Chapter 2

Arrays and structures

All the programs in this file are selected from

Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
"Fundamentals of Data Structures in C",

Outline

Arrays, Structures, and Unions

2.1, 2.2, 2.3; page 51 - 63

Arrays

Array: a set of index and value

data structure

For each index, there is a value associated with that index.

representation

implemented by using <u>consecutive memory</u>.

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

	0	1	2	3	4
List					

ADT2.1: Abstract Data Type *Array*

```
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 ith element is the size of the
                             ith 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
```

Arrays 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 integers
plist[0], plist[1], plist[2], plist[3], plist[4]

implementation of 1-D array

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

Arrays in C *(Continued)*

Compare int *list1 and int list2[5] in C.

Same: list1 and list2 are (**pointers).**
Difference: list2 reserves five locations.

Notations:
list2: a pointer to list2[0]
(list2 + i): a pointer to list2[i] (&list2[i])
*(list2 + i): (list2[i])

Example: 1-dimension array addressing

int one[] = {0, 1, 2, 3, 4};

Goal: print out address and value

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

call print1(&one[0], 5)

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

*Figure 2.1: One- dimensional array addressing

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;
```

Unions

Similar to struct, but only one field is **active**.

Example: Add fields for male and female.

```
typedef struct gender_type {
    enum tag_field {female, male} gender;
    union {
        int children;
        int beard;
        } u;
    };
typedef struct human_being {
    char name[10];
    int age;
    float salary;
    date dob;
    gender_type gender_info;
    }
```

```
human_being person1, person2;
person1.gender_info.gender=male;
person1.gender_info.u.beard=FALSE;
```

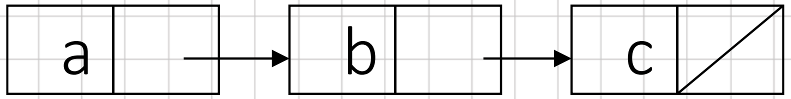
Self-Referential Structures

One or more of its components is a pointer to itself.

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

Construct a list with three nodes
item1.link=&item2;
item2.link=&item3;
malloc(): obtain a node
free(): free memory

```
list item1, item2, item3;
item1.data='a';
item2.data='b';
item3.data='c';
item1.link=item2.link=item3.link=NULL;
```



Polynomials

2.4 page64 - 71

Ordered List Examples

ordered (linear) list: (item1, item2, item3, ..., item *n*)

- (MONDAY, TUEDSAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAYY, SUNDAY)
- (2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace)
- (1941, 1942, 1943, 1944, 1945)
- (a1, a2, a3, ..., an-1, an)

Operations on Ordered List

- Find the length, n , of the list.
- Read the items from left to right (or right to left).
- Retrieve the ith element.
- Store a new value into the ith position.
- Insert a new element at the position i , causing elements numbered i, i+1, …, n to become numbered
  i+1, i+2, …, n+1
- Delete the element at position i , causing elements numbered i+1, …, n to become numbered i, i+1, …, n-1
  array (sequential mapping)?

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

Structure *Polynomial* is

objects: $p(x) = a_1 x^{e_1} + \dots + a_n x^{e_n}$; 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
                                                $poly \bullet coef \bullet x^{expon}$
Polynomial Add(poly1, poly2)                  ::= return the polynomial
                                                $poly1 + poly2$
Polynomial Mult(poly1, poly2)                 ::= return the polynomial
                                                $poly1 \bullet poly2$
```

End *Polynomial*

***ADT2.2:**Abstract data type *Polynomial*

Polynomial Addition

$A(X)=3X^{20}+2X^{5}+4$

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

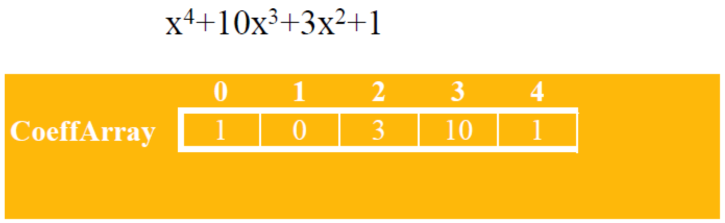
$C(X) = A(X) + B(X)$ ， $C(X) = ?$

Polynomial Addition

```
/* 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;
```

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

data structure 1:


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

	0	1	2	3	4
CoeffArray	1	0	3	10	1

```
#define MAX_DEGREE 101 (100 + 1)
typedef struct {
        int degree;
        float coef[MAX_DEGREE];
        } polynomial;
```

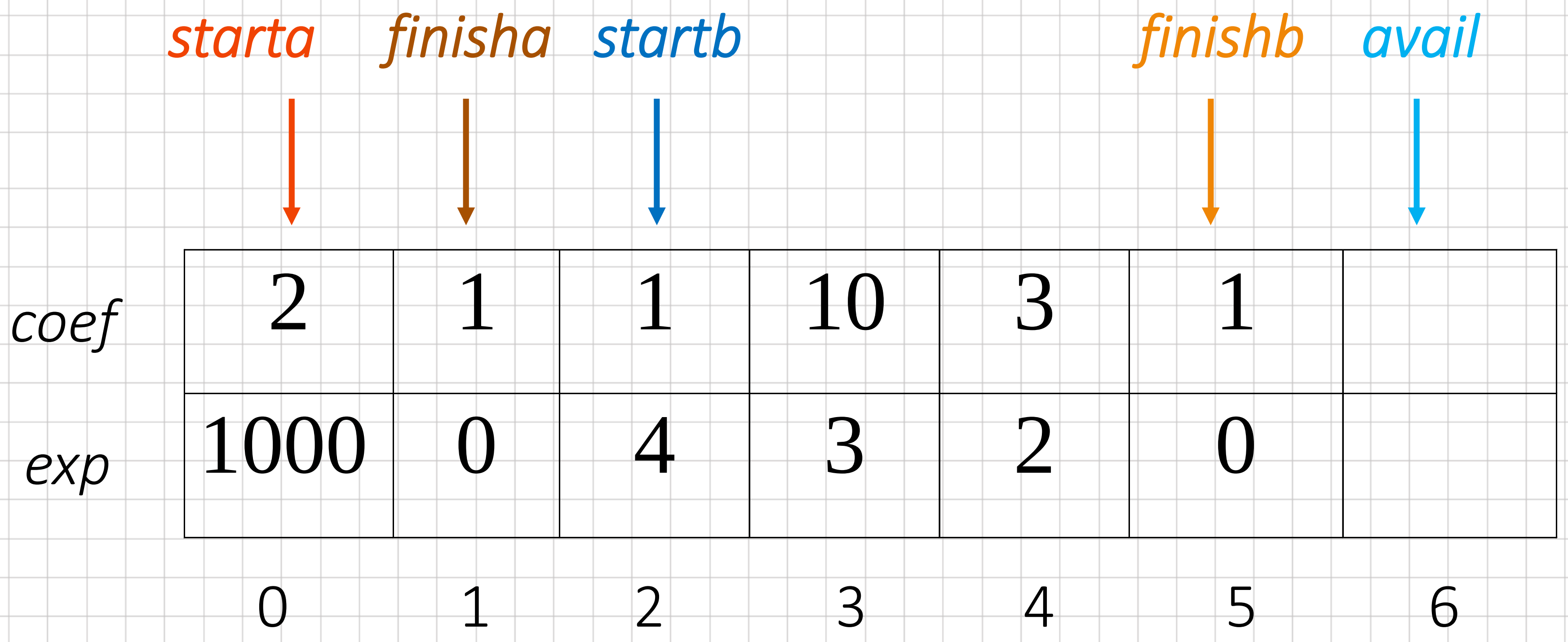
```
case 1: d =
          Attach(d, Coef (a, Lead_Exp(a)), Lead_Exp(a));
          a = Remove(a, Lead_Exp(a));
      }
  }
```

<u>insert any remaining terms of a or b into d</u>

advantage: easy implementation
disadvantage: waste space when sparse

*Program 2.5 :Initial version of *padd* function data structure 2:

use one global array to store all polynomials



	0	1	2	3	4	5	6
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	

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

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

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

*Figure 2.3: Array representation of two polynomials

storage requirements: start, finish, 2*(finish-start+1)

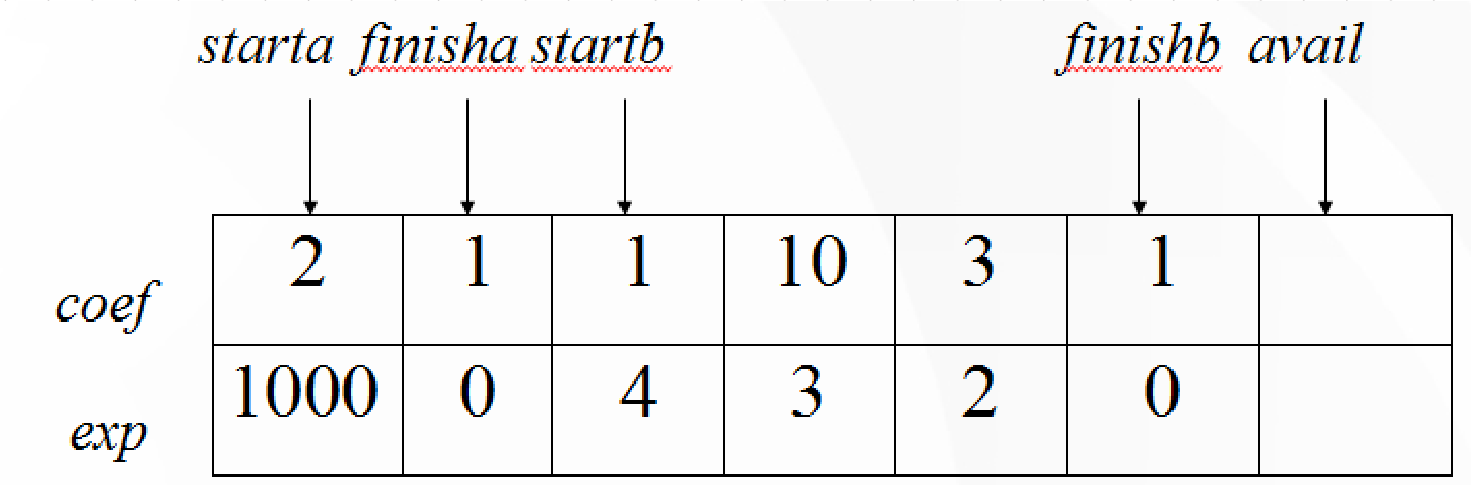
nonparse: twice as much as (1)

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;
```

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) is the number of nonzeros in A(B).

*Program 2.6: Function to add two polynomial

```
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 2.7:Function to add anew term

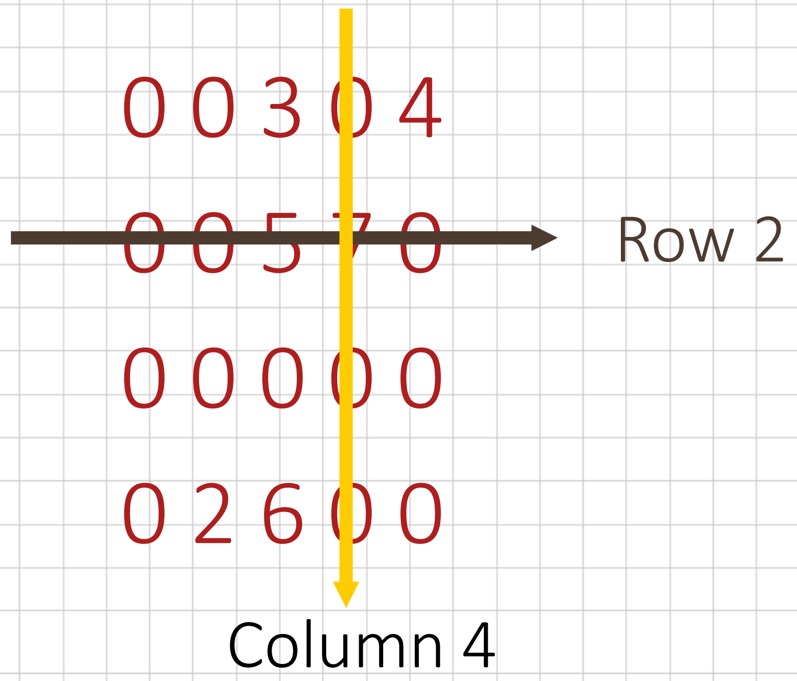
Problem: Compaction is required when polynomials that are no longer needed. (data movement takes time.)

Sparse Matrices

2.5; page 72 – 84

Sparse Matrices

Matrix ➔ table of values



$$\begin{array}{ccccc} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{array}$$

4 x 5 matrix

4 rows

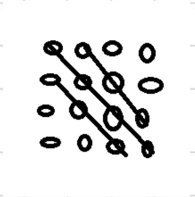
5 columns

20 elements

6 nonzero elements

Sparse Matrices

- Sparse matrix ➔ #nonzero elements/#elements is small.
- Examples:
  - Diagonal
    - Only elements along diagonal may be nonzero
    - n x n matrix → ratio is n/n^2 = 1/n
  - Tri-diagonal
    - Only elements on 3 central diagonals may be nonzero
    - Ratio is (3n-2)/n^2 = 3/n – 2/n^2



Sparse Matrices

- Lower triangular (?)
  - Only elements on or below diagonal may be nonzero
  - Ratio is $n(n+1)/(2n^2) \sim 0.5$

- These are structured sparse matrices. Nonzero elements are in a well-defined portion of the matrix.

Sparse Matrices

- An n x n matrix may be stored as an n x n array.
- This takes $O(n^2)$ space.
- The example structured sparse matrices may be mapped into a 1D array so that a mapping function can be used to locate an element quickly; the space required by the 1D array is less than that required by an n x n array.

Unstructured Sparse Matrices

- Airline flight matrix.

  airports are numbered 1 through n

  flight(i,j) = list of nonstop flights from airport i to airport j

  n = 1000 (say)

  n x n array of list pointers => 4 million bytes

  total number of nonempty flight lists = 20,000 (say)

  need at most 20,000 list pointers => at most 80,000 bytes

Unstructured Sparse Matrices

- Web page matrix.

  web pages are numbered 1 through n

  web(i,j) = number of links from page i to page j

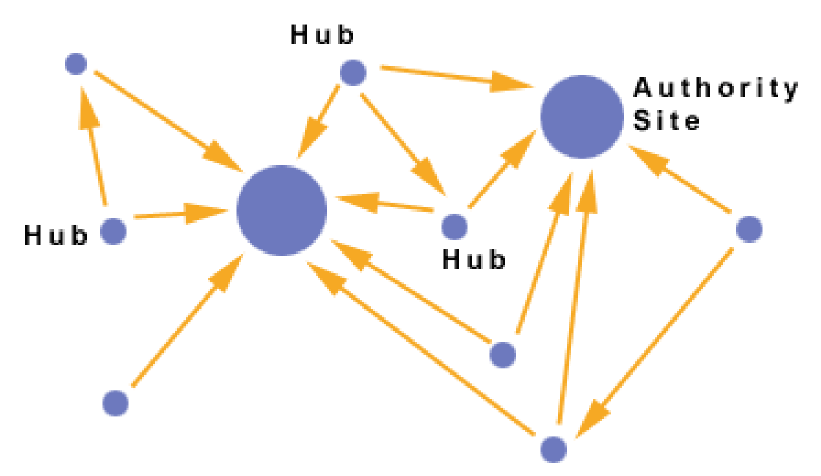
- Web analysis.

  authority page …

  page that has many links to it

  hub page …

  links to many authority pages



Sparse Matrix

$$\begin{array}{c|ccc} & \text{col 1} & \text{col 2} & \text{col 3} \\ \hline \text{row 1} & -27 & 3 & 4 \\ \text{row 2} & 6 & 82 & -2 \\ \text{row 3} & 109 & -64 & 11 \\ \text{row 4} & 12 & 8 & 9 \\ \text{row 5} & 48 & 27 & 47 \end{array}$$

5*3

$$\begin{array}{c|cccccc} & \text{col0} & \text{col1} & \text{col2} & \text{col3} & \text{col4} & \text{col5} \\ \hline \text{row0} & 15 & 0 & 0 & 22 & 0 & -15 \\ \text{row1} & 0 & 11 & 3 & 0 & 0 & 0 \\ \text{row2} & 0 & 0 & 0 & -6 & 0 & 0 \\ \text{row3} & 0 & 0 & 0 & 0 & 0 & 0 \\ \text{row4} & 91 & 0 & 0 & 0 & 0 & 0 \\ \text{row5} & 0 & 0 & 28 & 0 & 0 & 0 \end{array}$$

6*6

(a) 15/15

(b) 8/36

*Figure 2.4:Two matrices

sparse matrix
data structure?

SPARSE MATRIX ABSTRACT DATA TYPE

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 x max_col and whose maximum row size is max_row and whose maximum column size is max_col.

```
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.
```

* Structure 2.3: Abstract data type Sparse-Matrix

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 number of rows and columns and the number of nonzero elements

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

(2) Each element is characterized by <row, col, value>.

# of rows (columns)

# of nonzero terms

	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)

$\xrightarrow{\text{transpose}}$

	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 2.5:Sparse matrix and its transpose stored as triples

```
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 2.8: Transpose of a sparse matrix

Scan the array "columns" times.
The array has "elements" elements.

==> Time complexity
O(columns*elements)

Discussion: compared with 2-D array representation
for time complexity

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

elements --> columns * rows, when non-sparse
O(columns*columns*rows)

Problem: Scan the array “columns” times.

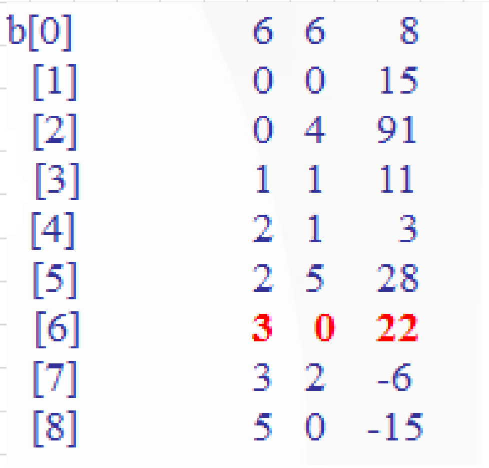
Solution: fastTranspose

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

After transpose

⇨



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

	[0]	[1]	[2]	[3]	[4]	[5]
row_terms =	2	1	2	$\boxed{2}$	0	1
starting_pos =	$\boxed{1}$	3	4	$\boxed{6} \rightarrow$	8	8

```
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*/
          for (i = 0; i < num_cols; i++)
columns          row_terms[i] = 0;
          for (i = 1; i  <= num_terms; i++)
elements         row_term [a[i].col]++
          starting_pos[0] = 1;
          for (i =1; i < num_cols; i++)
columns          starting_pos[i]=starting_pos[i-1] +row_terms [i-1];
```

```
          for (i=1; i <= num_terms, i++) {
elements                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 2.9:Fast transpose of a sparse matrix

Compared with 2-D array representation **for time complexity**

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

elements --> columns * rows when nonsparse

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.

Sparse Matrix Multiplication

Definition: $[D]_{m*p}=[A]_{m*n}*[B]_{n*p}$

Procedure: Fix a row of A and find all elements in column j of B for j=0, 1, ..., p-1.

Alternative 1. Scan all of B to find all elements in column j.

Alternative 2. Compute the transpose of B.

(Put all column elements consecutively)

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{array}{ccc} \left[\begin{matrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix}\right] \\ \begin{matrix} \text{Col 0} & \text{Col 1} & \text{Col 2} \end{matrix} \end{array} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Sparse Matrix Multiplication

Alternative 1. Scan all of B to find all elements in j.

Alternative 2. Compute the transpose of B.

(Put all column elements consecutively)

D = A * B

$$\begin{array}{c} \boxed{15\quad 0\quad -1} \\ 0\quad 3\quad 5 \end{array} \qquad \boxed{\begin{array}{c} 0 \\ 5 \\ 3 \end{array}} \begin{array}{ccc} 0 & 9 & 0 \\ -1 & 0 & 4 \\ 0 & 1 & 5 \end{array}$$

a[0]	2	3	4
a[1]	0	0	15
a[2]	0	2	-1
a[3]	1	1	3
a[4]	1	2	5

b[0]	3	4	7
b[1]	0	2	9
b[2]	1	0	5
b[3]	1	1	-1
b[4]	1	3	4
b[5]	2	0	3
b[6]	2	2	1
b[7]	2	3	5

transpose

b[0]	3	4	7
b[1]	0	1	5
b[2]	0	2	3
b[3]	1	1	-1
b[4]	2	0	9
b[5]	2	2	1
b[6]	3	1	4
b[7]	3	2	5

```
void mmult (term a[ ], term b[ ], term d[ ] )
/* multiply two sparse matrices */
{
  int i, j, column, totalb = b[0].value, totald = 0;
  int rows_a = a[0].row, cols_a = a[0].col,
  totala = a[0].value; int cols_b = b[0].col,
  int row_begin = 1, row = a[1].row, sum =0;
  int new_b[MAX_TERMS];
  if (cols_a != b[0].row){
       fprintf (stderr, “Incompatible matrices\n”);
       exit (1);
}
```

```
fast_transpose(b, new_b);                cols_b + totalb
/* set boundary condition */

a[totala+1].row = rows_a;
new_b[totalb+1].row = cols_b;
new_b[totalb+1].col = 0;

for (i = 1; i <= totala; ) {    at most rows_a times
   column = new_b[1].row;
   for (j = 1; j <= totalb+1;) {
   /* mutiply row of a by column of b */
   if (a[i].row != row)  {
      storesum(d, &totald, row, column, &sum);
      i = row_begin;
      for (; new_b[j].row == column; j++)
        ;
      column =new_b[j].row
   }
else switch (COMPARE (a[i].col, new_b[j].col)) {
            case -1: /* go to next term in a */
                       i++; break;
            case 0: /* add terms, go to next term in a and b */
                       sum += (a[i++].value * new_b[j++].value);
                       break;
             case 1: /* advance to next term in b*/
                        j++
          }
      } /* end of for j <= totalb+1 */
      for (; a[i].row == row; i++)
           ;
       row_begin = i; row = a[i].row;
    } /* end of for i <=totala */
    d[0].row = rows_a;
    d[0].col = cols_b; d[0].value = totald;
}
```

*Program 2.10: Sparse matrix multiplication

```
void storesum(term d[ ], int *totald, int row, int column,
                                                  int *sum)
{
/* if *sum != 0, then it along with its row and column
   position is stored as the *totald+1 entry in d */
   if (*sum)
      if (*totald < MAX_TERMS) {
         d[++*totald].row = row;
         d[*totald].col = column;
         d[*totald].value = *sum;
      }
      else {
         fprintf(stderr, "Numbers of terms in product
                                        exceed %d\n", MAX_TERMS);
  exit(1);
  }
}
```

Program 2.11: storSum function

Analyzing the algorithm

cols_b * termsrow$_1$ + totalb +

cols_b * termsrow$_2$ + totalb +

… +

cols_b * termsrow$_p$ + totalb

= cols_b * (termsrow$_1$ + termsrow$_2$ + … + termsrow$_p$) +
    rows_a * totalb

= cols_b * totala + row_a * totalb

O(cols_b * totala + rows_a * totalb)

Compared with matrix multiplication using array

```
for (i =0; i < rows_a; i++)
   for (j=0; j < cols_b; j++) {
      sum =0;
      for (k=0; k < cols_a; k++)
          sum += (a[i][k] *b[k][j]);
      d[i][j] =sum;
  }
```

O(rows_a * cols_a * cols_b) vs.
O(cols_b * total_a + rows_a * total_b)

optimal case: total_a < rows_a * cols_a
total_b < cols_a * cols_b
worse case: total_a --> rows_a * cols_a, or
total_b --> cols_a * cols_b

String

2.7; page 87 - 97

String

Usually string is represented as a character array.

General string operations include comparison, string concatenation, copy, insertion, string matching, printing, etc.

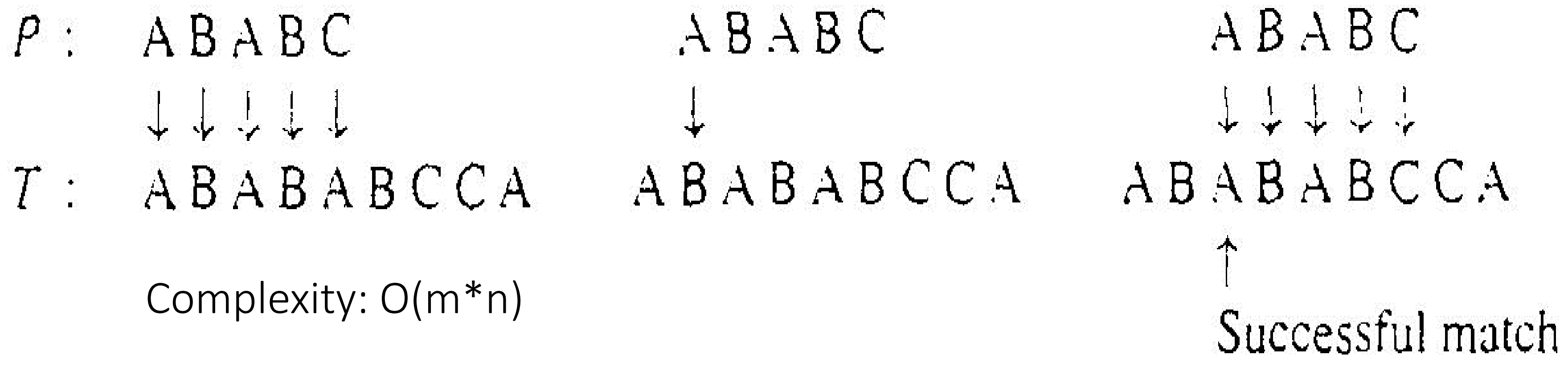
H	e	l	l	o		W	o	r	l	d	\0

String Matching: Straightforward solution

- Algorithm: Simple string matching e.g., String. indexAt
- Input: P and T, the pattern and text strings; m, the length of P and n, the length of T

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.



Complexity: O(m*n)

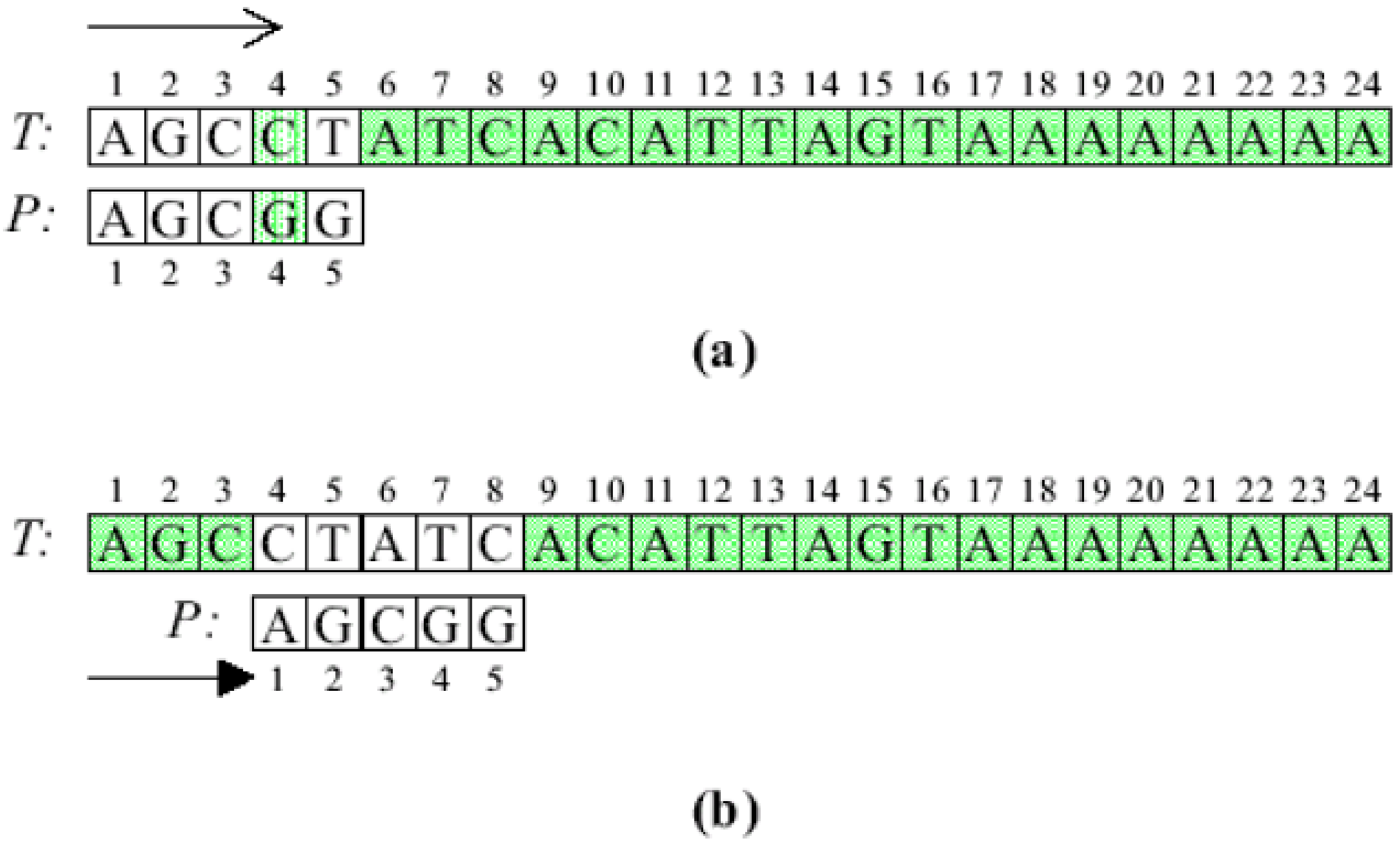
Two Phases of KMP

Knuth, Morris, Pratt pattern matching algorithm

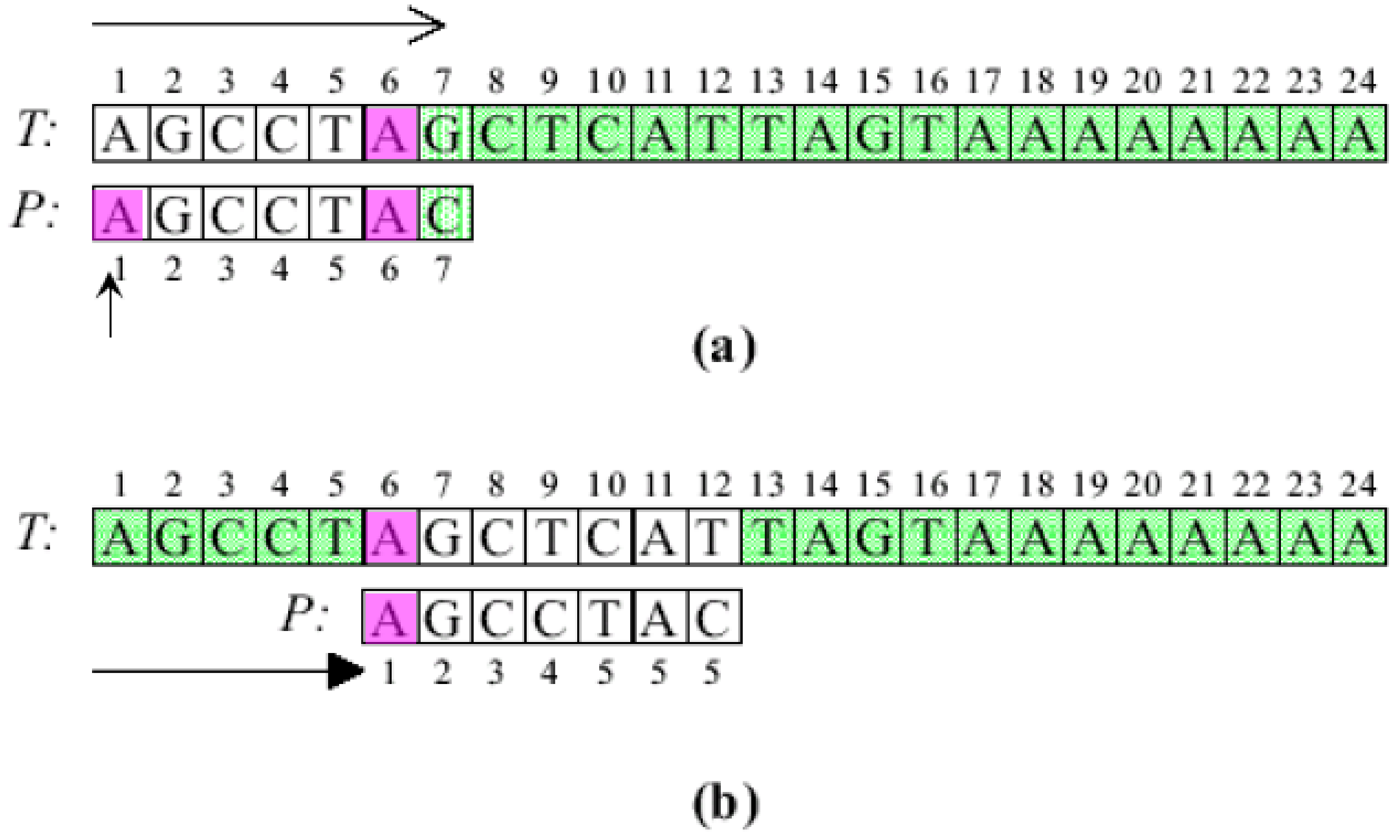
Phase 1：generate an array to indicate the moving direction.

Phase 2：make use of the array to move and match string

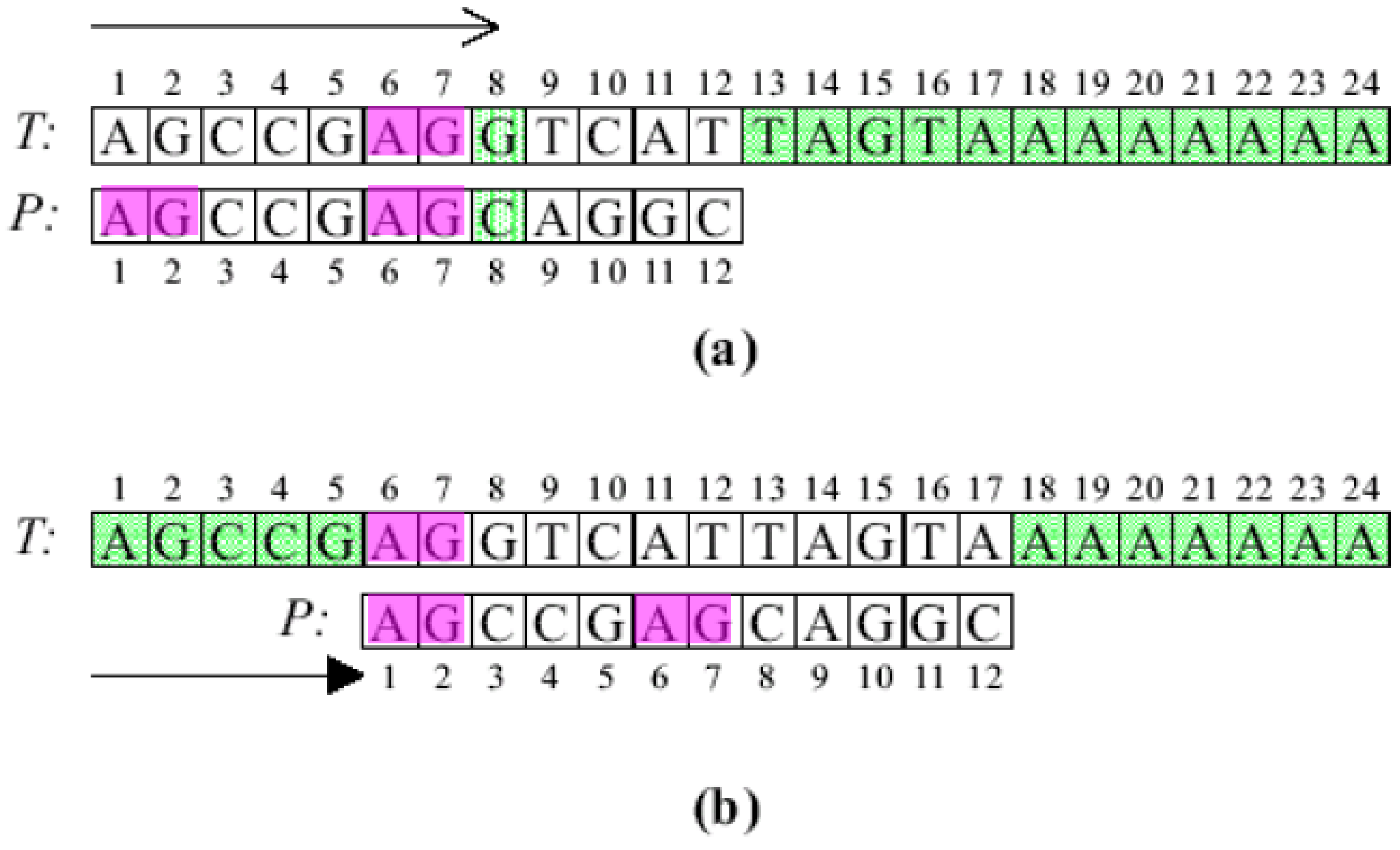
The first Case for the KMP Algorithm



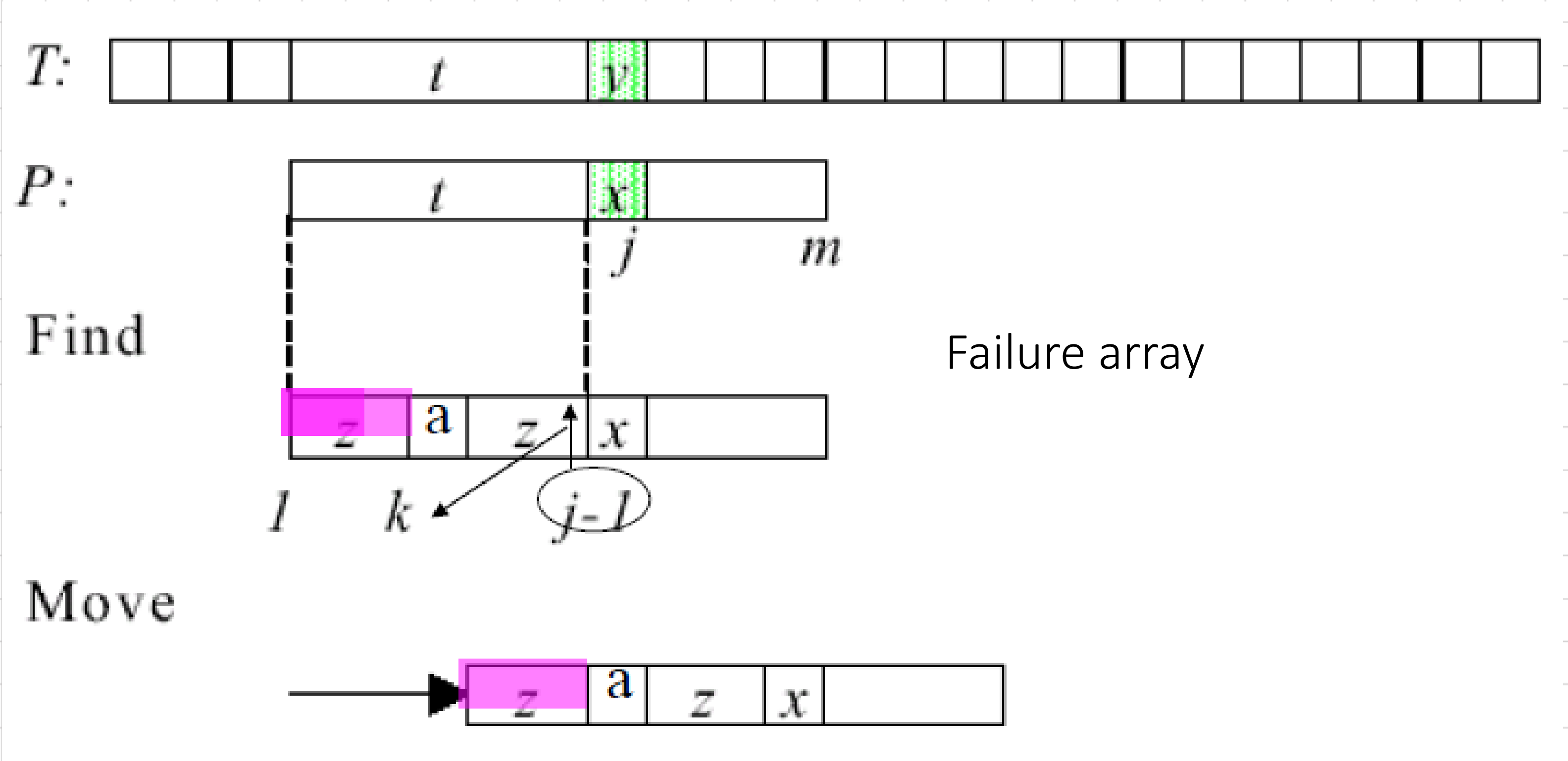
The Second Case for the KMP Algorithm



The Third Case for the KMP Algorithm



The KMP Alogrithm



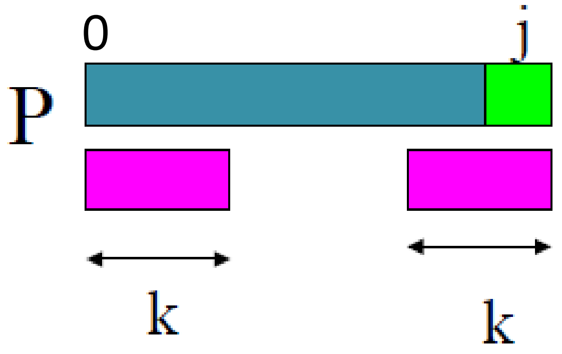
Failure array

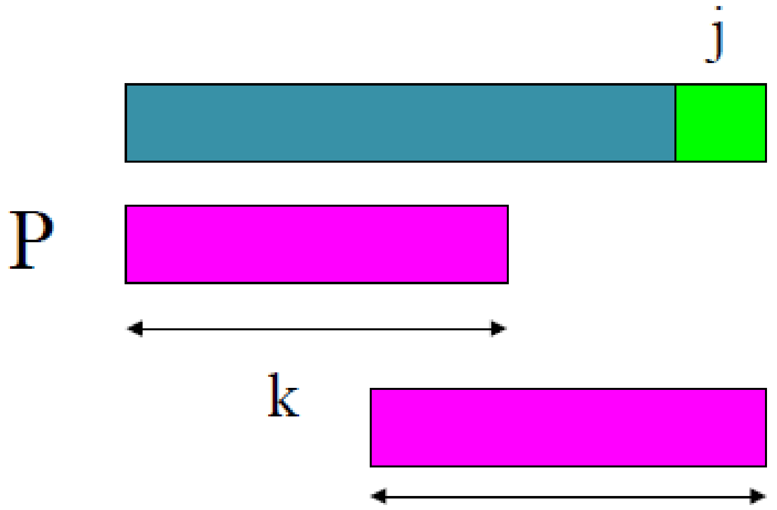
String Matching The Knuth-Morris-Pratt Algorithm

- Definition: If $P = p_0\, p_1\, p_2\, p_3 \ldots p_{n-1}$ is a pattern, then its failure function, f, is defined as

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

Failure Function





The prefix function, Π

Following pseudocode computes the prefix fucnction, Π:

Compute-Prefix-Function (p)

```
1     m ← length[p]          //'p' pattern to be matched
2     Π[1] ← 0
3     k ← 0
4     for q ← 2 to m
5         do while k > 0 and p[k+1] != p[q]
6                do k ← Π[k]
7         If p[k+1] = p[q]
8            then k ← k +1
9         Π[q] ← k
10    return Π
```

Example: compute Π for the pattern ‘p’ below:

p							
	a	b	a	b	a	c	a

Initially: m = length[p] = 7

Π[1] = 0

k = 0

Step 1: q = 2, k=0

Π[2] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0					

Step 2: q = 3, k = 0,

Π[3] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1				

Step 3: q = 4, k = 1

Π[4] = 2

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2			

Step 4: $q = 5, k = 2$

$\Pi[5] = 3$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3		

Step 5: $q = 6, k = 0$

$\Pi[6] = 0$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	

Step 6: $q = 7, k = 0$

$\Pi[7] = 1$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	1

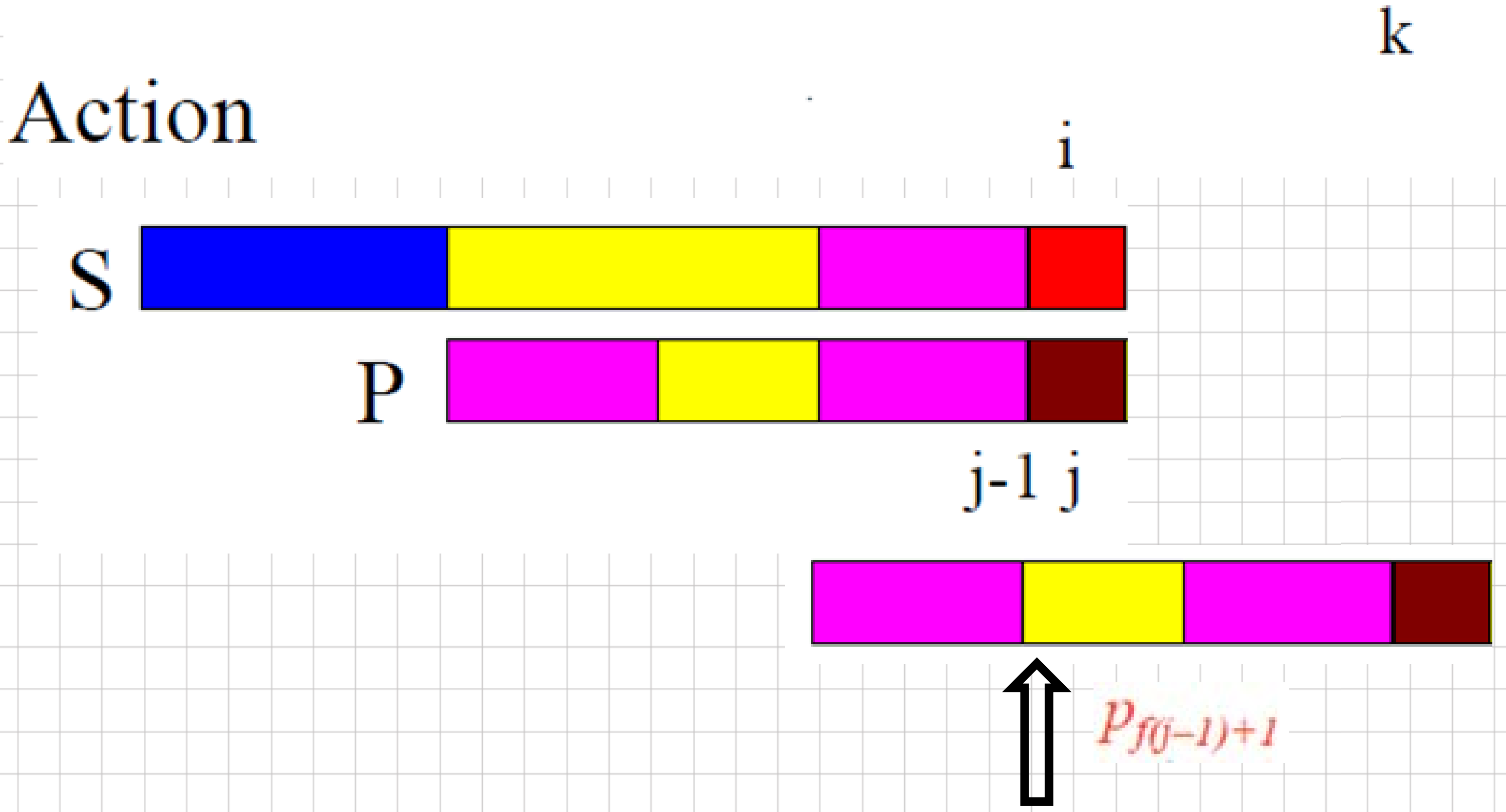
After iterating 6 times, the prefix function computation is complete: →

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	1

The KMP Alogrithm (cont'd)

- 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 $j = 0$, we may continue by comparing s_{i+1} *and* p_0.

Action



The KMP Matcher

```
int pmatch(char *string, char *pat)
{  /* Knuth, Morris, Pratt的字串樣式比對演算法 */
    int i = 0, j = 0;
    int lens = strlen(string);
    int lenp = strlen(pat);
    while ( i < lens && j < lenp ) {
          if (string[i] == pat[j]) {
              i++;
              j++;
          } else if (j == 0)
              i++;
          else
              j = failure[j-1]+1;
    }
    return ( (j == lenp) ? (i-lenp) : -1);
}
```

Illustration: given a String 'S' and pattern 'p' as follows:

S	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

p	a	b	a	b	a	c	a

Let us execute the KMP algorithm to find whether 'p' occurs in 'S'.

For 'p' the prefix function, Π was computed previously and is as follows:

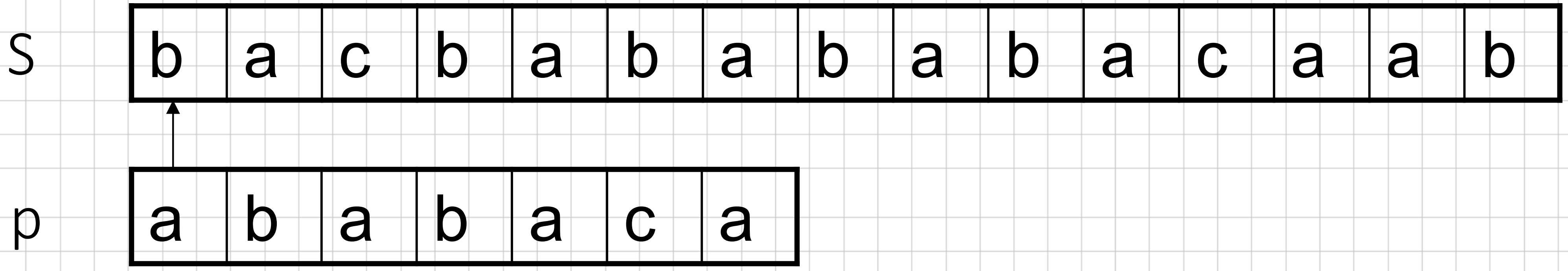
q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	1

Failure function

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	1

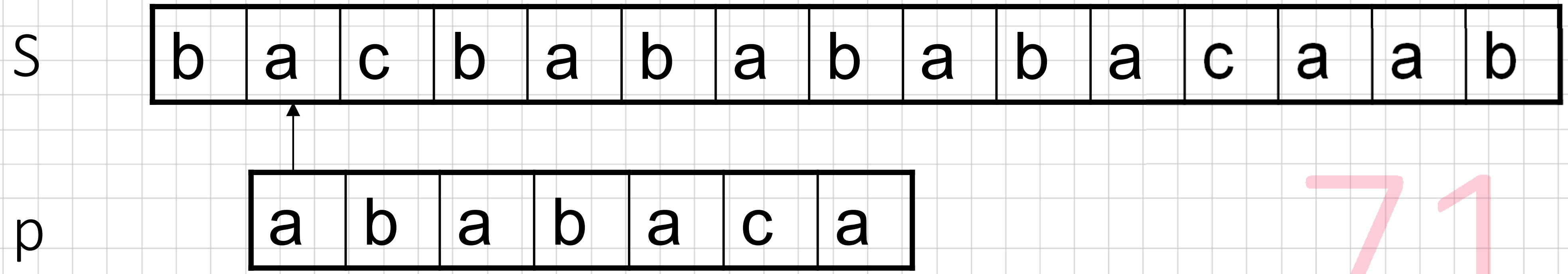
Initially: n = size of S = 15;
m = size of p = 7

Step 1: i = 1
comparing p[1] with S[1]



P[1] does not match with S[1]. ‘p’ will be shifted one position to the right.

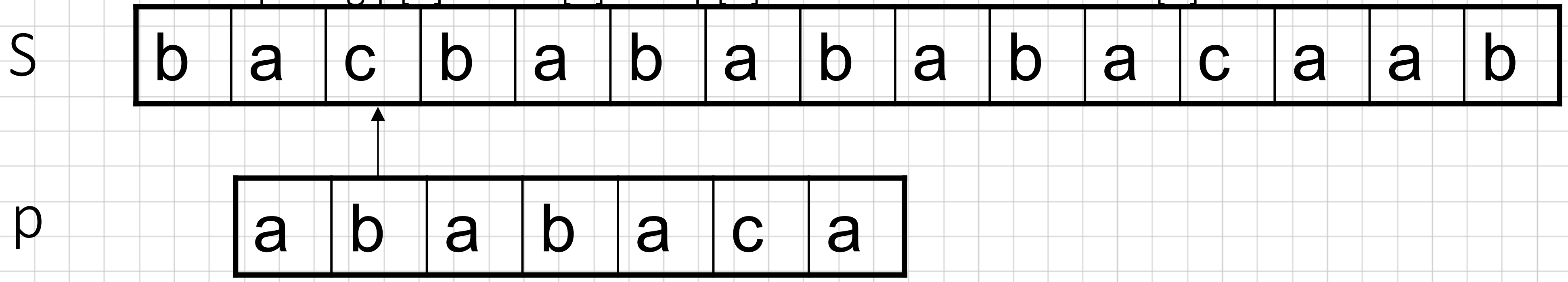
Step 2: i = 2
comparing p[1] with S[2]



P[1] matches S[2]. Since there is a match, p is not shifted.

Step 3: i = 3

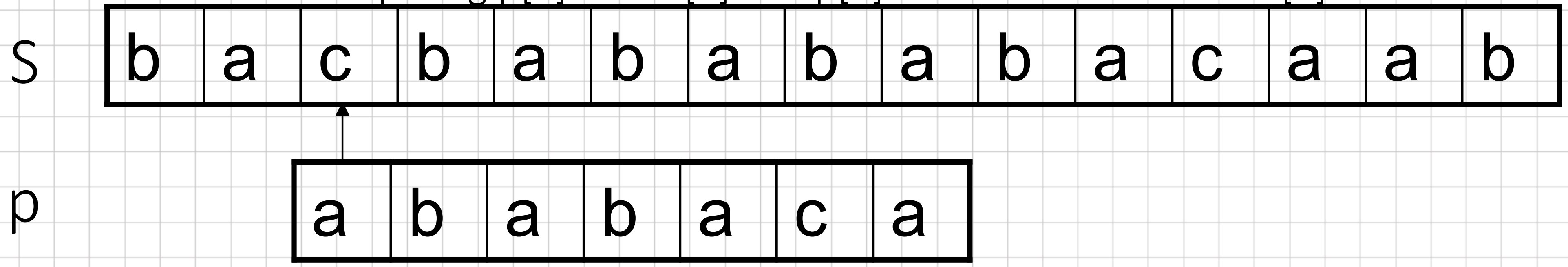
Comparing p[2] with S[3]   p[2] does not match with S[3]



Backtracking on p, comparing p[1] and S[3] ($\Pi[1] + 1 = 0 + 1 = 1$)

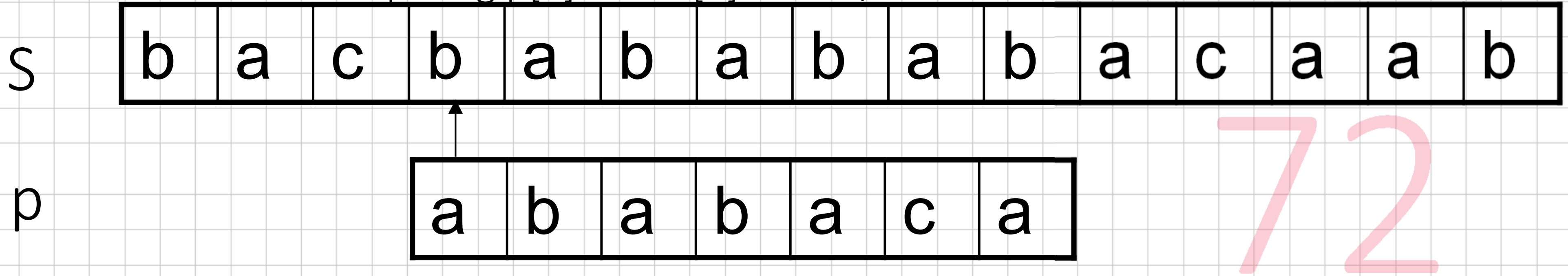
Step 4: i = 3

comparing p[1] with S[3]   p[1] does not match with S[3]

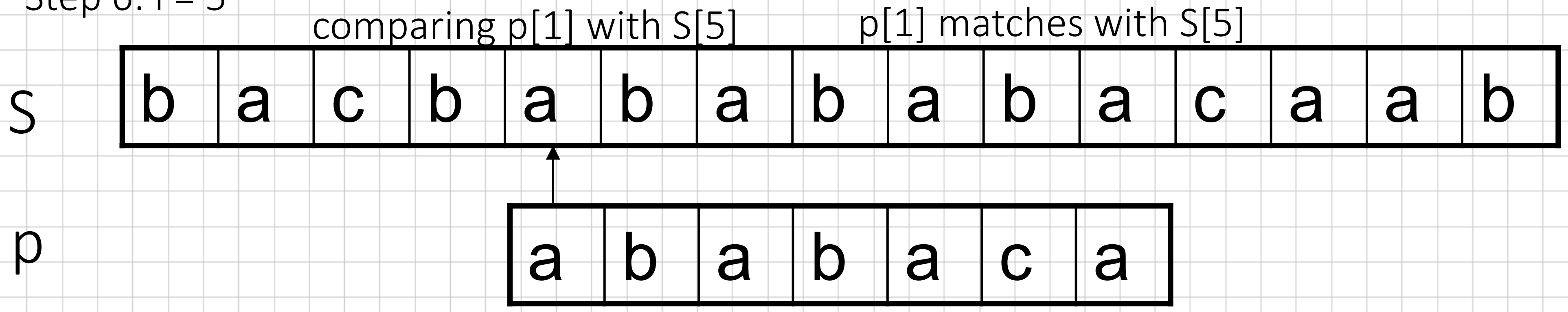


Step 5: i = 4

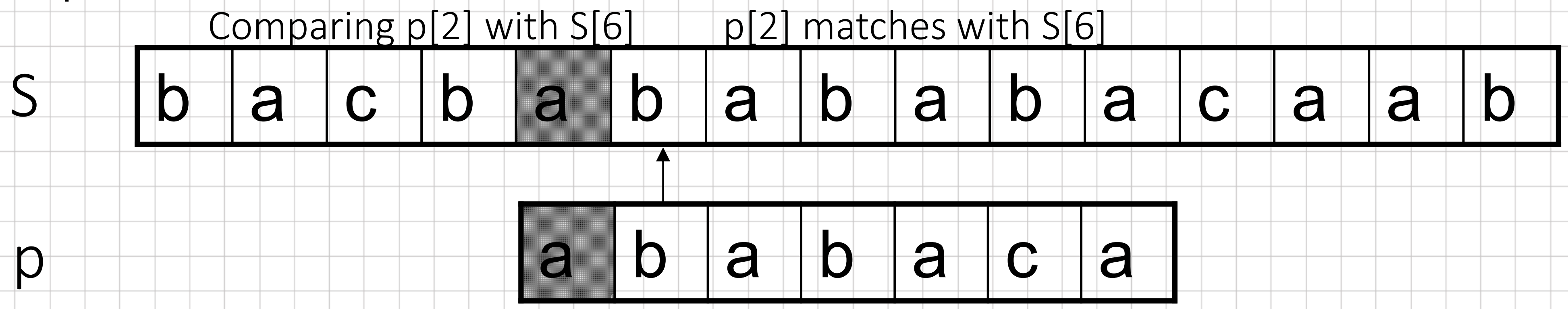
comparing p[1] with S[4]   p[1] does not match with S[4]



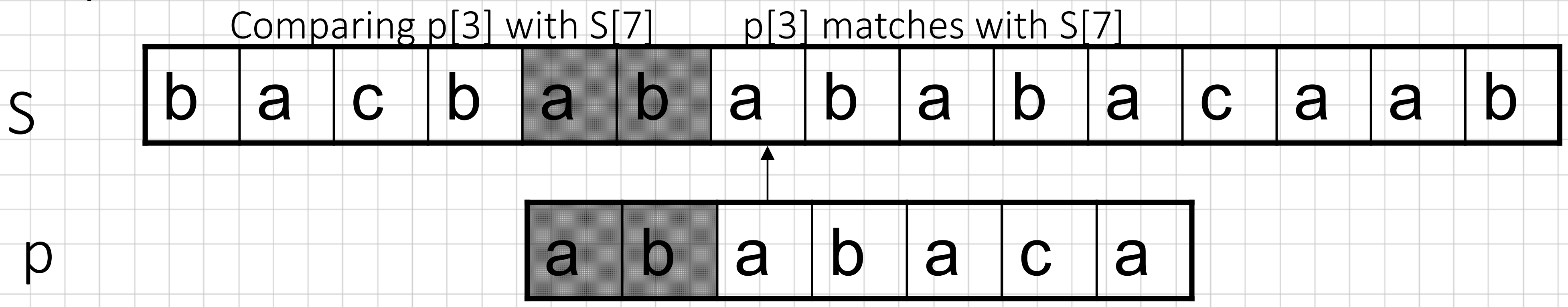
Step 6: i = 5



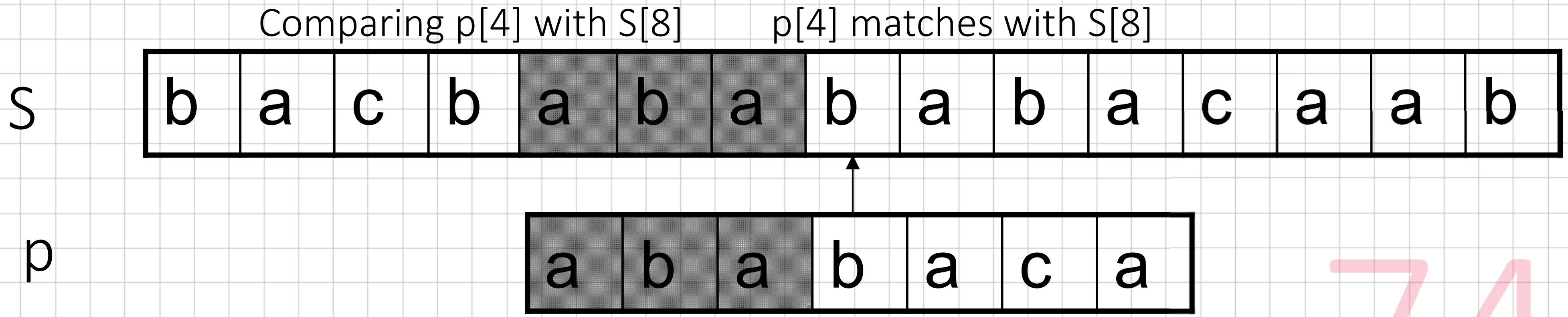
Step 7: i = 6



Step 8: i = 7

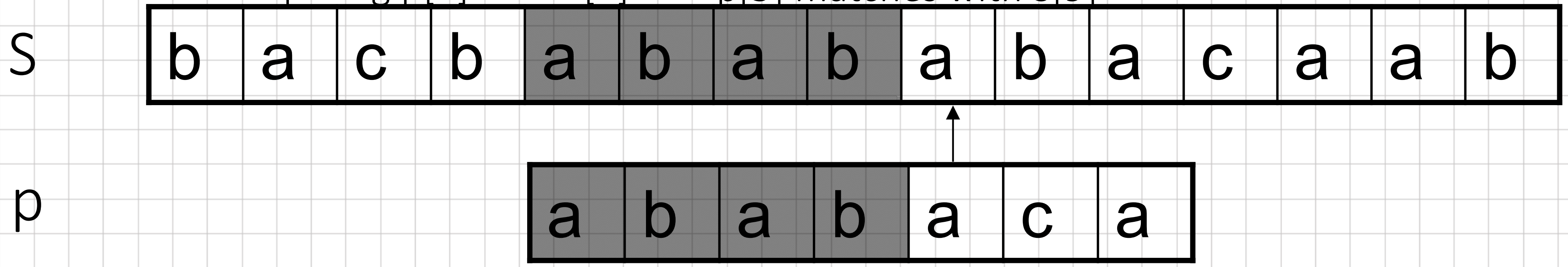


Step 9: i = 8



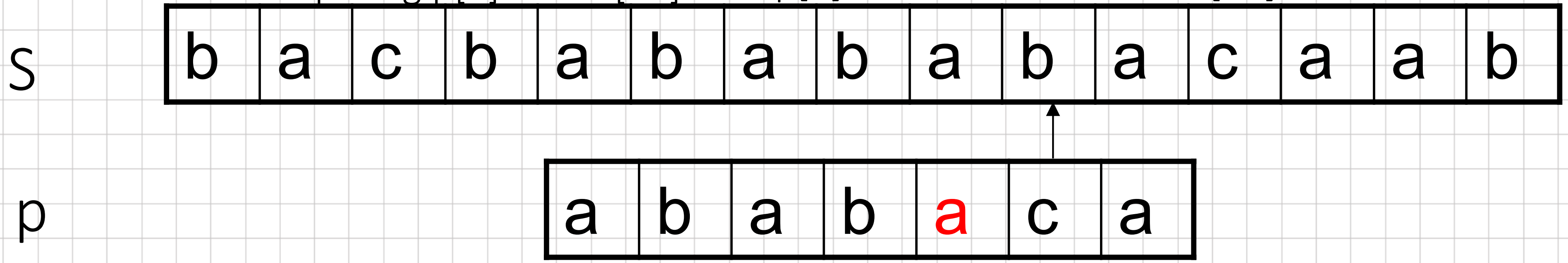
Step 10: i = 9

Comparing p[5] with S[9] p[5] matches with S[9]



Step 11: i = 10

Comparing p[6] with S[10] p[6] doesn't match with S[10]

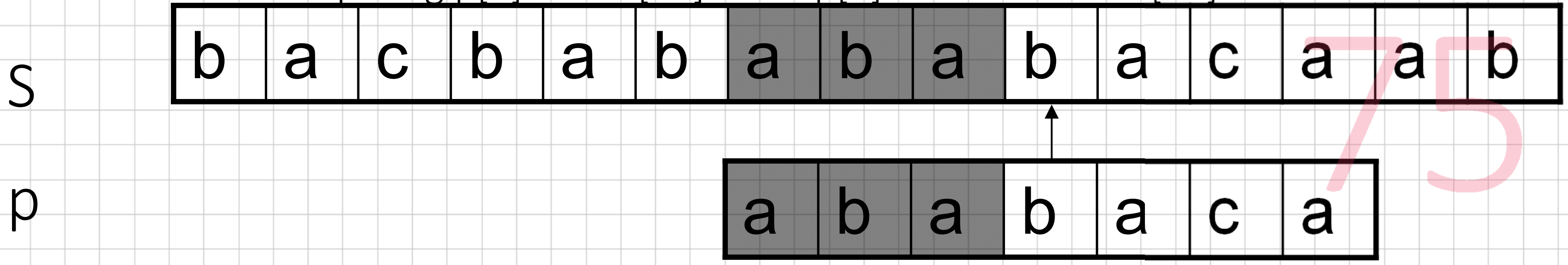


Backtracking on p, comparing p[4] with S[10] because after mismatch q = Π[5] = 3

(Π[5] + 1 = 3 + 1 = 4

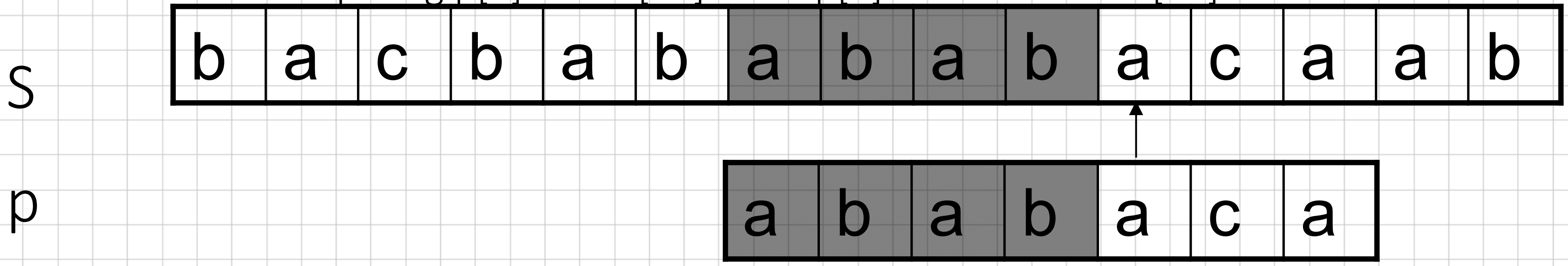
Step 12: i = 11

Comparing p[4] with S[10] p[4] matches with S[10]



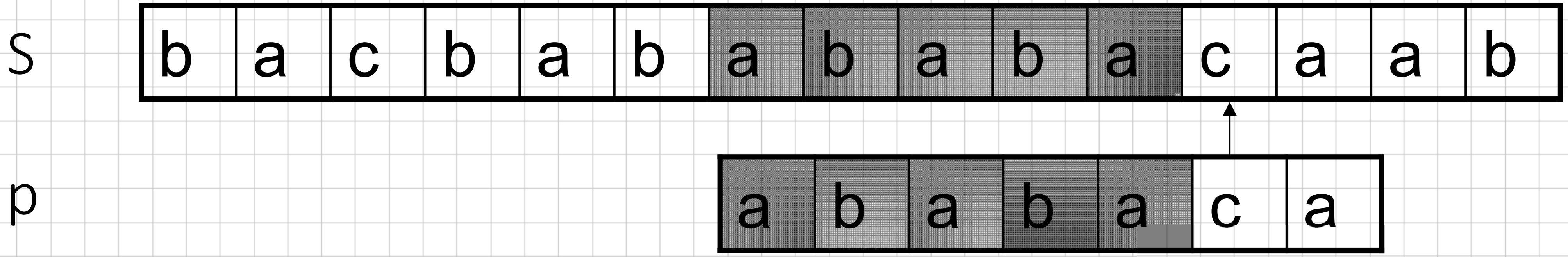
Step 13: i = 12

Comparing p[5] with S[11] p[5] matches with S[11]

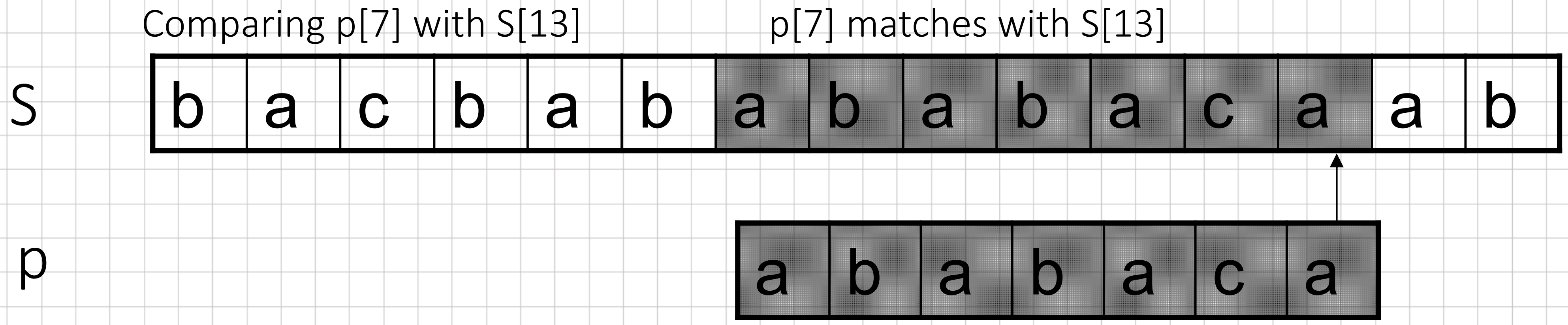


Step 14: i = 13

Comparing p[6] with S[12] p[6] matches with S[12]



Step 15: i = 13



Pattern ‘p’ has been found to completely occur in string ‘S’.

The analysis of the K.M.P. Algorithm

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