	C	C	T	A	T	C	A	C	A	T	T	A	G	T	A	A	A	A
P:	A	G	C	G	C															
	0	1	2	3	4															

	0	1	2	3	4	5	6	7	8	9	…									
T:	A	G	C	C	T	A	T	C	A	C	A	T	T	A	G	T	A	A	A	A
				P:																
				A	G	C	G	C												
				0	1	2	3	4												

THE SECOND CASE FOR THE KMP ALGORITHM

	0	1	2	3	4	5	6	7	8	9	…									
T:	A	G	C	C	T	A	G	C	T	C	A	T	T	A	G	T	A	A	A	A
P:	A	G	C	C	T	A	C													
	0	1	2	3	4															

	0	1	2	3	4	5	6	7	8	9	…									
T:	A	G	C	C	T	A	G	C	T	C	A	T	T	A	G	T	A	A	A	A
					P:	A	G	C	C	T	A	C								
						0	1	2	3	4										

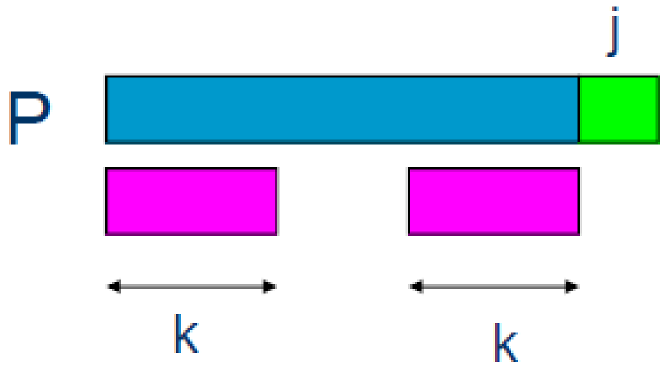
THE THIRD CASE FOR THE KMP ALGORITHM

	0	1	2	3	4	5	6	7	8	9	…									
T:	A	G	C	C	T	A	G	G	T	C	A	T	T	A	G	T	A	A	A	A
P:	A	G	C	C	T	A	G	C	A	G	G	C								
	0	1	2	3	4															

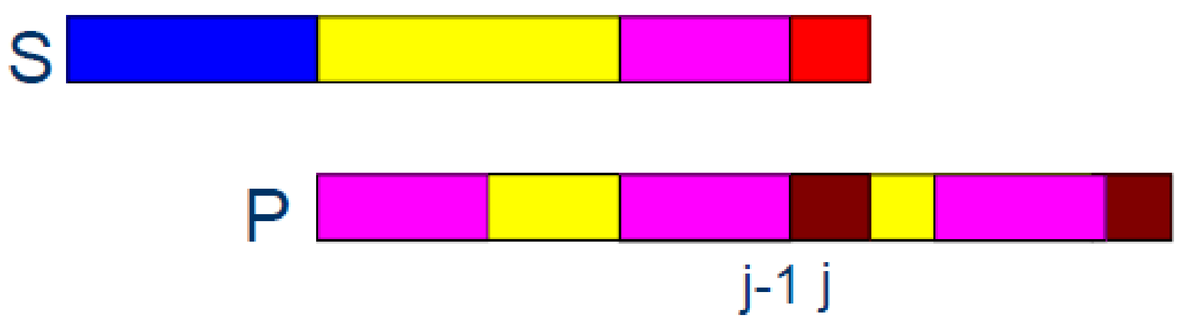
	0	1	2	3	4	5	6	7	8	9	…									
T:	A	G	C	C	T	A	G	G	T	C	A	T	T	A	G	T	A	A	A	A
					P:	A	G	C	C	T	A	G	C	A	G	G	C			
						0	1	2	3	4										

KMP ALGORITHM (CONT'D)

Failure Function



Action



KMP ALGORITHM (CONT'D)

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

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

- If a partial match is found such that $s_{i-j} \dots s_{i-1} = p_0 p_1 \dots p_{j-1}$ 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. $p_0 p_1 \dots p_k = p_{j-k} p_{j-k+1} \dots p_j$
- $j = 0$
  - 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 $p_1 = b$ ➔ X
  - $f(1) = -1$

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 (CONT'D)

- $j = 2$
  - Since $k < 2$ and $k \geq 0$, 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
  - $f(2) = -1$
- $j = 3$
  -
  -
  -
  -
  - $f(3) = 0$

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						

FAST MATCHING EXAMPLE: FAILURE FUNCTION CALCULATION (CONT'D)

- $j = 4$
  - Since $k < 4$ and $k \geq 0$, 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: FAILURE FUNCTION CALCULATION (CONT'D)

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

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* (posP-1)

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

posP = pat.*f* [posP-1]+1

THE ANALYSIS OF THE KMP ALGORITHM

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