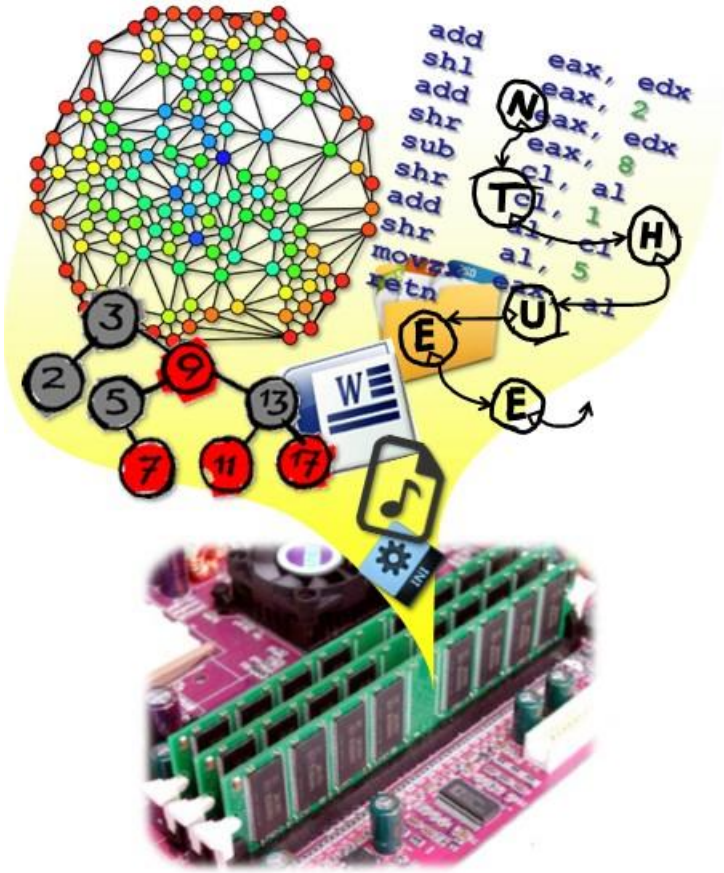
Data Structures

CH2 Arrays

Prof. Ren-Shuo Liu

NTHU EE

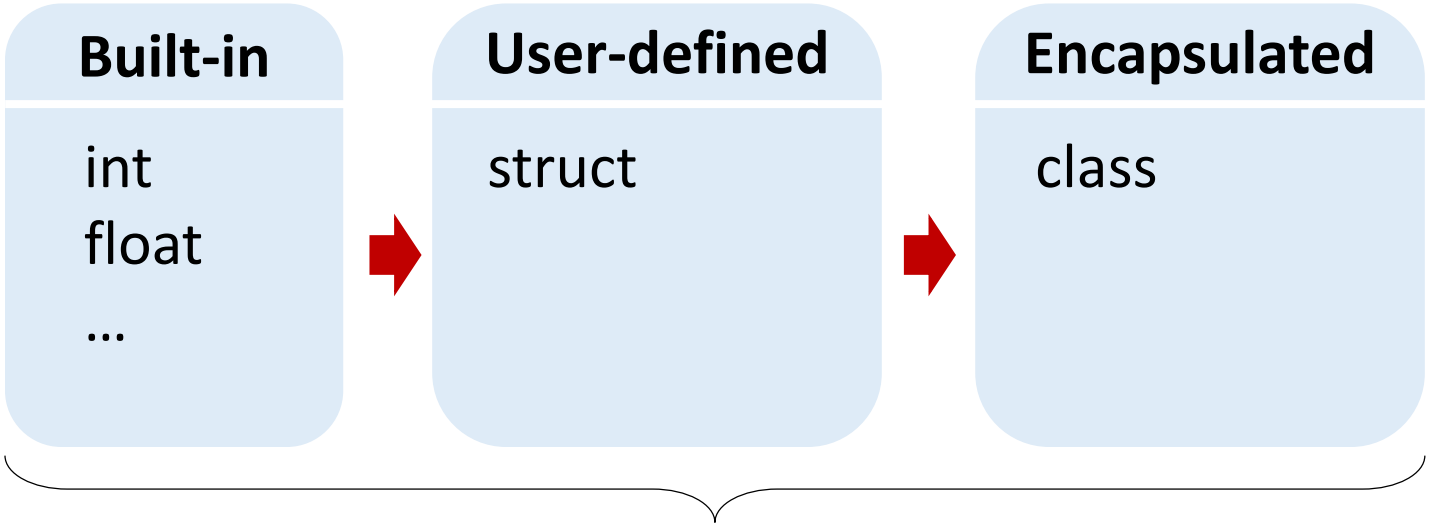
Spring 2019



Outline

- **2.1 Abstract Data Types and C++ Class**
- 2.2 The Array as an Abstract Data Type
- 2.3 The Polynomial Abstract Data Type
- 2.4 Sparse Matrices
- 2.5 Representation of Arrays
- 2.6 The String Abstract Data Type

From *int* to *class*



"Computability" is not changed: They can solve the same set of problems.

Limited number of basic types → Unlimited number of desired types → Improved protection and reusability

Built-in Types

- C++ natively supports integers and their operations
  - -1, 0, 1, 99, 1+99, 64/8
- Rectangles with integral height and width are not natively supported

- Using basic types to compute the area ratio of two rectangles is thus not very elegant

```
float ratio(int h1, int w1, int h2, int w2)
{
  float area1 = h1 * w1;
  float area2 = h2 * w2;
  return (a1/a2);
};
```

This is not very elegant.
The number of arguments becomes six if we want to handle cubes!

struct


```
struct Rectangle{
  int h;
  int w;
};
float ratio(Rectangle r1, Rectangle r2);
int main()
{
   Rectangle a, b;
   a.h = 4; a.w = 8;    b.h = 5; b.w=20; //initialization
   cout << ratio(a, b) << endl;
   return 0;
}
float ratio(Rectangle r1, Rectangle r2)
{
  float area1 = r1.h * r1.w;
  float area2 = r2.h * r2.w;
  return (area1/area2);
};
```

Rectangle acts as a user-defined type!

More elegant here!

Still not very elegant

struct with an Initialization Function

```
struct Rectangle{
  int h;   int w;
};
void initialize(Rectangle * r, int hi, int wi)
{
   assert( hi > 0 && wi  > 0 );
   r->h = hi;   r->w = wi;
   return;
}
int main()
{
   Rectangle a, b;
   initialize(&a, 4, 8);
   initialize(&b, 5, 20);
   cout << ratio(a, b) << endl;
   return 0;
}
```

A bit better, right?

struct with a Member Function

```
struct Rectangle{
  int h;   int w;
  void initialize(int hi, int wi);
};
void Rectangle::initialize(int hi, int wi)
{
   assert( hi > 0 && wi  > 0 );
   h = hi;    w = wi;
   return;
}
int main()
{
   Rectangle a, b;
   a.initialize(4, 8);
   b.initialize(5, 20);
   b.h = 0;    //wrong value
   cout << ratio(a, b) << endl;
   return 0;
}
```

Initialize() becomes a member function of Rectangle.

"::"is **Scope Resolution Operator**

- "Rectangle" is a scope in which this initialize() resides.

This is the best so far, but it would be better if the language can prevent users from:

- forgetting to initialize a Rectangle
- directly accessing the internal values of Rectangle

class

```
class Rectangle
{
public:
   Rectangle(int hv , int wv);
   ~Rectangle();
   int GetHeight();
   int GetWidth();
   void SetHeight(int i);
   void SetWidth(int i);
private:
   int h, w;
};
```

- Components of a class
  - Class name
    - Rectangle
  - Data members
    - h, w
  - Member functions
    - Constructor
      - Rectangle()
    - Destructor
      - ~Rectangle()
    - Others
      - GetHeight(), SetHeight() …
  - Level of program access
    - Public, private, protected

Constructor


```
class Rectangle
{
public:
   Rectangle(int hv , int wv);
   ~Rectangle();
   int GetHeight();
   int GetWidth();
   void SetHeight(int i);
   void SetWidth(int i);
private:
   int h, w;
};
```

- Objective
  - Initialization of objects
- Rules
  - Constructor name must be identical to the class name
  - Must be public
  - No return type
  - No return value
  - Automatically invoked when an object is created

Destructor


```
class Rectangle
{
public:
   Rectangle(int hv , int wv);
   ~Rectangle();
   int GetHeight();
   int GetWidth();
   void SetHeight(int i);
   void SetWidth(int i);
private:
   int h, w;
};
```

- Objective
  - Do cleaning jobs, e.g., free allocated memory
- Rules
  - Destructor name must be identical to the class name prefixed with a tilde, "~"
  - Must be public
  - No return type
  - No return value
  - Automatically invoked when an object
    - goes out of scope
    - gets deleted

Member Function Implementation

```
class Rectangle
{
public:
   Rectangle(int hv , int wv);
   ~Rectangle();
   int GetHeight();
   int GetWidth();
   void SetHeight(int i);
   void SetWidth(int i);
private:
   int h, w;
};
```

```
Rectangle::Rectangle(int hv, int wv)
{
   SetHeight(hv);  SetWidth(wv);
};
```

```
Rectangle::~Rectangle()
{ /*do nothing*/}
```

```
void Rectangle::SetHeight(int i)
{
  if(i>0) h = i;
  else throw("invalid height");
}
```

```
int Rectangle::GetHeight()
{
  return h;
}
//SetWidth and GetWidth are omitted here
```

Using class


```
float ratio(Rectangle r1, Rectangle r2)
{
  float area1 = r1.GetHeight() * r1.GetWidth();
  float area2 = r2.GetHeight() * r2.GetWidth();
  return (area1/area2);
};
```

```
int main()
{
   Rectangle a(4, 8), b(5, 20);

   cout << ratio(a, b) << endl;
   return 0;
}
```

- Declaring and initializing Rectangle become very natural.
- This is the reward of our hard work in designing the class.

We want to further improve this representation...

Operator Overloading


```
int main()
{
   Rectangle a(4, 8), b(5, 20);

   //cout << ratio(a, b) << endl;
   cout << a/b << endl;
   return 0;
}
```

Can we use division operator to denote the area ratio of two rectangles?
Yes! we can write some code to specify this additional definition, i.e., operator overloading

```
int main()
{
   Rectangle a(4, 8), b(5, 20);

   //cout << ratio(a, b) << endl;
   cout << (a/3)/b << endl;
   return 0;
}
```

Can we also use division operator to denote partitioning a rectangle by an integer?
Yes! we can additionally define dividing Rectangle by an integer.

struct vs. *class* in C++

- *struct* and *class* are identical in the C++ language except for the default level of program access
  - default *public* for *struct*
  - default *private* for *class*

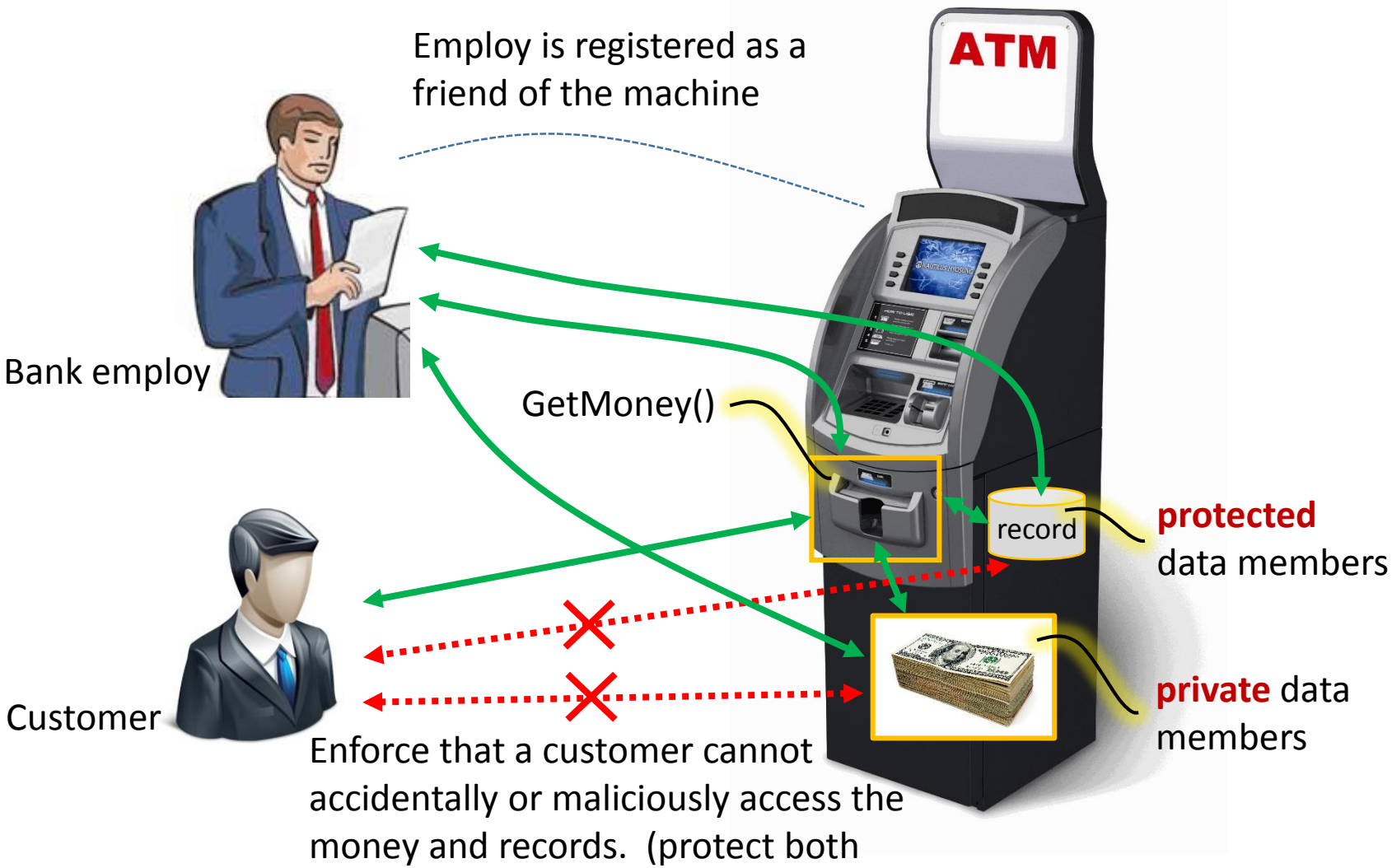
Level of Program Access


```
class Rectangle
{
public:
    Rectangle();
    ~Rectangle();
    int GetHeight();
    int GetWidth();
    void SetHeight(int i);
    void SetWidth(int i);
private:
    int h, w;
};
```

Accessed by	public	protected	private
Same class	Allowed	Allowed	Allowed
Friend class	Allowed	Allowed	Allowed
Derived class	Allowed	Allowed	Disallowed
Others	Allowed	Disallowed	Disallowed

- Declaring all data members of a class as *private*
  → enforce data encapsulation
  (also known as information hiding)

Automatic Teller Machine



Advanced Encapsulation

```
class Rectangle
{
public:
    Rectangle();
    ~Rectangle();
    int GetHeight();
    int GetWidth();
    void SetHeight(int i);
    void SetWidth(int i);
private:
    int h, w;
};
```

- Students interested in more aggressive encapsulation
  - Types, names, and quantity of private data can also be hidden from the class users by using an opaque pointer
    - Objective: the implementation of the class can be changed without the need to recompile the program using it
  - Keyword: Cheshire Cat opaque pointer

Address and Pointer of Objects

```
class Rectangle
{
public:
   Rectangle(int hv, int wv);
   ~Rectangle();
   int GetHeight();
   int GetWidth();
   void SetHeight(int i);
   void SetWidth(int i);
private:
   int h, w;
};
```

```
int main()
{
 Rectangle a(4, 8);
 Rectangle * p = &a.

 a.SetHeight(12);
 *(&a).SetHeight(12);
 (*p).SetHeight(12);
 p->SetHeight(12);

 return 0;
};
```

same functionality

"ptr->" is the shorthand of "(*ptr)"

Outline

- 2.1 Abstract Data Types and C++ Class
- **2.2 The Array as an Abstract Data Type**
- 2.3 The Polynomial Abstract Data Type
- 2.4 Sparse Matrices
- 2.5 Representation of Arrays
- 2.6 The String Abstract Data Type

General Array as an ADT

Default argument

```
class GeneralArray {
public:
    GeneralArray(int j, RangeList list, float initValue = defaultValue);
    // j: dimension
    // list: a finite ordered set of dimension j
    // initValue: just as its name suggests

    float Retrieve(index i);
    // Return the float corresponding to the index i;
    // throw an exception if i is not in the index set

    void Store(index i, float x);
    // Replace the float corresponding to the index i;

};
```

General Array vs. C++'s Raw Array

	General Array	Raw Array
Index set	more flexible composition 勝	consecutive integers starting at 0
Range checking for indexing	Yes 勝	No
Access methods	Less-intuitive **Retrieve**() and **Store**() functions	intuitive **[]** and **=** operators 勝
Access speed	Lower (at least due to range checking)	Higher 勝

Is it possible for the general array to also accept **[]** and **=** operators ?
A: Yes, by using operator overloading.

Outline

- 2.1 Abstract Data Types and C++ Class
- 2.2 The Array as an Abstract Data Type
- **2.3 The Polynomial Abstract Data Type**
- 2.4 Sparse Matrices
- 2.5 Representation of Arrays
- 2.6 The String Abstract Data Type

ADT Polynomial


```
class Polynomial {
// p(x) = a0 x^e0 + … + an x^en
// where ai is nonzero float and ei is non-negative int
public:
    Polynomial( );
    // construct the polynomial p(x) = 0

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

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

    float Eval(float f );
    // Evaluate the polynomial *this at f and return the results
};
```

c++ *this* pointer 

Polynomial Representation

- Basic idea
  - Use an ordered list to store the coefficients and exponents
- Three representations
  - Fixed array of coefficients
  - Dynamic array of coefficients
  - Dynamic array of (coefficient, exponent)-tuples

Three Representations

- $f(x) = 3x^5 + 7x$

9	0
8	0
7	0
6	0
5	3
4	0
3	0
2	0
1	7
0	0

Fixed array of coefficients

5	3
4	0
3	0
2	0
1	7
0	0

Dynamic array of coefficients

1	3	5
0	7	1

Dynamic array of (coefficient, exponent)-tuples

Fixed Array of Coefficients


```
class Polynomial {
public:
    //omit
private:
    int degree;
    float coef[MaxDegree + 1]
};
```

Dynamic Array of Coefficients


```
class Polynomial {
public:
    Polynomial(int d);
    //omit
private:
    int degree;
    float * coef;
}

Polynomial::Polynomial(int d)
{
    degree = d;
    coef = new float[degree+1];
};
```

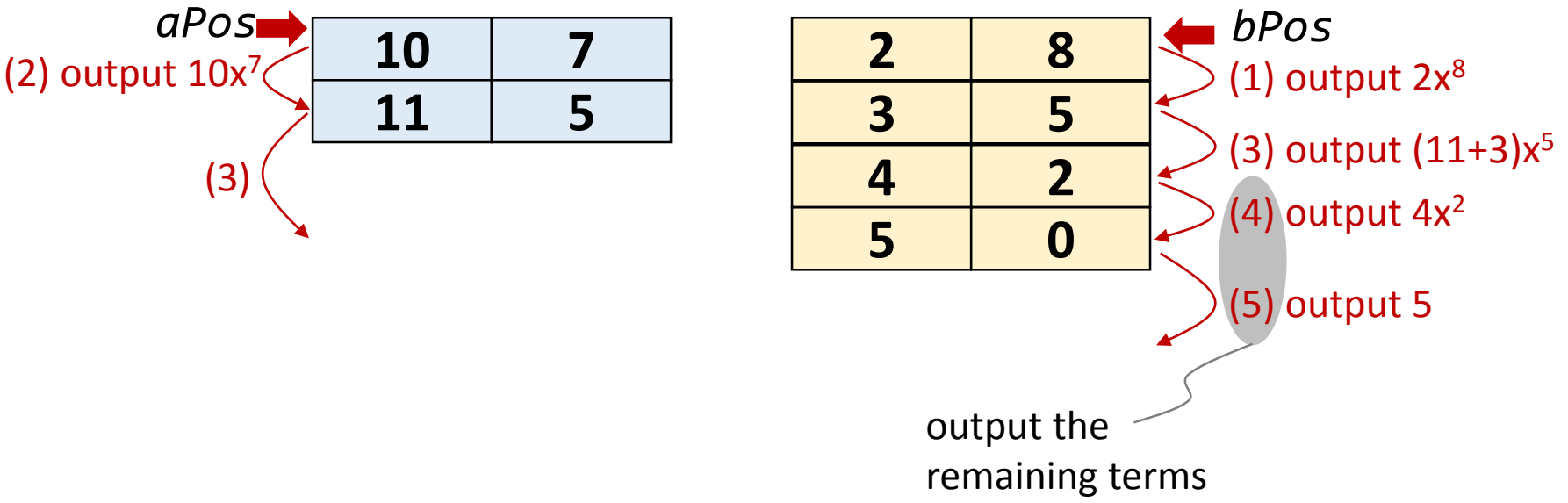
Dynamic Array of Tuples


```
class Polynomial;  //forward declaration

class Term{
friend Polynomial;
private:
    float coef;  //coefficient
    int exp;   //exponent
};
class Polynomial{
    //omit
private:
    Term * termArray;
    int capacity;
    int terms;
}
```

Polynomial Addition Algorithm

$(10x^7 + 11x^5) + (2x^8 + 3x^5 + 4x^2 + 5)$



Polynomial Addition


```
Polynomial Polynomial::Add(Polynomial 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++;
    }
    else {
      c.NewTerm (termArray[aPos].coef, termArray[aPos].exp);
      aPos++;
    }
  }
  //please continue on the next page …
```

// usage: c = a.Add(b)

Polynomial Addition (Cont'd)


```
    for ( ; aPos < terms ; aPos++)  //output the remaining terms
      c.NewTerm (termArray[aPos].coef, termArray[aPos].exp);

    for ( ; bPos < b.terms ; bPos++) //output the remaining terms
      c.NewTerm (b.termArray[bPos].coef, b.termArray[bPos].exp);

    return c;
}
```

Add New Term into Polynomial

```
void Polynomial::NewTerm(const float theCoeff, const int theExp)
{// Add a new term to the end of termArray
    if (terms == capacity)  // termArray full
    {// double the capacity of termArray
        capacity *= 2;
        term *temp = new term[capacity];
        copy(termArray, termArray + terms, temp);
        delete [] termArray ;
        termArray = temp;
    }
    termArray [terms].coef = theCoeff;
    termArray [terms++].exp = theExp;
}
```

Polynomial Addition (Cont'd)

- Time complexity = O(*m + n* + array doubling)
  - *m* and *n* are the number of terms of the two arrays
  - At least one of *aPos* and *bPos* increases by one per iteration
  - While loop terminates if *aPos*==*m* or *bPos*==*n*

- O(array doubling) = O(m + n)
  - Suppose at the end of computing *c* = *a* + *b*, the array size of *c* is 2^k
  - $2^{k-1} \leq$ (m+n); otherwise, the array should only grow to 2^{k-1}
  - Time spent in array doubling is $(1 + 2 + 4 + \ldots + 2^k)$ $= O(2^{k+1}) = O(2^k)$
  - $O(2^{k+1}) = O(2^k) = O(2^{k-1}) = O(m+n)$
  - → O(*m + n* + array doubling) = O(*m + n*)

Outline

- 2.1 Abstract Data Types and C++ Class
- 2.2 The Array as an Abstract Data Type
- 2.3 The Polynomial Abstract Data Type
- **2.4 Sparse Matrices**
- 2.5 Representation of Arrays
- 2.6 The String Abstract Data Type

Matrix

- Dense matrix
  - Many non-zeros
- Sparse matrix
  - Many zero terms
  - Sparse matrix is very popular
  - E.g., the visiting counts of people to places

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

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

ADT Sparse Matrix


```
class SparseMatrix
{
public:
    SparseMatrix(int r, int c, int t);
    //constructor.
    //r is #row, c is #col, t is #non-zero terms

    SparseMatrix Transpose( );

    SparseMatrix Add(SparseMatrix b);

    SparseMatrix Multiply(SparseMatrix b);
};
```

Sparse Matrix Representation

- 2D array representation wastes not only memory space but also computing time
- Dynamic array of (row, col, value) is better
  - Each triple stands for a non-zero term
  - Terms are ordered by row and then by columns

	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

Eight non-zeros

col

row

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

Sparse Matrix Representation


```
class SparseMatrix;  //forward declaration

class MatrixTerm {
friend SparseMatrix;
private:
    int row, col, value;  // a triple representing a term
};

class SparseMatrix
{
public:
    SparseMatrix Transpose();
    SparseMatrix Multiply(SparseMatrix);
    //…
private:
    int rows, cols, terms, capacity;
    MatrixTerm * smArray;
};
```

} //described in the following slides

Transpose a Sparse Matrix

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

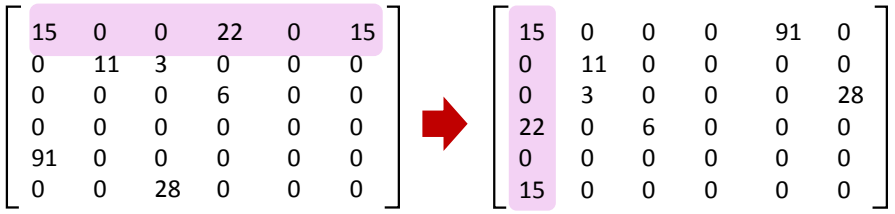
Matrix A

15	0	0	0	91	0
0	11	0	0	0	0
0	3	0	0	0	28
22	0	6	0	0	0
0	0	0	0	0	0
15	0	0	0	0	0

Matrix B = A^T

Row-by-Row Transpose

- We cannot determine the positions in the output array to write to!

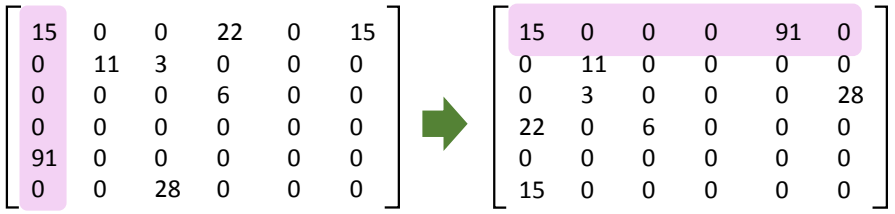


	row	col	value
smArray[0]	0	0	15
[1]	0	3	22
[2]	0	5	15
[3]	1	1	11
[4]	1	2	3
[5]	2	3	6
[6]	4	0	91
[7]	5	2	28

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

Column-by-Column Transpose

- Exploit the fact that we can search the input array for terms belonging to a particular column



	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	col	value
smArray[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	0	6
[7]	5	2	15

Transpose a Sparse Matrix


```
SparseMatrix SparseMatrix::Transpose( )
{
  SparseMatrix b(cols , rows , terms);
  if (terms > 0){
    int currentB = 0;
    for (int c = 0 ; c < cols ; c++)
       for (int i = 0 ; i < terms ; i++)
          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;
}
```

time complexity: O(cols · terms)

Space complexity: O(1)

Transpose a Sparse Matrix (Cont'd)

- We have shown an O(cols · terms)-time algorithm for *Sparse Matrix Transpose*
- Transpose can be done in O(cols · rows) = O(terms) time if the matrix is in traditional 2D array representation

```
for (int y = 0 ; y < cols ; y++)
  for (int x = 0 ; x < rows; x++)
    b[y][x] = a[x][y];
```

- Is there also an O(terms)-time algorithm for *Sparse Matrix Transpose*?

Strategy

- Scan the array for a constant number of passes (e.g., 3 passes)
  - Each pass is O(terms)
  - 3 · O(terms) = O(terms)
- Early pass collects information to assist the following passes
  - Specifically, determine the positions in the output array to write to

Fast Transpose


```
SparseMatrix SparseMatrix::FastTranspose( )
{
  SparseMatrix b(cols , rows , terms);
  if (terms > 0) {
    int *rowSize = new int[cols];
    int *rowStart = new int[cols];


    // calculate the row size of the new matrix
    fill(rowSize, rowSize + cols, 0);
    for (int i = 0 ; i < terms ; i ++)
      rowSize[smArray[i].col]++;


    // calculate the starting array index of each row
    // of the new matrix
    rowStart[0] = 0;
    for (int i = 1 ; i < cols ; i++)
      rowStart[i] = rowStart[i-1] + rowSize[i-1];
```

1st pass, O(terms)

2nd pass, O(cols)

FastTranspose


```
      for (int i = 0 ; i < terms ; i++){
        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

      delete [] rowSize;
      delete [] rowStart;

    } // end of if
    return b;
}
```

3rd pass, O(terms)

Space complexity: O(cols)

Issue of Row-by-Row Transpose

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

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

row	col	value
0	0	15
0	3	22
0	5	15
1	1	11
1	2	3
2	3	6
4	0	91
5	2	28

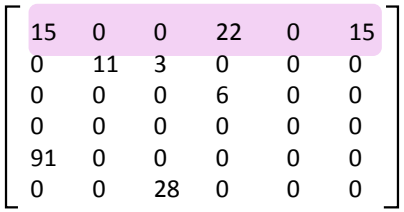
(0, 0) → (0, 0)

(0, 3) → (3, 0) → ?

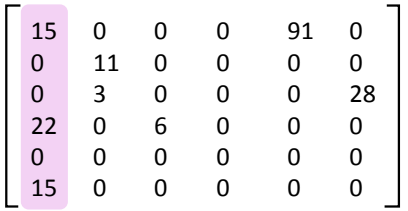
Since rows 0, 1, 2 haven't finish yet, we do not know where row 3 begins!

row	col	value
0	0	15

Fast Transpose Strategy



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



15	0	0	0	91	0
0	11	0	0	0	0
0	3	0	0	0	28
22	0	6	0	0	0
0	0	0	0	0	0
15	0	0	0	0	0

row	col	value
0	0	15
0	3	22
0	5	15
1	1	11
1	2	3
2	3	6
4	0	91
5	2	28

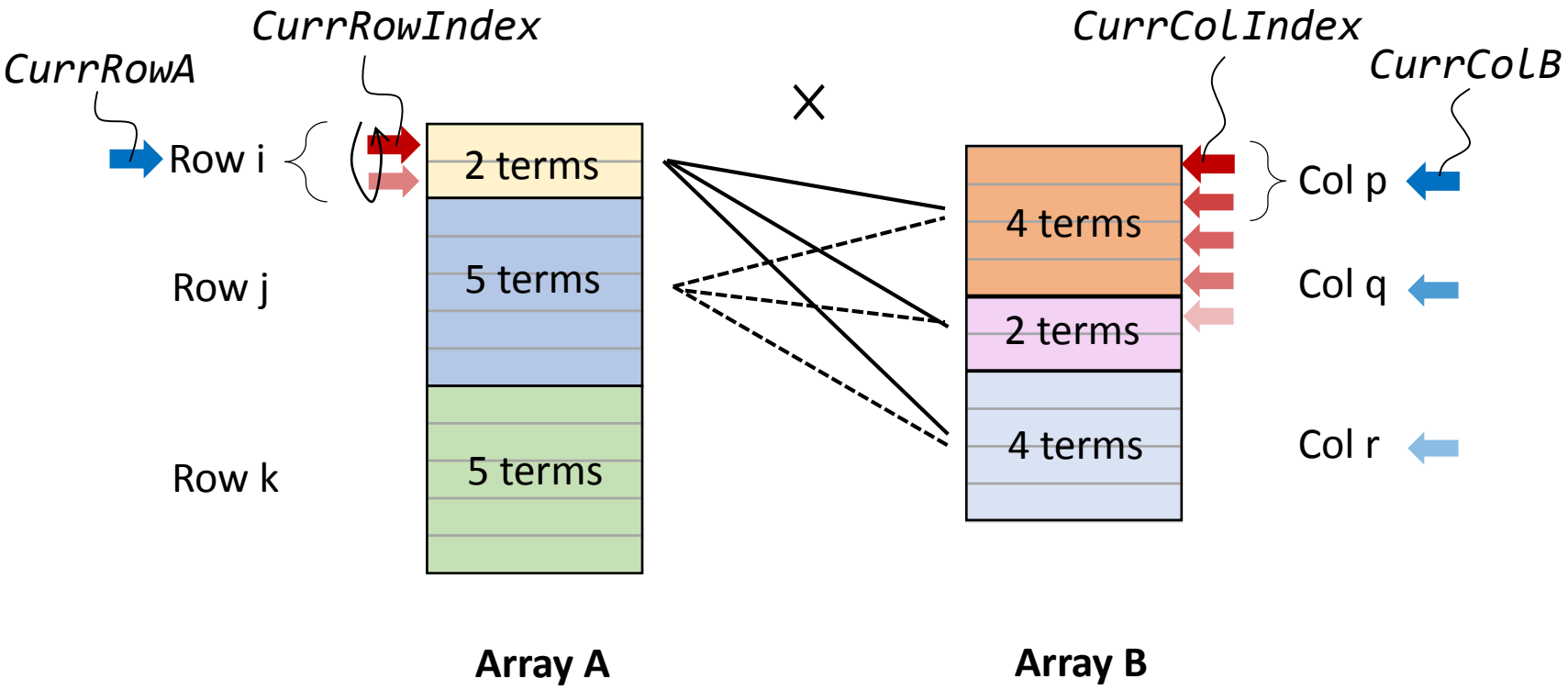
(0, 0) → (0, 0)

(0, 3) → (3, 0)

The starting index of each new row is calculated already!

row	col	value
0	0	15
3	0	22

Multiply Two Arrays

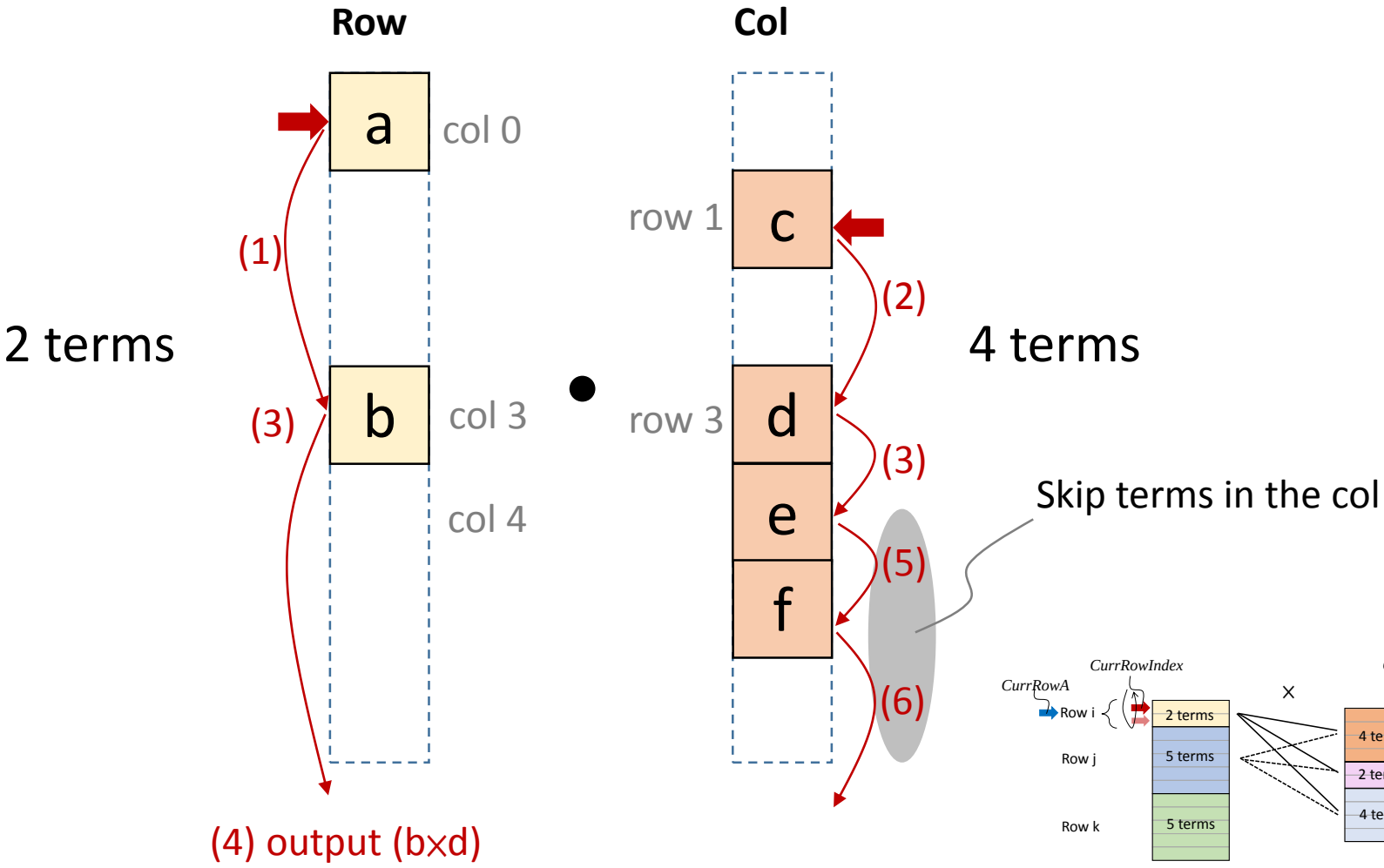


Multiply Two Matrixes


```
// C = A * B
// some preparations

for (all terms in A) {
  for (all terms in B) {
    if (end of the curr row in A) {
       store the sum and reset the sum;
       rewind to the first term of the curr row;
       skip the remain terms of the curr col;
    } else if (end of the curr col in B) {
       store the sum and reset the sum;
       rewind to the first term of the curr row;
    } else {
       accumulate the product of two terms if possible;
    }
  }
  CurrRowA = the next row of matrix A;
 skip the remain terms of the curr row;
}
```

Inner Product of RowA and ColB



Store the Dot Product Sum


```
void SparseMatrix::StoreSum(const int sum, const int r, const int c)
{
    if (sum != 0) {
        if (terms = = capacity)
            ChangeSize1D(2*capacity);
        smArray[terms].row = r;
        smArray[terms].col = c;
        smArray[terms++].value = sum;
    }
}
```

Change Array Size


```
void ChangeSize1D(const int newSize)
{ // change the array size to newSize
    if (newSize < terms)
        throw “New size must be >= number of terms”;

    MatrixTerm *temp = new MatrixTerm[newSize]; // new array

    copy(smArray, smArray + terms, temp);
    delete [] smArray;

    smArray = temp;  // make smArray point to the newly created array
    capacity = newSize;
}
```

Sparse Matrix Multiplication

- Transposing matrix B eases sparse matrix handling
  - Non-zeros terms of a row are stored together

$$\begin{bmatrix} 1 & 2 & 0 \\ 3 & 0 & 0 \\ 3 & 0 & 6 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \\ 0 & 5 & 0 \\ 0 & 0 & 7 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

A B C

$$\begin{bmatrix} 1 & 2 & 0 \\ 3 & 0 & 0 \\ 3 & 0 & 6 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 2 & 5 & 0 \\ 3 & 0 & 7 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

A B^T C

Multiply


```
SparseMatrix SparseMatrix::Multiply(SparseMatrix b)
{
  if (cols != b.rows) // error handling
    throw “Incompatible matrices”;

  SparseMatrix bXpose = b.FastTranspose( );   // transpose b
  SparseMatrix d (rows, b.cols, 0);  // create the output matrix d
  int currRowIndex = 0,
    currRowBegin = 0,
    currRowA = smArray[0].row;

  // introduce dummy terms for handling boundary condition
  if (terms == capacity)
    ChangeSize1D(terms + 1);

  // introduce dummy terms for handling boundary condition
  bXpose.ChangeSize1D(bXpose.terms + 1);
  smArray[terms].row = rows;
  bXpose.smArray[b.terms].row = b.cols;
  bXpose.smArray[b.terms].cols = -1;
```

Multiply (Cont'd)


```
int sum = 0;
while (currRowIndex < terms) { // check currRowA is valid
  int currColB = bXpose.smArray[0].row;
  int currColIndex = 0;
  while (currColIndex <= b.terms){ // process B matrix term by term
    if (smArray[currRowIndex].row != currRowA) {  // row end
      d.StoreSum(sum,currRowA,currColB);   // store the sum
      sum = 0;     // reset the sum
      currRowIndex = currRowBegin; // rewind the row

      while (bXpose.smArray[currColIndex].row == currColB)
        currColIndex++;   // skip terms in the curr col

      currColB = bXpose.smArray[currColIndex].row;  // next col
    } else if (bXpose.smArray[currColIndex].row != currColB) {
      // col end
      d.StoreSum(sum,currRowA,currColB); // output the sum
      sum = 0; // reset the sum
      currRowIndex = currRowBegin; rewind the row
      currColB = bXpose.smArray[currColIndex].row; // next col
    }
```

Multiply (Cont'd)


```
      else {
        if (smArray[currRowIndex].col <
            bXpose.smArray[currColIndex].col)
          currRowIndex++;
        else if (smArray[currRowIndex].col ==
                 bXpose.smArray[currColIndex].col) {
          sum += smArray[currRowIndex].value *
                 bXpose.smArray[currColIndex].value;
          currRowIndex++;
          currColIndex++;
        }
        else
          currColIndex++;
      } // end of if-elseif-else
    } // end of the inner while (currColIndex <= b.terms)
    while (smArray[currRowIndex].row == currRowA)
      currRowIndex++;  // skip terms in the curr row
    currRowBegin = currRowIndex;  //next row
    currRowA = smArray[currRowIndex].rows; //next row
  } // end of the outer while (currRowIndex < terms)
  return d;
}
```

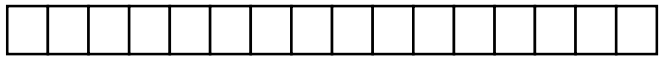
Outline

- 2.1 Abstract Data Types and C++ Class
- 2.2 The Array as an Abstract Data Type
- 2.3 The Polynomial Abstract Data Type
- 2.4 Sparse Matrices
- **2.5 Representation of Arrays**
- 2.6 The String Abstract Data Type

N-Dimensional Array

- Example
  - **float** hour[60]
  - **int** day[24][60]
  - **char** year[365][24][60]
  - **T** a[u_1][u_2][u_3], …, [u_n]

} Common practice uses "**T**" to denote a type

- Memory comprises a linear sequence of bytes
  - 
- Array representation and access
  - Computed index
  - C/C++'s array

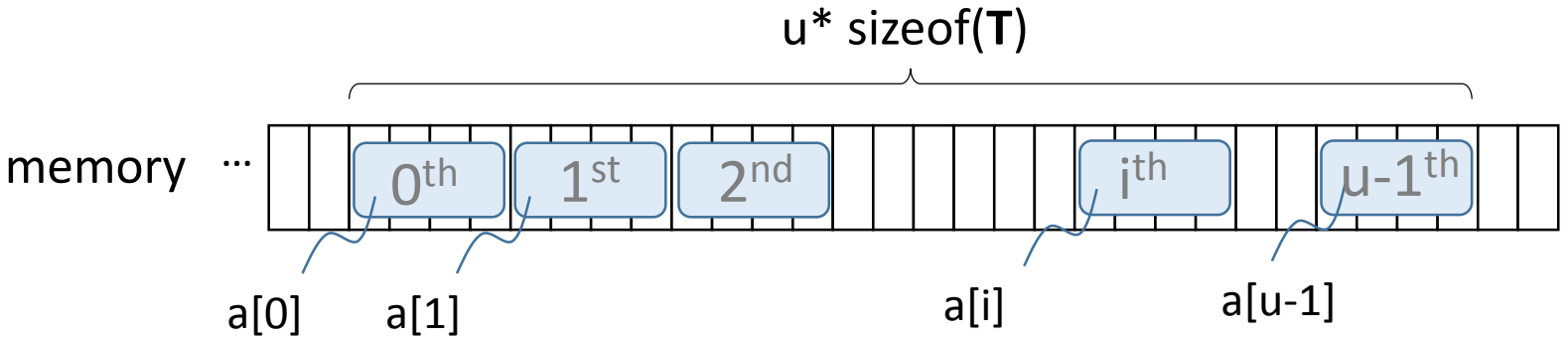
Array Indexing

- 一天24小時、1小時60分鐘、1分鐘60秒鐘
  - 請問 14:20:30 秒是一天的第幾秒?
    - 13 * (60*60) + 19 * (60) + 30 = 47970

- **float** temperature[24][60][60]
  - 請問 14:20:30 的溫度 (temperature[13][19][29]) 在記憶體的哪個 byte address?

1-D Array

- **T** a[u]
  - sizeof(**T**) is *C* bytes (*C* = 4 in the following examples)

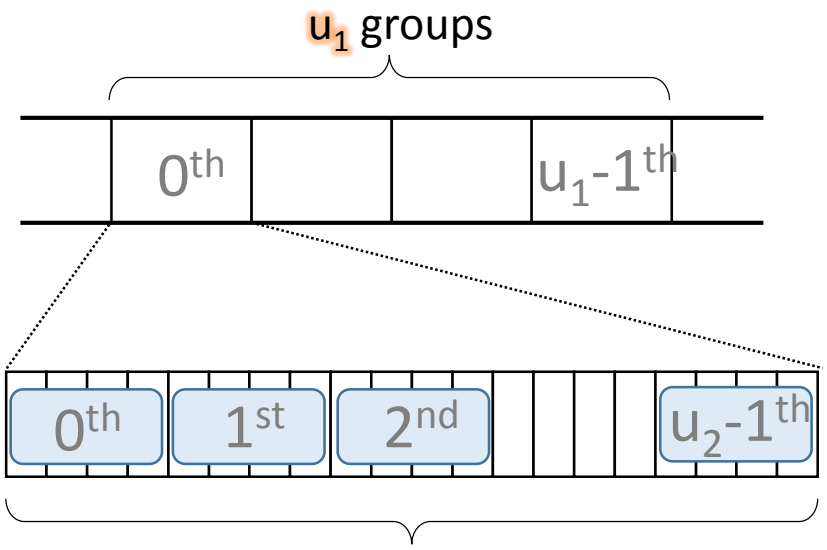


- Addr. of a[i] to access the element =
  - α // base (α denotes the address of first array element)
    + i · *C* // offset

2-D Array

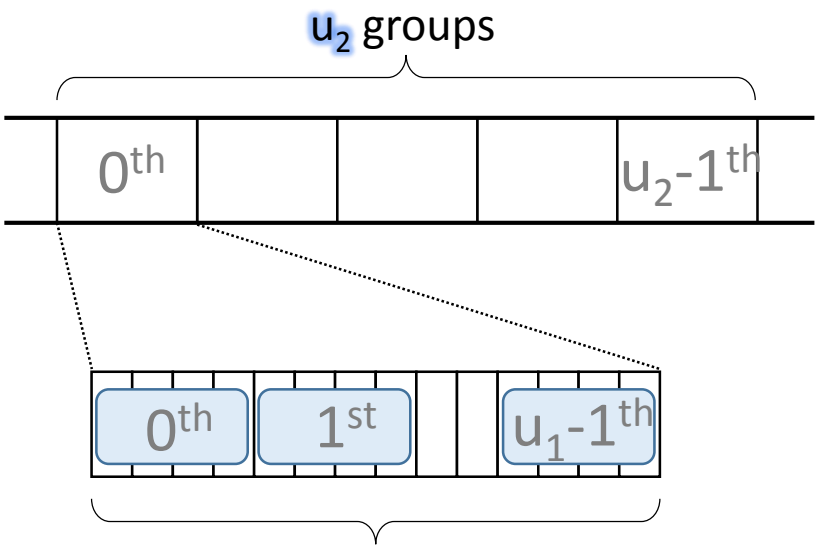
- **T** a[u_1][u_2] (a total of u_1*u_2 elements)

Row Major Order



each group further contains u_2 elements

Column Major Order

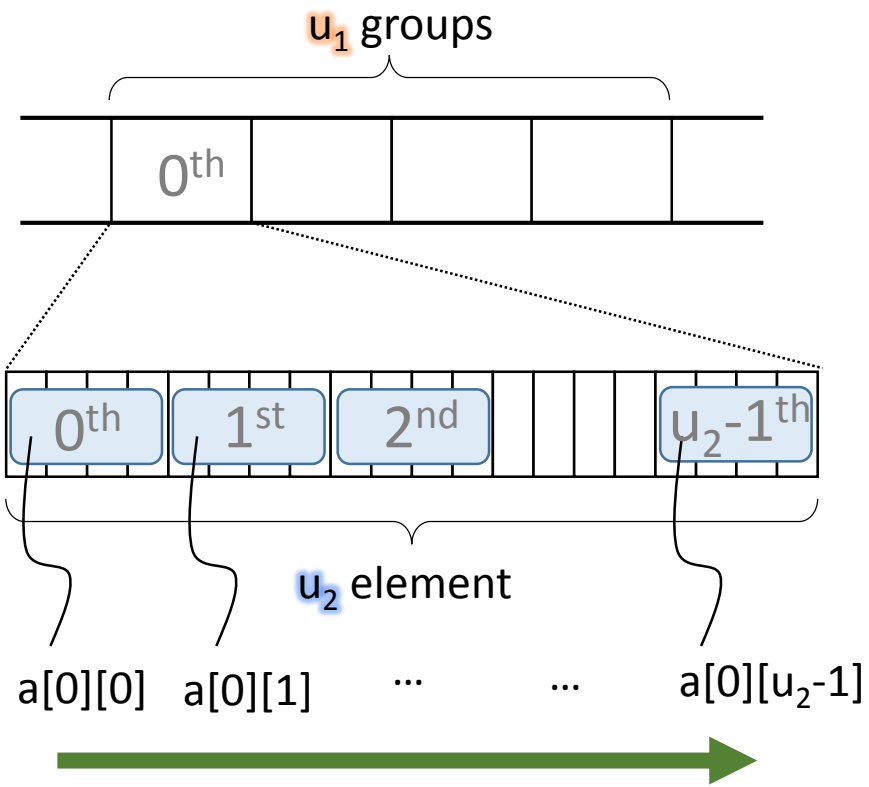


each group further contains u_1 elements

2-D Array

- **T** a[u_1][u_2]

Row Major Order



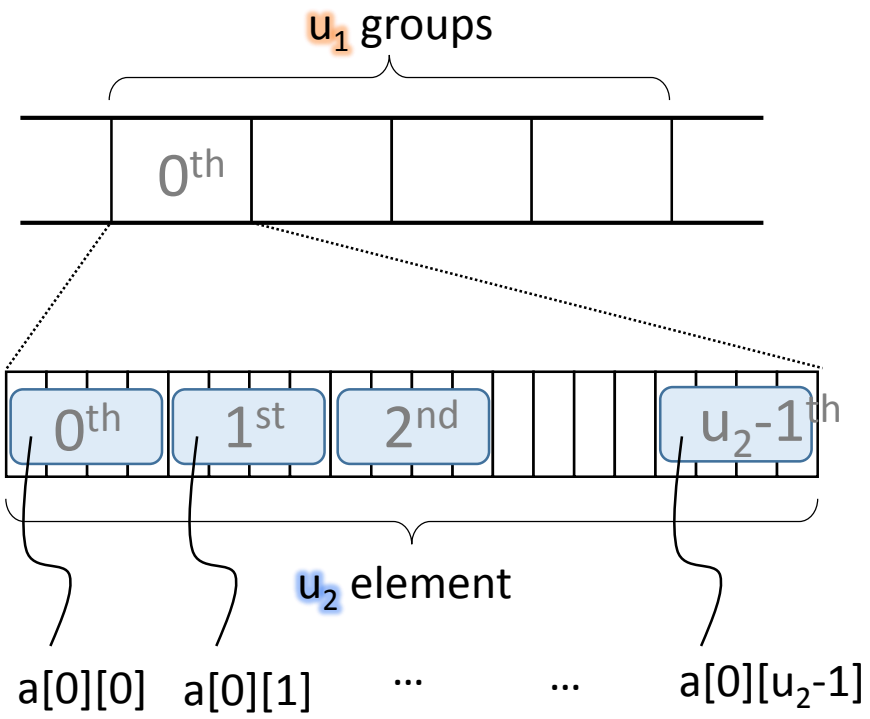
- Elements are placed in memory in
  - lexicographic order (字典順序)
  - also known as **numerical** order

Row major order
= **numerical** order
= **lexicographic** order

2-D Array

- **T** a[u_1][u_2]

Row Major Order

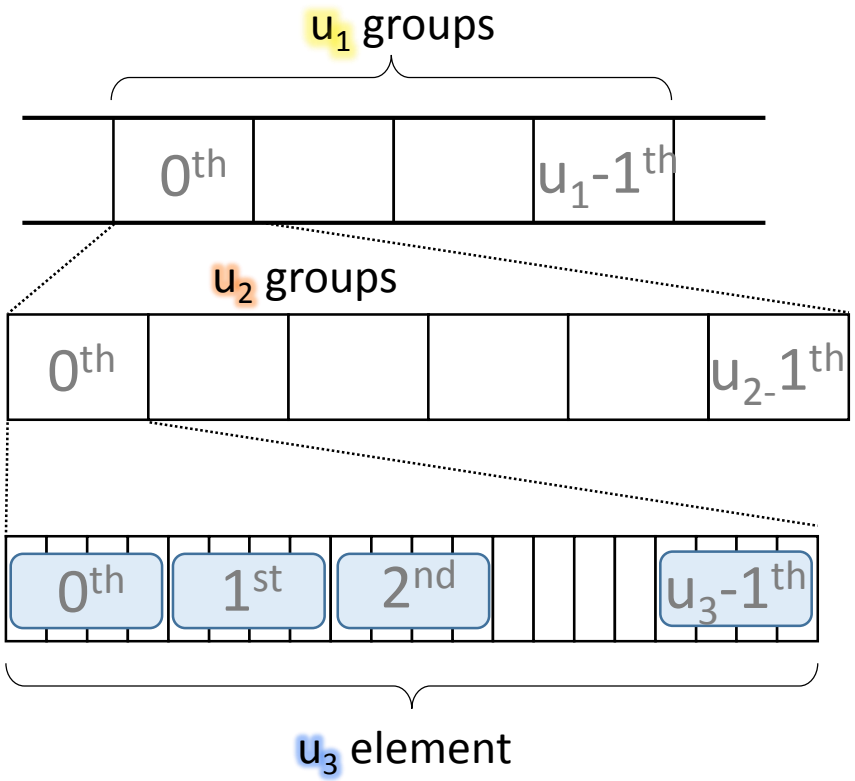


- Addr. of a[i][j]
  - i^{th} group
  - j^{th} position
- Addr. calculation
  - α // base
    + (i · u_2) · C // offset
    + (j) · C // offset

3-D Array

- **T** a[u_1][u_2][u_3] (a total of u_1*u_2*u_3 elements)

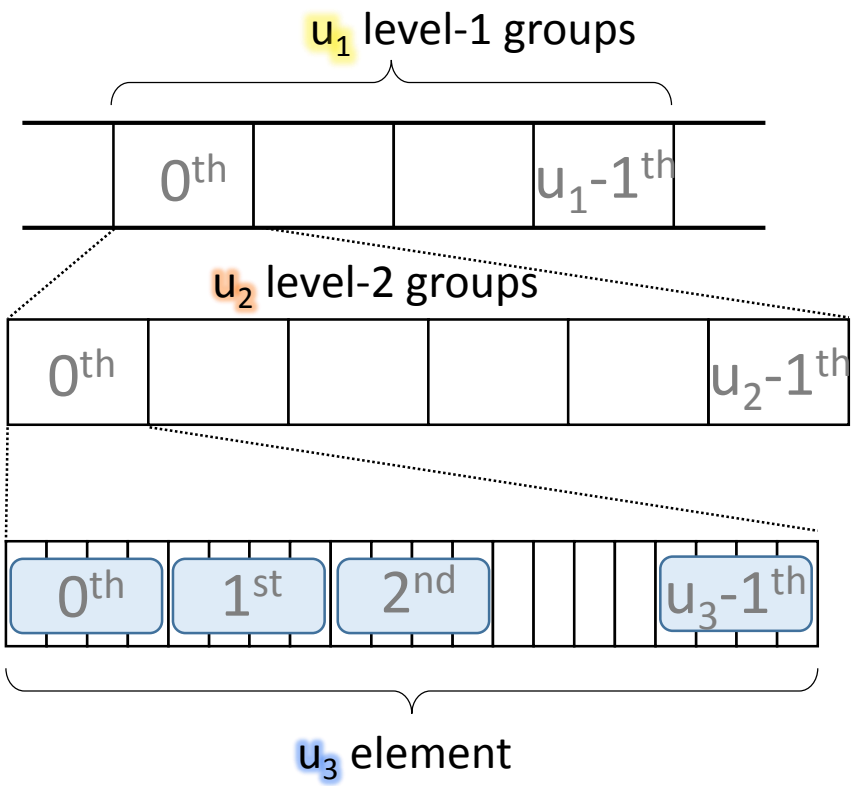
Row Major Order



3-D Array

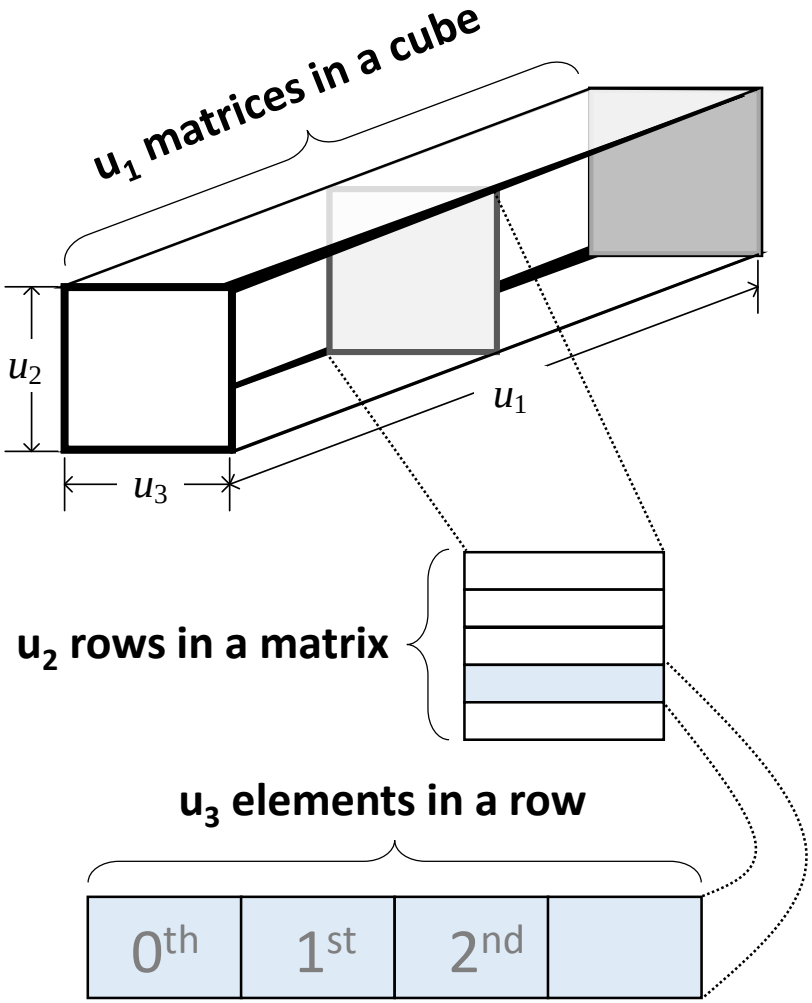
- **T** a[u_1][u_2][u_3]

Row Major Order



- a[i][j][k]
  - i^{th} level-1 group
  - j^{th} level-2 group
  - k^{th} element
- Addr. of a[i][j][k]
  - α // base
    + $(i \cdot u_2u_3) \cdot C$ // offset
    + $(j \cdot u_3) \cdot C$ // offset
    + $(k) \cdot C$ // offset

3-D Array (Textbook's Illustration)

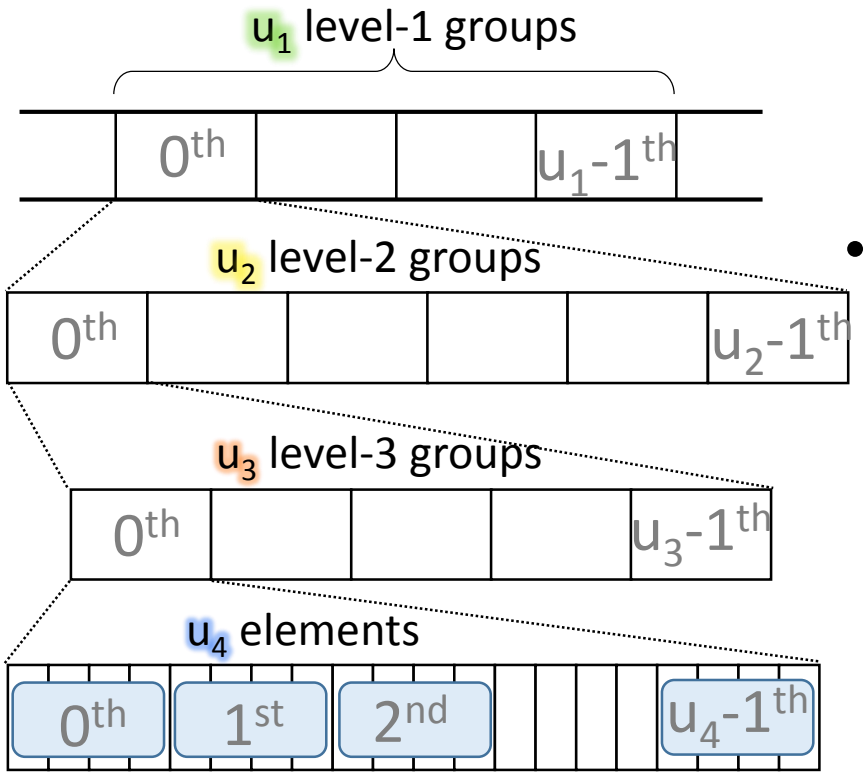


- Emphasize what "row-major" means
- But in fact, memory is linear instead of cubic

4-D Array

- **T** a[u_1][u_2][u_3][u_4]

Row Major Order



- a[i][j][k][m]
  - m^{th} level-1 group
  - i^{th} level-2 group
  - j^{th} level-3 group
  - k^{th} element

- Addr. of a[i][j][k][m]
  - α // base
    $+ (i \cdot u_2u_3u_4) \cdot C$ // offset
    $+ (j \cdot u_3u_4) \cdot C$ // offset
    $+ (k \cdot u_4) \cdot C$ // offset
    $+ (m) \cdot C$ // offset

N-Dimension Array

- **T** a[u_1][u_2][u_3], …, [u_n]
  - Total number of elements
    - $u_1 * u_2 \cdot \cdots * u_n = \prod_{j=1}^{n} u_j$
  - Total memory usage
    - $(\prod_{j=1}^{n} u_j) \cdot \text{sizeof(T)}$
  - Addr. of a[i_1][i_2][i_3], …, [i_n]
    - α // base
      + ($i_1 u_2 u_3 \ldots u_n$) · C // offset
      + ($i_2\ u_3 \ldots u_n$) · C // offset
      + …
      + ($i_{n-1} u_n$) · C // offset
      + (i_n) · C // offset

      $= \alpha + \sum_{j=1}^{n} i_j a_j$ where $a_j = u_{j+1} \cdot a_{j+1} = \begin{cases} \prod_{k=j+1}^{n} u_k & , 1 \le j < n \\ C & , j = n \end{cases}$

N-Dimension Array

- **T** a[u_1][u_2][u_3], …, [u_n]
  - Addr. of a[i_1][i_2][i_3], …, [i_n]
    - α
      + ($i_1 u_2 u_3 \ldots u_n$) · C
      + ($i_2\ u_3 \ldots u_n$) · C
      + …
      + ($i_{n-1} u_n$) · C
      + (i_n) · C

      = $\alpha + \sum_{j=1}^{n} i_j a_j$

Observations

- $u_2 u_3 \ldots u_n$, i_1, i_2, …, i_n, α, and C are musts, but u_1 is not required

C/C++'s Arrays

- Representation and access methods differ for the following two array types

- Statically-allocated arrays
  - Computed index

- Dynamically-allocated arrays
  - Chain of pointers

- We will discuss this in the future, time permitting.

Statically-Allocated Arrays


```
int main()
{
    float num[12][31][24];

    cout << sizeof(num[2][19][14]) << endl;
    cout << sizeof(num[2][19]) << endl;
    cout << sizeof(num[2]) << endl;
    cout << sizeof(num) << endl;

    return 0;
}
```

```
4
96
2976
35712
```

Some people refer to this design as **array of arrays**

Static Array as an Argument


```
int main()
{
    float num[12][31][24];
    clearAll(num);
    return 0;
}
```

- Syntax for array arguments

```
void clearAll(float a[][31][24])
{
    for(int m=0; m<12; m++)
        for(int d=0; d<31; d++)
            for(int h=0; h<24; h++)
                a[m][d][h] = 0;
    return;
}
```

- Syntax for array arguments
  - Specifying the size of the first dimension is optional
  - Specifying the sizes of the other dimensions is a must

- Address calculation
  - u_1 is optional
  - $u_2 \ldots u_n$ are musts

Array arguments allow *side effects*: assignments to the array are visible to the calling function.

Static Array as an Argument


```
int main()
{
    float num[12][31][24];
    // a specific month or day is a sub-array
    clearMonth(num[2]);                    clearDay(num[5][19]);
    return 0;
}
```

```
void clearMonth (float a[][24])
{
    for(int d=0; d<31; d++)
        for(int h=0; h<24; h++)
            a[m][d][h] = 0;
    return;
}
```

```
void clearDay (float a[])
{
    for(int h=0; h<24; h++)
        a[m][d][h] = 0;
    return;
}
```

Outline

- 2.1 Abstract Data Types and C++ Class
- 2.2 The Array as an Abstract Data Type
- 2.3 The Polynomial Abstract Data Type
- 2.4 Sparse Matrices
- 2.5 Representation of Arrays
- **2.6 The String Abstract Data Type**

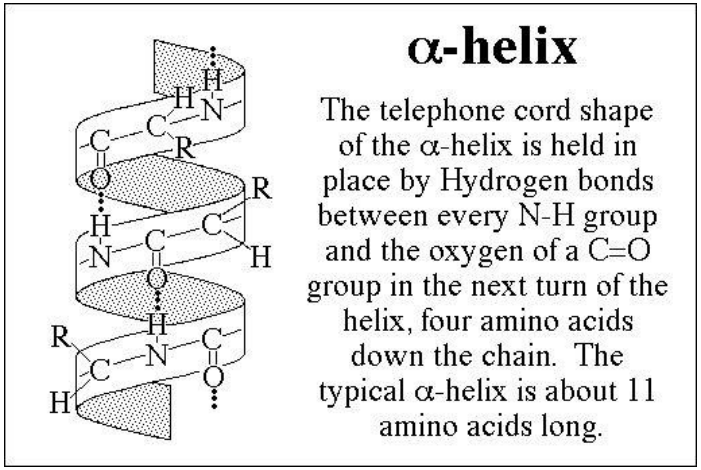
ADT String


```
class String
{
public:
    String(char *init, int m);
    // constructor using input string init of length m
    bool operator == (String t);  //equality test, true or false
    bool operator!( );  // empty test, true or false
    int Length( );  // get the number of characters of the object
    String Concat(String t);  // concatenation with another string t
    String Substr(int i, int j);  // generate a substring
    int Find(String pat);
    // Return an index i such that pat matches the substring of the
    // object begins at position i.  Return -1 is pat is empty or not
    // a substring of the object
}
```

String Matching

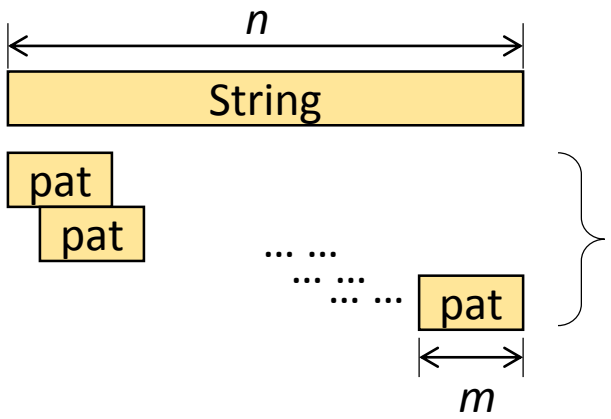
- Important class of algorithm

- Applications
  - Search engines
  - Natural language processing
  - Bioinformatics
    - Identification of α-helices in protein sequences
    - Matching profiles or probabilistic sequences



Exhaustive String Matching

```
int String::Find(String pat)
{
  for (int start = 0; start <= Length( ) - pat.Length(); start ++) {
    int j;
    for (j = 0; j < pat.Length( ) && str[start+j] == pat.str[j]; j++);
    if (j == pat.Length( ))
      return start;
  }
  return -1 ;
}
```



Outer loop iterates O(n) times

Total time complexity is O(n·m)

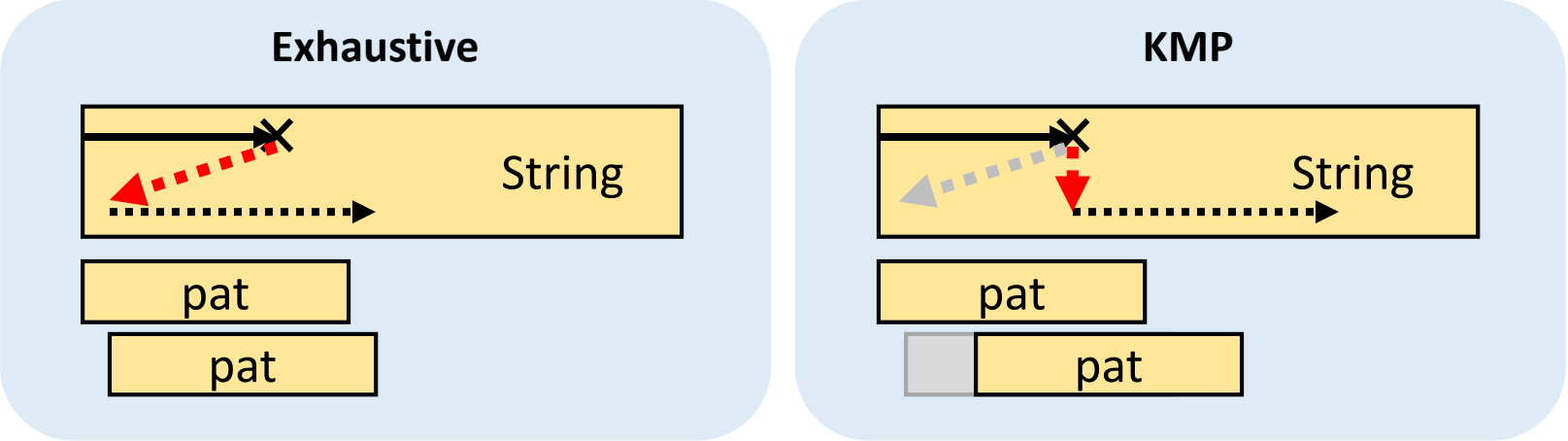
String Matching

- Lower bound of the time complexity
  - $O(n + m)$
  - In the worst case it is necessary to look at all characters in the two strings at least once
- In comparison, exhaustive string matching is way much costlier
  - $O(n \times m)$
  - Can prohibit us from performing searching on large databases or long protein sequences
- Does *String Matching Problem* belong to the $O(n + m)$ class*?*

Knuth-Morris-Pratt (KMP) Algorithm

- O($m + n$)-time

- Strategy to achieve linear time complexity
  - Prevent the algorithm from moving backwards in the string
  - Exploit the knowledge of a mismatch to determine where we should resume the search



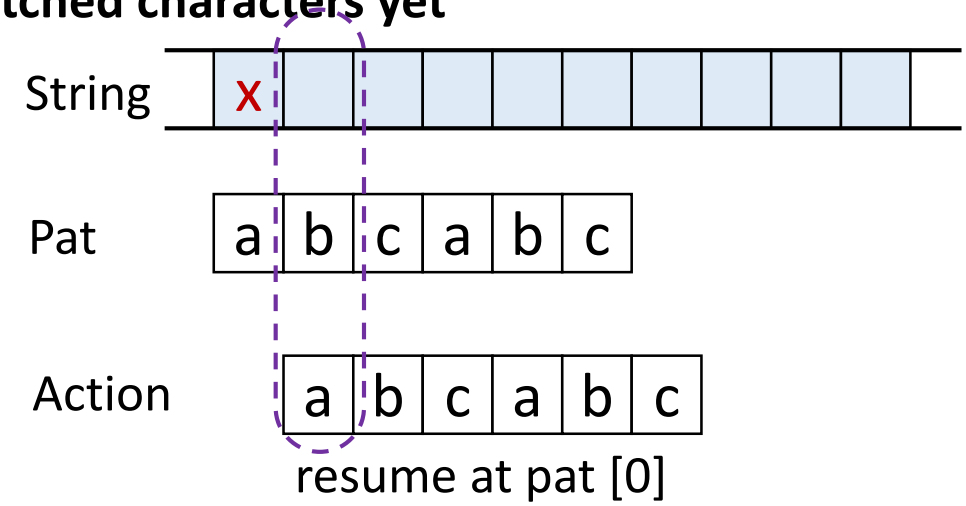
KMP Example

- Searching a string for a pattern "abcabc"

- Since the string consists of 6 characters, we need to handle a total of 7 cases
  1. No matched characters yet
  2. Last match at pat[0] (pat[1] is a mismatch)
  3. Last match at pat[1] (pat[2] ….
  4. Last match at pat[2] (pat[3] ….
  5. Last match at pat[3] (pat[4] ….
  6. Last match at pat[4] (pat[5] ….
  7. Last match at pat[5] (i.e, a matched pattern is found)

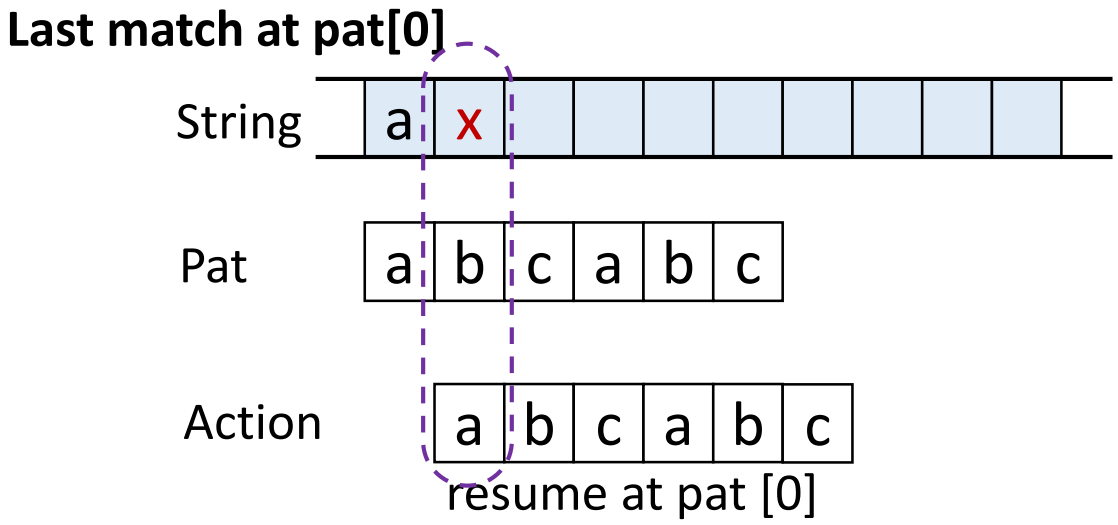
KMP Example

No matched characters yet



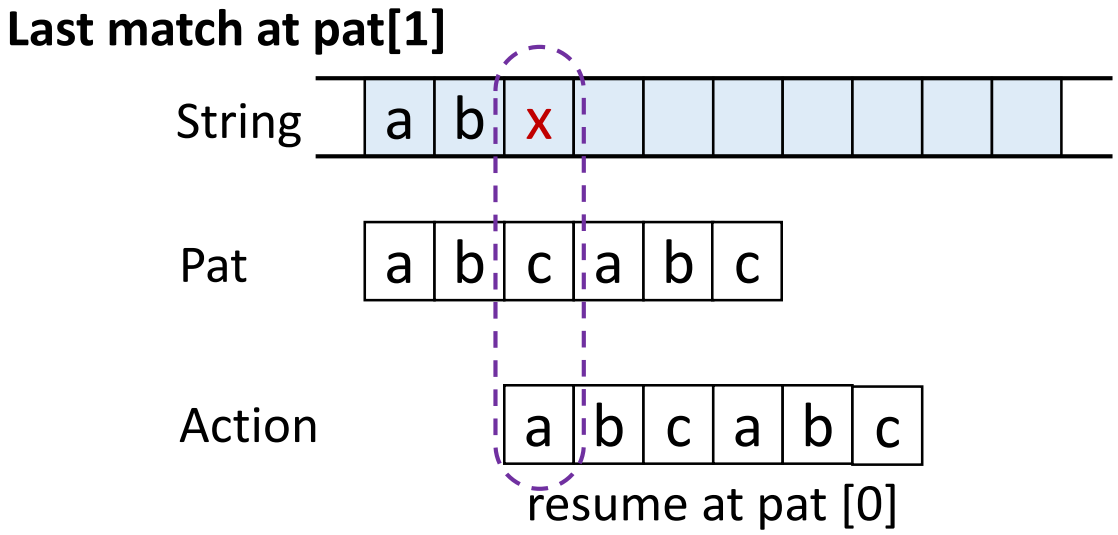
In this case, KMP behaves the same as exhaustive search

KMP Example



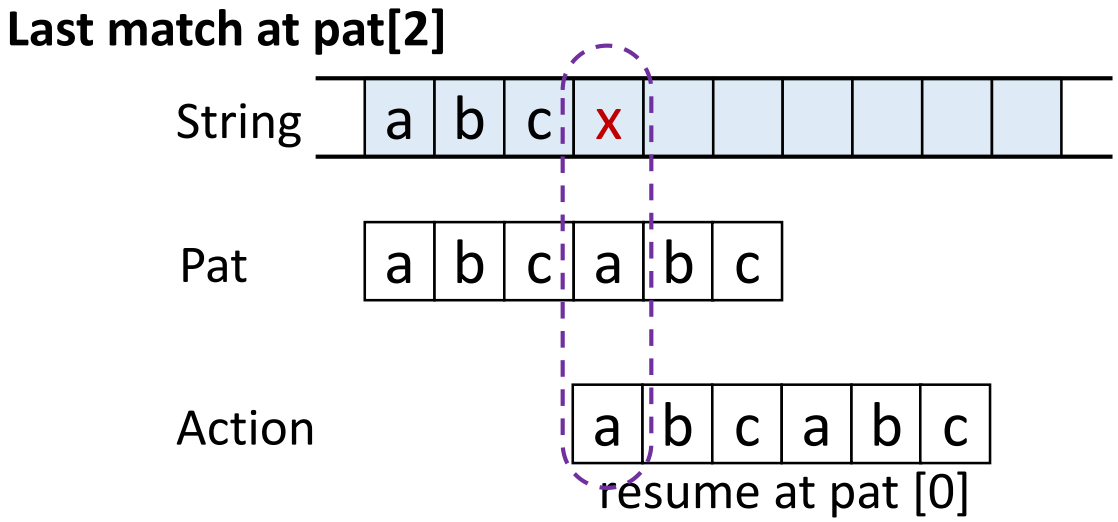
In this case, KMP behaves the same as exhaustive search

KMP Example



In this case, KMP is better than exhaustive search

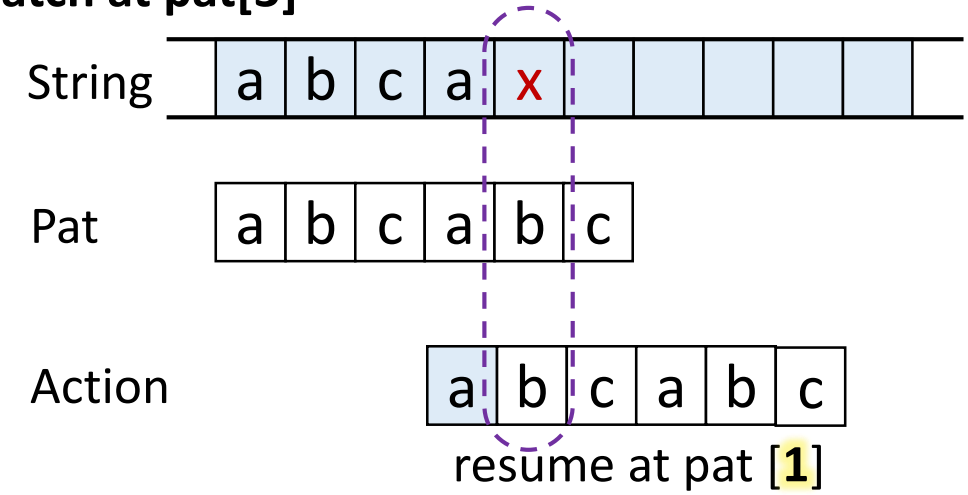
KMP Example



In this case, KMP is better than exhaustive search

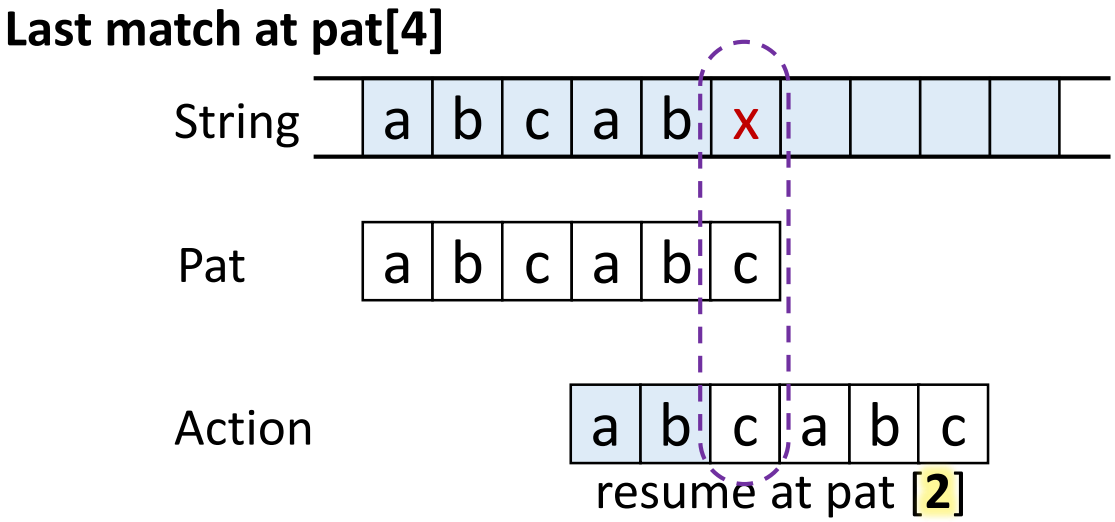
KMP Example

Last match at pat[3]



In this case, KMP is better than exhaustive search

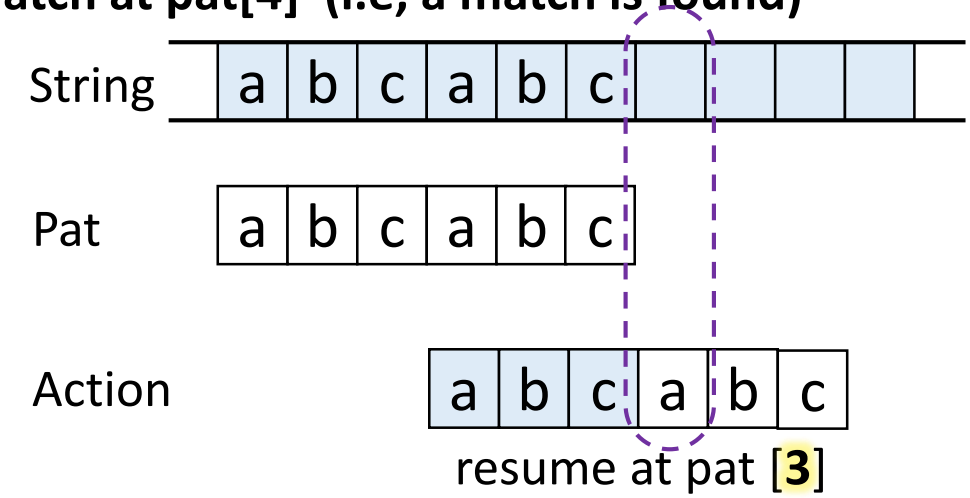
KMP Example



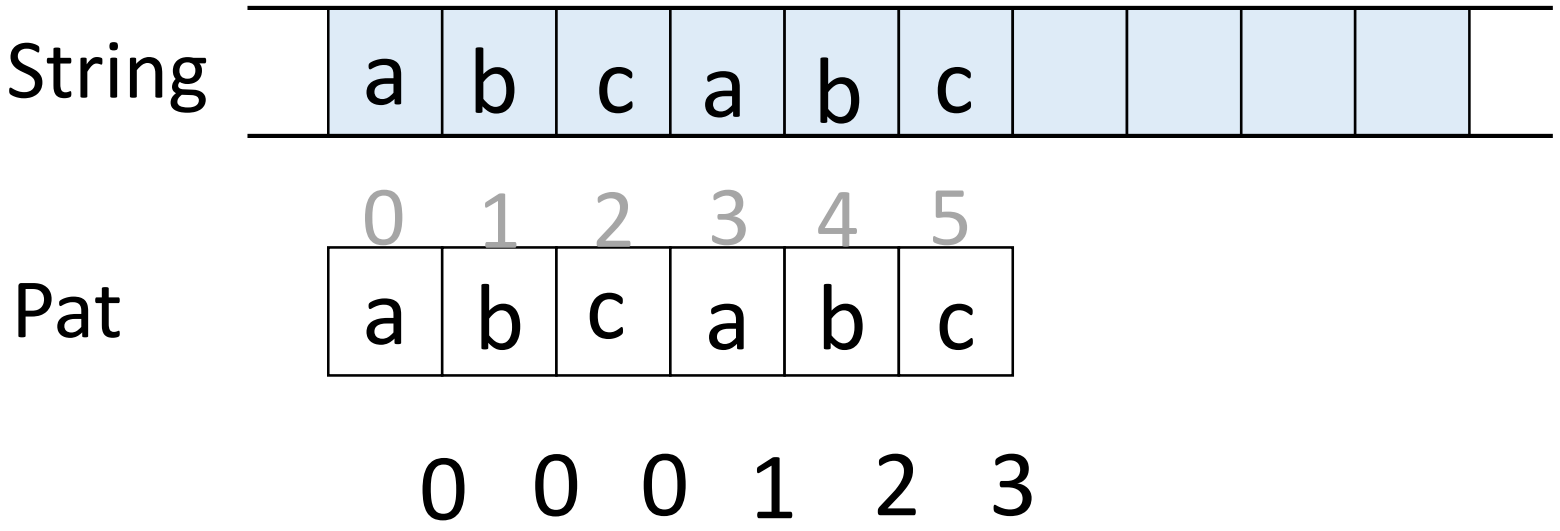
In this case, KMP is better than exhaustive search

KMP Example

Last match at pat[4] (i.e, a match is found)



In this case, KMP is better than exhaustive search


String
a b c a b c
0 1 2 3 4 5
Pat
a b c a b c
0 0 0 1 2 3

KMP Example Summary

- No matched characters
  - Advance the searching point in the string
  - Resume at pat[0]
- Others
  - Keep the searching point in the string
  - Table look-up to determine the pattern resume point

- Table contents depend on the pattern (instead of the string)

Pattern:	a	b	c	a	b	c
Resume point vs last match:	0	0	0	1	2	3

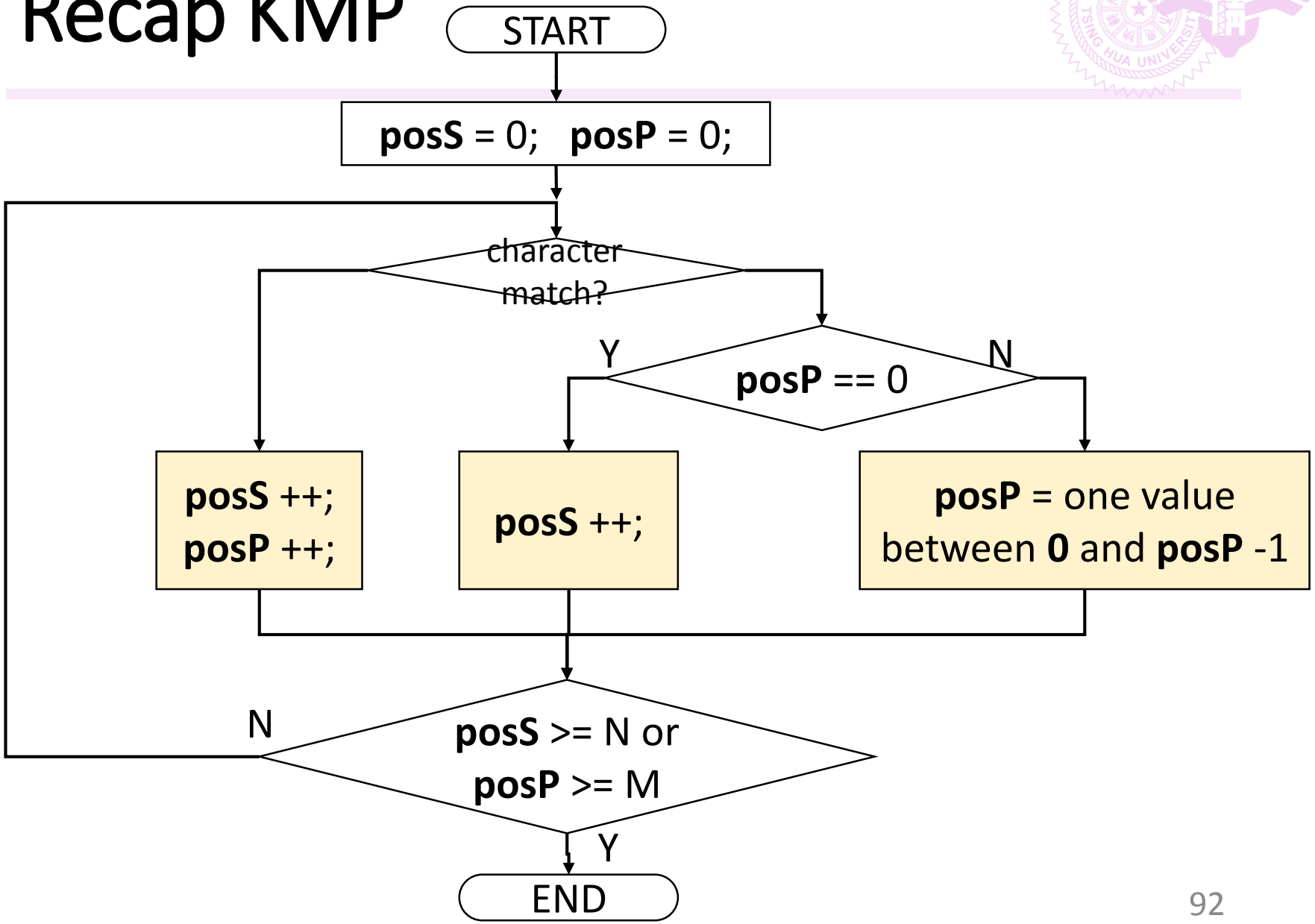
Textbook refers to an equivalent table as **failure function**	-1	-1	-1	0	1	2

Fast Find Using the KMP Algorithm

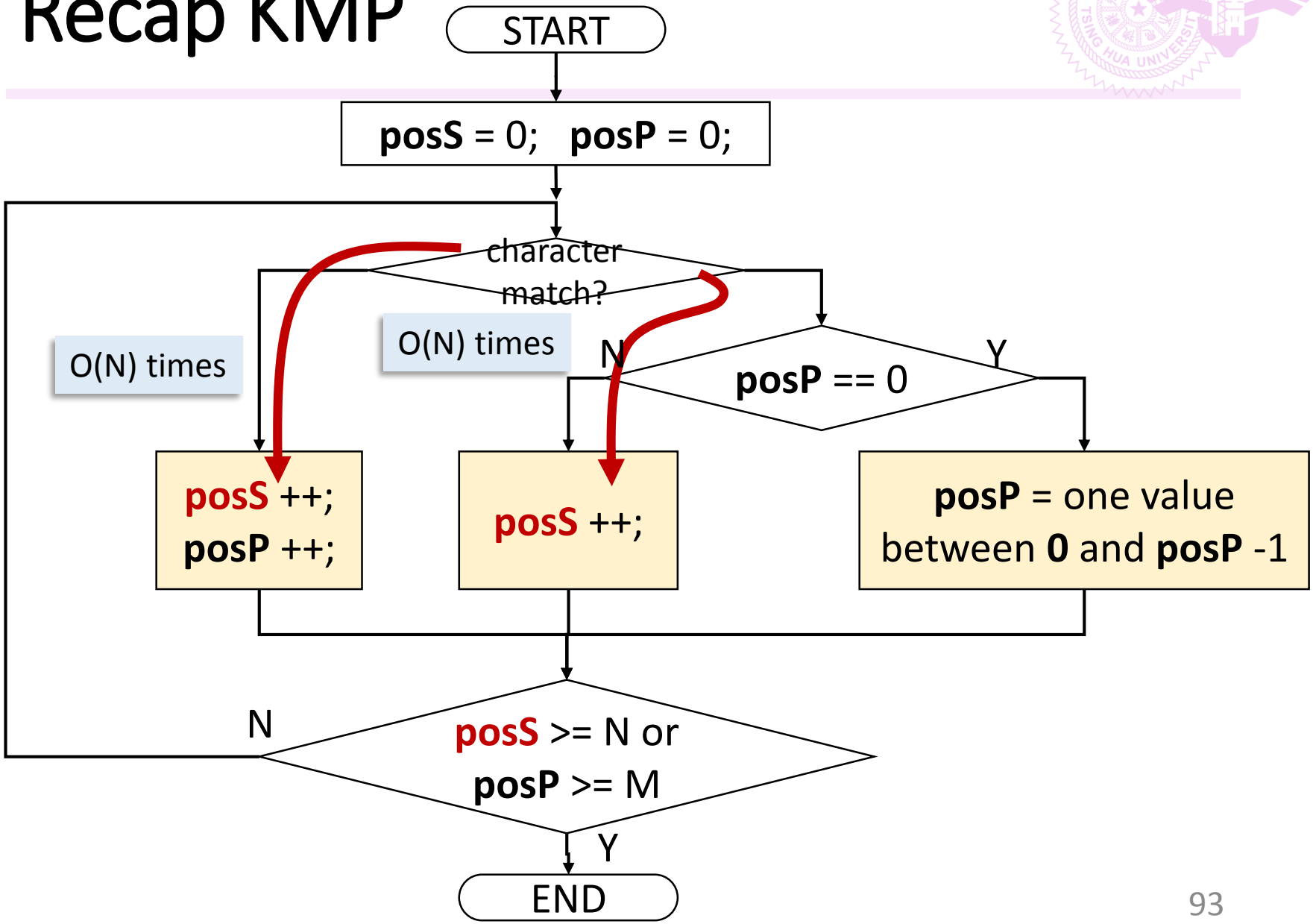
```
int String::FastFind(String pat)
{
  int posP = 0, posS = 0;
  int lengthP = pat.Length(), lengthS = Length();
  while((posP < lengthP) && (posS < lengthS))
    if (pat.str[posP] == str[posS]) {
      posP++;
      posS++;
    }else{
      if (posP == 0)
        posS++;
      else
        posP = pat.failure_function[posP－1] + 1;
    }
  if (posP < lengthP)
      return -1;
  else // Textbook stops pattern matching once a match is found
      return posS-lengthP;
}
```

time = O(lengthS)

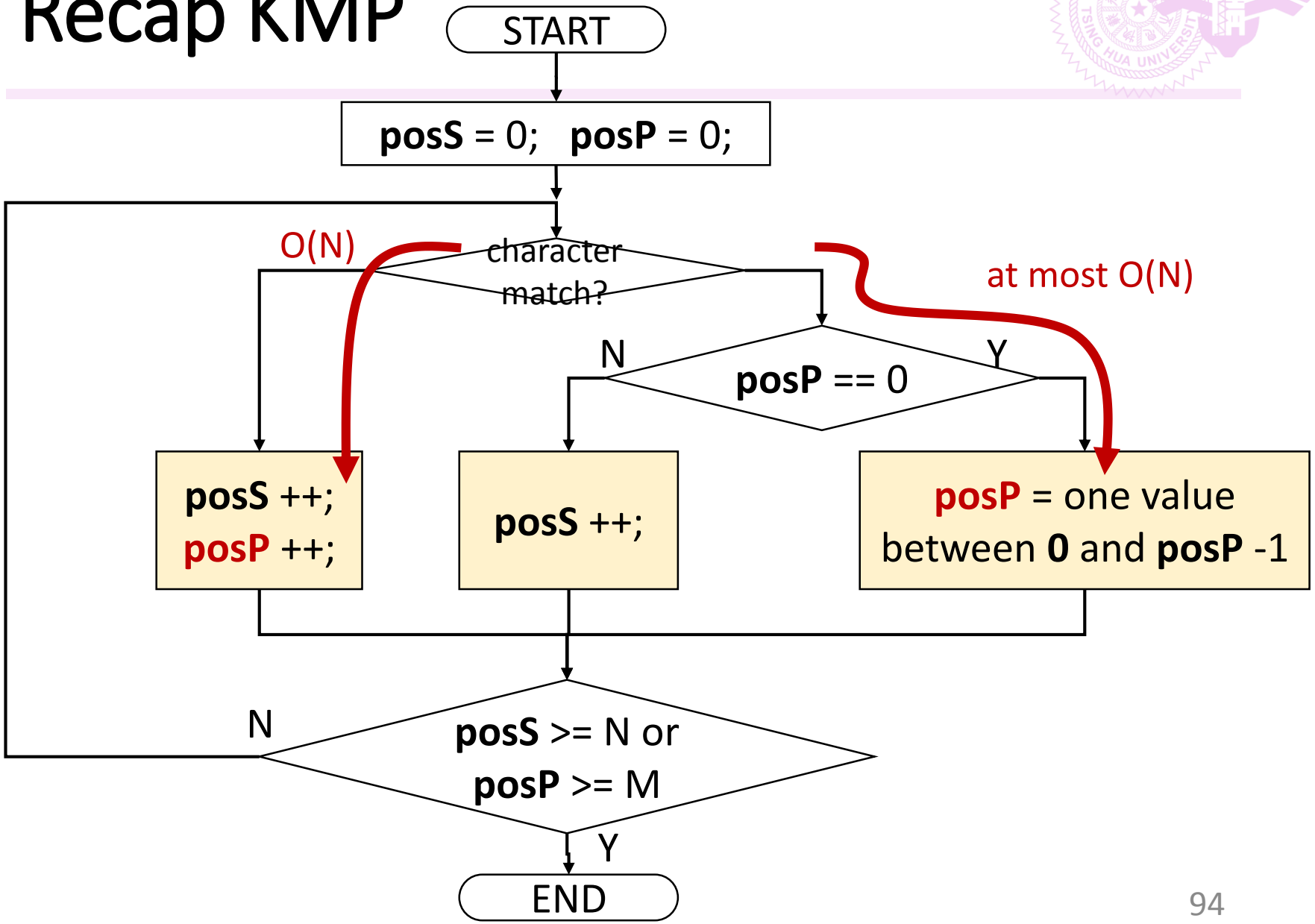
Recap KMP


START
posS = 0; posP = 0;
character match?
Y
N
posP == 0
posS ++;
posP ++;
posS ++;
posP = one value between 0 and posP -1
N
posS >= N or posP >= M
Y
END

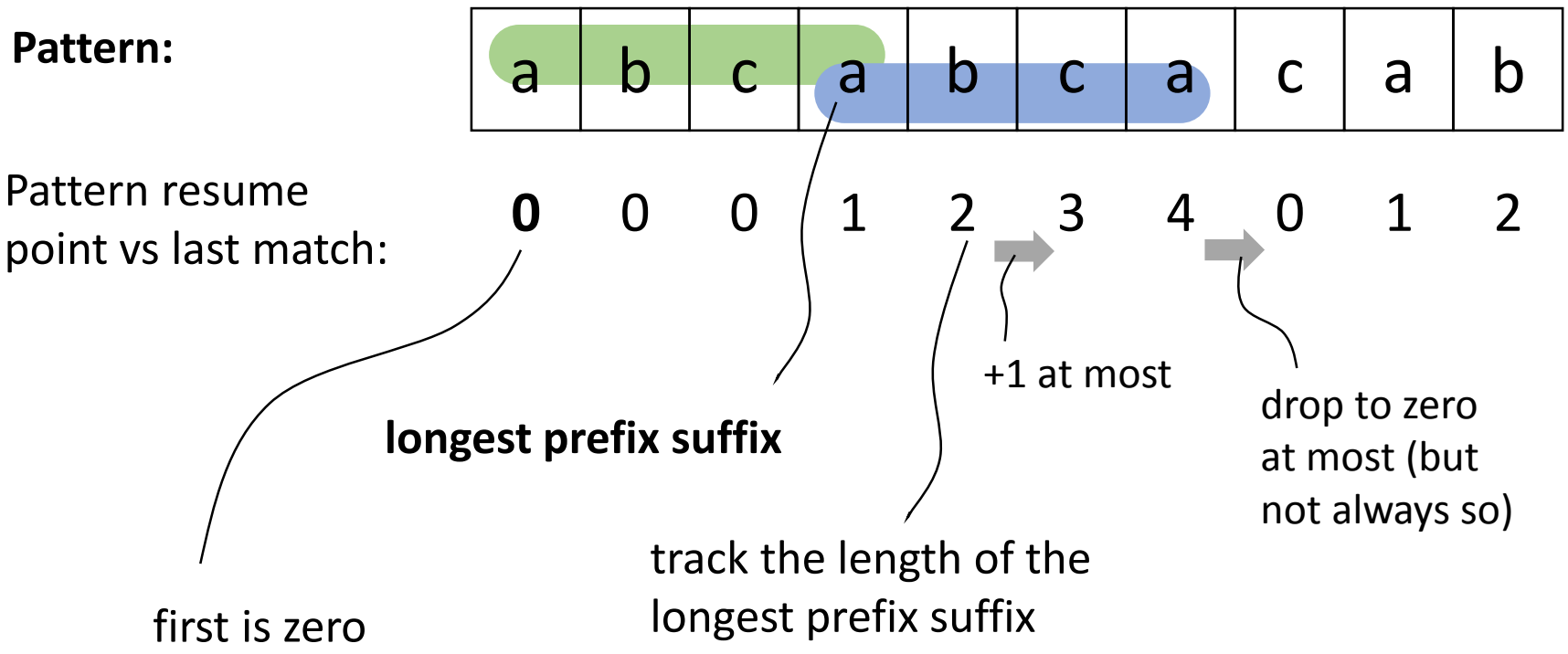
Recap KMP


START
posS = 0; posP = 0;
character match?
O(N) times
O(N) times
N
posP == 0
Y
posS ++;
posP ++;
posS ++;
posP = one value between 0 and posP -1
N
posS >= N or posP >= M
Y
END

Recap KMP

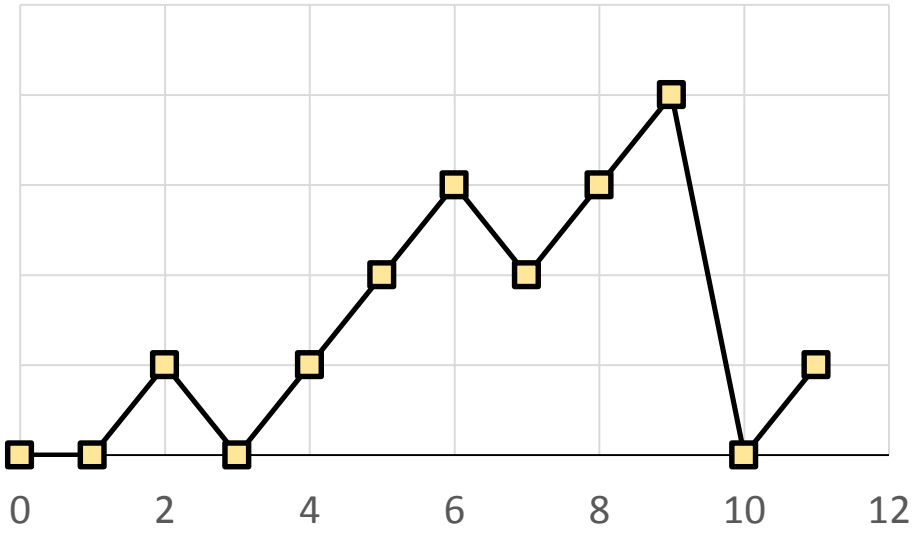

START
posS = 0; posP = 0;
O(N)
character match?
at most O(N)
N
posP == 0
Y
posS ++;
posP ++;
posS ++;
posP = one value between 0 and posP -1
N
posS >= N or posP >= M
Y
END

More KMP Explanations



More KMP Explanations

Pat:	a	b	a	c	a	b	a	b	a	c	d	a
	0	0	1	0	1	2	3	2	3	4	0	1



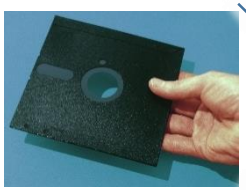
Failure Function Calculation


```
void String::FailureFunction()
{
    int lengthP = Length( );
    f[0] = -1;
    for (int j = 1; j < lengthP; 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;
    }
}
```

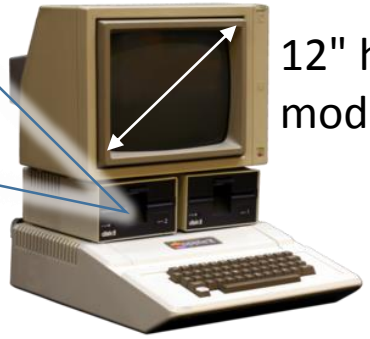
time = O(lengthP)

Time When the KMP Alg. Was Created

- KMP was published in a journal paper in 1977



5.25" floppy
160KB~1.2 MB



12" high resolution
mode: 280×192

	Apple II (1977)		Apple iMac (2015)
		38 yrs →	
Memory	4~64 KB	million × →	8GB
Frequency	1MHz	thousands × →	2~4GHz
Exhaustive	O(mn)	= →	O(mn)
KMP	O(m+n)	= →	O(m+n)