HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

Nguyễn Đắc Tiến

BÁO CÁO
MỘT SỐ THUẬT TOÁN TÌM KIẾM MẪU

LỚP : HỆ THỐNG THÔN TIN - M15CQIS02-B
MÔN : THUẬT TOÁN NÂNG CAO

NGƯỜI HƯỚNG DẪN : TIẾN SĨ NGUYỄN DUY PHƯƠNG

HÀ NỘI, THÁNG 3 NĂM 2016

MỤC LỤC

I. GIỚI THIỆU VẤN ĐỀ

1. Đặt vấn đề

Đối sánh xâu (String matching) là một chủ đề quan trọng trong lĩnh vực xử lý văn bản. Các thuật toán đối sánh xâu được xem là những thành phần cơ sở được cài đặt cho các hệ thống thực tế đang tồn tại trong hầu hết các hệ điều hành. Hơn thế nữa, các thuật toán đối sánh xâu cung cấp các mô hình cho nhiều lĩnh vực khác nhau của khoa học máy tính: xử lý ảnh, xử lý ngôn ngữ tự nhiên, tin sinh học và thiết kế phần mềm.

String-matching được hiểu là việc tìm một hoặc nhiều xâu mẫu (pattern) xuất hiện trong một văn bản (có thể là rất dài). Ký hiệu xâu mẫu hay xâu cần tìm là $X =(x_0, x_1,..,x_{m-1})$ có độ dài m. Văn bản $Y =(y_0, y_1,..,y_{n-1})$ có độ dài n. Cả hai xâu được xây dựng từ một tập hữu hạn các ký tự Alphabet ký hiệu là $\sum$ với kích cỡ là σ. Như vậy một xâu nhị phân có độ dài n ứng dụng trong mật mã học cũng được xem là một mẫu. Một chuỗi các ký tự ABD độ dài m biểu diễn các chuỗi AND cũng là một mẫu.

Input:

- Xâu mẫu $X =(x_0, x_1,.., x_m-1)$, độ dài m.
- Văn bản $Y =(y_0, x_1,.., y_{n-1})$, độ dài n.

Output:

- Tất cả vị trí xuất hiện của X trong Y

2. Phân loại các thuật toán đối sánh mẫu

Thuật toán đối sánh mẫu đầu tiên được đề xuất là Brute-Force. Thuật toán xác định vị trí xuất hiện của X trong Y với thời gian O(m.n). Nhiều cải tiến khác nhau của thuật toán Brute-Force đã được đề xuất nhằm cải thiện tốc độ tìm kiếm mẫu. Ta có thể phân loại các thuật toán tìm kiếm mẫu thành các lớp :

- **Tìm kiếm mẫu từ bên trái qua bên phải**: Harrison Algorithm, Karp-Rabin Algorithm, Morris-Pratt Algorithm, Knuth- Morris-Pratt Algorithm, Forward Dawg Matching algorithm , Apostolico-Crochemore algorithm, Naive algorithm.
- **Tìm kiếm mẫu từ bên phải qua bên trái:** Boyer-Moore Algorithm , Turbo BM Algorithm, Colussi Algorithm, Sunday Algorithm, Reverse Factorand Algorithm, Turbo Reverse Factor, Zhu and Takaoka and Berry-Ravindran Algorithms.
- **Tìm kiếm mẫu từ một vị trí cụ thể:** Two Way Algorithm, Colussi Algorithm , Galil-Giancarlo Algorithm, Sunday's Optimal Mismatch

Algorithm, Maximal Shift Algorithm, Skip Search, KMP Skip Search and Alpha Skip Search Algorithms.

- **Tìm kiếm mẫu từ bất kỳ:** Horspool Algorithm, Boyer-Moore Algorithm, Smith Algorithm , Raita Algorithm

3. Một số khái niệm và định nghĩa cơ bản về tìm kiếm mẫu:

Giả sử Alphabet là tập hợp (hoặc tập con) các mã ASSCI. Một từ w =(w_0, w_1,..,w_l) có độ dài l, w_l=null giống như biểu diễn của ngôn ngữ C. Khi đó ta định nghĩa một số thuật ngữ sau:

- *Prefix (tiền tố).* Từ u được gọi là tiền tố của từ w nếu tồn tại một từ v để w =uv (v có thể là rỗng). Ví dụ: u =“**AB**” là tiền tố của w =“**AB**CDEF” và u =“**com**” là tiền tố của w =“**com**munication”.
- *Suffix (hậu tố)*. Từ v được gọi là hậu tố của từ w nếu tồn tại một từ u để w =uv (u có thể là rỗng). Ví dụ: v =“**EF**” là hậu tố của w =“ABCD**EF**” và v =“**tion**” là hậu tố của w =“communica**tion**”.
- *Factor (substring, subword).* Một từ z được gọi là một xâu con, từ con hay nhân tố của từ w nếu tồn tại hai từ u, v (u, v có thể rỗng) sao cho w = uzv. Ví dụ từ z =“**CD**” là factor của từ w =“AB**CD**EF” và z =“**muni**” là factor của w =“com**muni**cation”.
- *Period (đoạn).* Một số tự nhiên p được gọi là đoạn của từ w nếu với mọi i (0<=i<m-1) thì w[i] = w[i+p]. Giá trị đoạn nhỏ nhất của w được gọi là đoạn của w ký hiệu là pre(w). Ví dụ w =“ABABCDEF”, khi đó tồn tại p=2.
- *Periodic (tuần hoàn).* Từ w được gọi là tuần hoàn nếu đoạn của từ nhỏ hơn hoặc bằng l/2. Trường hợp ngược lại được gọi là không tuần hoàn. Ví dụ từ w=“ABAB” là từ tuần hoàn. Từ w =“ABABCDEF” là không tuần hoàn.
- *Basic word (từ cơ sở).* Từ w được gọi là từ cơ sở nếu nó không thể viết như lũy thừa của một từ khác. Không tồn tại z và k để zk= w .
- *Boder word (từ biên).* Từ z được gọi là boder của w nếu tồn tại hai từ u, v sao cho w = uz =zv . Khi đó z vừa là tiền tố vừa là hậu tố của w . Trong tình huống này |u| = |v| là một đoạn của w .
- *Reverse word (Từ đảo).* Từ đảo của từ w có độ dài l ký hiệu là wR=(w_{r-1} ,w_{r-2} ,…, w_1,w_0).
- Deterministic Finite Automata (DFA). Một automat hữu hạn A là bộ bốn (Q, q0, T, E) trong đó:
  - ✓ *Q là tập hữu hạn các trạng thái.*
  - ✓ *q0 là trạng thái khởi đầu.*
  - ✓ *T là tập con của Q là tập trạng thái dừng.*

✓ *E là tập con của (Q, ∑,T) tập các chuyển dịch.*

Ngôn ngữ L(A) đoán nhận bởi A được định nghĩa :

$\{w \in \sum^* : \exists q_0, q_1, ...q_n, n=|w|, q_n \in T; \forall\ 0 \le i \le n; (q_i, w[i], q_{i+1}) \in \sigma\ \}$

II. MỘT SỐ THUẬT TOÁN TÌM KIẾM MẪU

1. Thuật toán tìm kiếm từ phải sang trái

1.1. Thuật toán Brute-Force

a.Phát biểu thuật toán

Thuật toán Brute-Force để tìm sự xuất hiện của một chuỗi (được gọi là mẫu) trong một văn bản bằng cách kiểm tra từng vị trí trong văn bản ở đó mẫu có thể khớp được, cho đến khi chúng khớp nhau thực sự.

Thuật toán Brute Force không cần giai đoạn tiền xử lý cũng như các mảng phụ cho quá trình tìm kiếm. Độ phức tạp tính toán của thuật toán này là O(N*M).

b.Mã hóa thuật toán

Thuật toán Brute-Force:

Input :

- Xâu mẫu X =(x_0, x_1,..,x_m), độ dài m.
- Văn bản nguồn Y=(y_1, y_2,..,y_n) độ dài n.

Output:

- Mọi vị trí xuất hiện của X trong Y.

Formats: Brute-Force(X, m, Y, n);

Actions:

```
for ( j = 0; j <= (n-m); j++) { //duyệt từ trái qua phải xâu X
        for (i =0; i<m && X[i] == Y[i+j]; i++) ; //Kiểm tra mẫu
            if (i>=m) OUTPUT (j);
}
```

EndActions.

c.Kiểm nghiệm

X = 10100111 (m = 8)

Y = 10011101001010001010011 10 (n = 25)

j	i	X[i] == Y[i+j]?	OUTPUT (j)

0	0 1 2	1 == 1 (Y) 0 == 0 (Y) 1 == 0 (N)	
1	0	1 == 0 (N)	
2	0	1 == 0 (N)	
3	0 1	1 == 1 (Y) 0 == 1 (N)	
4	0 1	1 == 1 (Y) 0 == 1 (N)	
5	0 1 2 3 4 5 6	1 == 1 (Y) 0 == 0 (Y) 1 == 1 (Y) 0 == 0 (Y) 0 == 0 (Y) 1 == 1 (Y) 1 == 0 (N)	
6	0	1 == 0 (N)	
7	0 1 2	1 == 1 (Y) 0 == 0 (Y) 1 == 0 (N)	
8	0	1 == 0 (N)	
9	0	1 == 0 (N)	
10	0 1 2 3 4 5	1 == 1 (Y) 0 == 0 (Y) 1 == 1 (Y) 0 == 0 (Y) 0 == 0 (Y) 1 == 0 (N)	
11	0	1 == 0 (N)	
12	0	1 == 1 (Y) 0 == 0 (Y)	

		1 == 0 (N)	
13	0	1 == 0 (N)	
14	0	1 == 0 (N)	
15	0	1 == 0 (N)	
16	0 1 2 3 4 5 6 7	1 == 1 (Y) 0 == 0 (Y) 1 == 1 (Y) 0 == 0 (Y) 0 == 0 (Y) 1 == 1 (Y) 1 == 1 (Y) 1 == 1 (Y)	OUTPUT(16)
17	0	1 == 0 (N)	

1.2. Thuật toán Karp-Rabin

a.Đặc điểm

- Sử dụng 1 hàm băm để tìm chuỗi con
- Độ phức tạp thời gian và không gian tiền xử lý *O(m)*
- Độ phức tạp thời gian và không gian xử lý tìm kiếm *O(m+n)*

b.Hàm băm cơ bản

Hàm băm là giải thuật nhằm sinh ra các giá trị băm tương ứng với mỗi khối dữ liệu, một chuỗi kí tự, một đối tượng trong lập trình hướng đối tượng,.. Giá trị băm đóng vai gần như một khóa để phân biệt các khối dữ liệu, tuy nhiên, người ta chấp hiện tượng trùng khóa hay còn gọi là đụng độ và cố gắng cải thiện giải thuật để giảm thiểu sự đụng độ đó. Hàm băm thường được dùng trong bảng băm nhằm giảm chi phí tính toán khi tìm một khối dữ liệu trong một tập hợp, nhờ việc so sánh các giá trị băm nhanh hơn việc so sánh những khối dữ liệu có kích thước lớn.

Một hàm băm đơn giản nhất đó là tính toán giá trị băm dựa trên mã ASCII hoặc UNICODE của từng ký tự. Ví dụ với chuỗi nguồn là "abcdefgh" và chuỗi cần tìm có chiều dài là 4 thì giá trị băm đầu tiên như sau:

h1 = a + b + c + d

= 65+66+67+68

= 266

Giá trị băm tiếp theo cần tính là:

h2 = b + c + d + e

= h1 – a + e

= 266 – 65 + 69

c.Thuật toán

Thuật toán Rabin-Karp sử dụng hàm băm để so sánh giá trị băm của các chuỗi trước khi thực sự so sánh chuỗi. Phương pháp này giúp tiết kiệm được thời gian so sánh, đặc biệt với các chuỗi tìm kiếm dài.

Input:

- T[0 .. n-1] : là văn bản có n ký tự
- P[0 .. m -1]: là pattern có m ký tự với $m \leq n$
- ts : là giá trị băm của chuỗi con tuần tự T[s .. s+m-1] trong T với độ dịch chuyển là s, trong đó $0 \leq s \leq n-m$
- p: là giá trị băm của P.

Output :

Khi này thuật toán so sánh lần lượt giá trị ts với p với s chạy từ 0 đến n-m, bước tiếp theo của thuật toán sẽ xảy ra với hai trường hợp như sau:

• TH1: ts = p, thực hiện phép đối sánh chuỗi giữa T[s .. s+m-1] và P[0.. m-1]

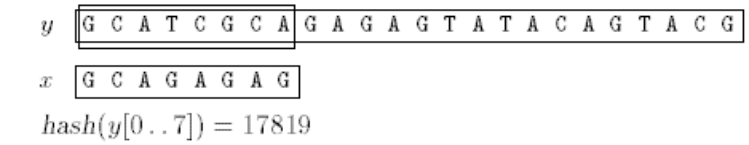
• TH2: ts ≠ p, nếu s ≤ m tính gán s = s+1 và tính tiếp giá trị băm ts .

d.Giải thuật :

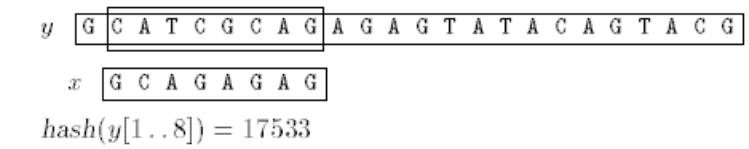
```
#define REHASH(a, b, h) ((((h) - (a)*d) << 1) + (b))

void KR(char *x, int m, char *y, int n) {
   int d, hx, hy, i, j;

   /* Preprocessing */
   /* computes d = 2^(m-1) with
      the left-shift operator */
   for (d = i = 1; i < m; ++i)
      d = (d<<1);

   for (hy = hx = i = 0; i < m; ++i) {
      hx = ((hx<<1) + x[i]);
      hy = ((hy<<1) + y[i]);
   }

   /* Searching */
   j = 0;
   while (j <= n-m) {
      if (hx == hy && memcmp(x, y + j, m) == 0)
         OUTPUT(j);
      hy = REHASH(y[j], y[j + m], hy);
      ++j;
   }

}
int main()
{
    char txt[] = "GCATCGCAGAGAGTATACAGTACG";
    char pat[] = "GCAGAGAG";
    int n=strlen(txt), m=strlen(pat);
    KR(pat, m, txt, n);
    getch();
    return 0;
}
```

e.Kiểm nghiệm

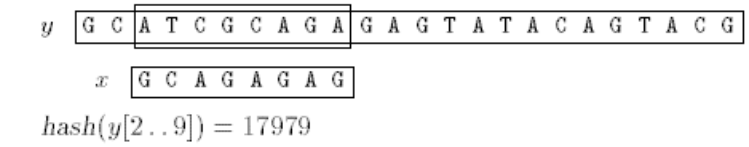
vòng lặp đầu tiên



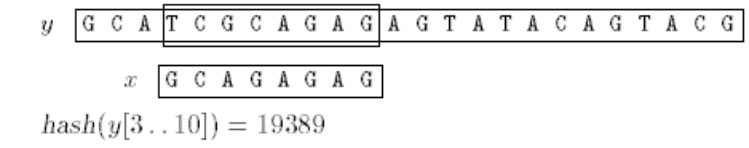
vòng lặp thứ 2



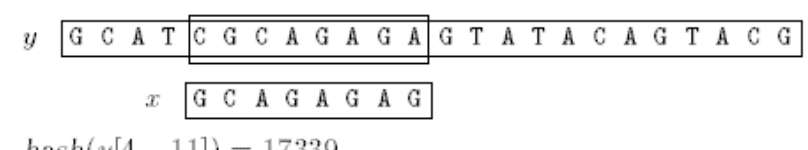
vòng lặp thứ 3



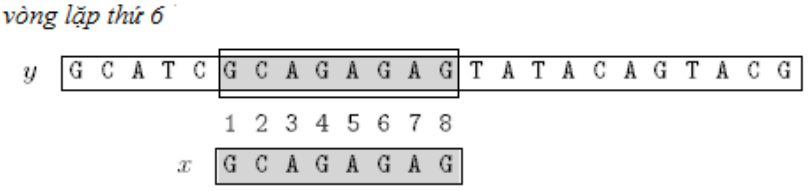
vòng lặp thứ 4



vòng lặp thứ 5



vòng lặp thứ 6



Thuật toán tìm được chuỗi khớp với chuỗi so sánh, thì lưu chuỗi rồi tiếp tục thực hiện như thế cho đến hết.

Kết quả là qua 1 vòng lặp so sánh bộ 8 kí tự và tìm được 1 bộ kí tự ở vòng lặp thứ trùng khớp.

1.3. Thuật toán Shitf-Or

a.Đặc điểm:

- Sử dụng các toán tử thao tác bít (Bitwise).
- Hiệu quả trong trường hợp độ dài mẫu nhỏ hơn một từ máy.
- Thực hiện pha tiền xử lý với thời gian O(m +σ);
- Pha tìm kiếm có độ phức tạp tính toán O(n).

b.Thuật toán

Input :

- Xâu mẫu X =(x_0, x_1,..,x_m), độ dài m.
- Văn bàn Y=(y1, y2,..,yn) độ dài n.

Output:

- Đưa ra mọi vị trí xuất hiện của X trong Y.

c.Mã hóa thuật toán

```
int preSo(char *x, int m, unsigned int S[]) {
    unsigned int j, lim;
    int i;
    for (i = 0; i < ASIZE; ++i)
      S[i] = ~0;
    for (lim = i = 0, j = 1; i < m; ++i, j <<= 1) {
      S[x[i]] &= ~j;
      lim |= j;
    }
    lim = ~(lim>>1);
    return(lim);
  }

  void SO(char *x, int m, char *y, int n) {
    unsigned int lim, state;
    unsigned int S[ASIZE];
    int j;
    if (m > WORD)
      error("SO: Use pattern size <= word size");

    /* Preprocessing */
    lim = preSo(x, m, S);

    /* Searching */
    for (state = ~0, j = 0; j < n; ++j) {
      state = (state<<1) | S[y[j]];
      if (state < lim)
        OUTPUT(j - m + 1);
    }
  }
```

d.Kiểm nghiệm thuật toán

	S_A	S_C	S_G	S_T
G	1	1	0	1
C	1	0	1	1
A	0	1	1	1
G	1	1	0	1
A	0	1	1	1
G	1	1	0	1
A	0	1	1	1
G	1	1	0	1

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
		G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
0	G	0	1	1	1	1	0	1	1	0	1	0	1	0	1	1	1	1	1	1	0	1	1	1	0
1	C	1	0	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	A	1	1	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	G	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	A	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5	G	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
6	A	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
7	G	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1

1.4. Thuật toán Morris-Pratt

a.Đặc điểm:

- Thực hiện từ trái sang phải.
- Có pha tiền xử lý với độ phức tạp O(m).
- Độ phức tạp thuật toán là O(n + m);

b.Thuật toán PreKmp: //thực hiện bước tiền xử lý

Input :

- Xâu mẫu X =(x_0, x_1,..,x_m), độ dài m.

Output: Mảng giá trị kmpNext[].

Formats:

PreKmp(X, m, kmpNext);

Actions:

```
i = 1; kmpNext[0] = 0; len = 0; //kmpNex[0] luôn là 0
     while (i < m) {
          if (X[i] == X[len] ) { //Nếu X[i] = X[len]
          len++; kmpNext[i] = len; i++;
     }
     else { // Nếu X[i] != X[len]
          if ( len != 0 ) { len = kmpNext[len-1]; }
     else { kmpNext[i] = 0; i++; }
     }
}
EndActions
```

c.Mã hóa thuật toán :

```
      void preMp(char *x, int m, int mpNext[]) {
         int i, j;

         i = 0;
         j = mpNext[0] = -1;
         while (i < m) {
            while (j > -1 && x[i] != x[j])
               j = mpNext[j];
            mpNext[++i] = ++j;
         }
      }


      void MP(char *x, int m, char *y, int n) {
         int i, j, mpNext[XSIZE];

         /* Preprocessing */
         preMp(x, m, mpNext);

         /* Searching */
```

```
        i = j = 0;
        while (j < n) {
            while (i > -1 && x[i] != y[j])
                i = mpNext[i];
            i++;
            j++;
            if (i >= m) {
                OUTPUT(j - i);
                i = mpNext[i];
            }
        }
    }
```

d.Kiểm nghiệm thuật toán

• X[] = “GCAGAGAG”, m = 8.

i=?	(X[i]=X[len])?	Len=?	kmpNext[i]=?
		Len = 0	kmpNext[0]= -1
i=1	('C'=='G'):No	Len = 0	kmpNext[1]=0
i=2	('A'=='G'):No	Len = 0	kmpNext[2]=0
i=3	('G'=='G'):Yes	Len = 1	kmpNext[3]=0
i=4	('A'=='C'):No	Len = 0	kmpNext[4]=1
i=5	('G'=='G'):Yes	Len = 1	kmpNext[5]=0
i=6	('A'=='G'):No	Len = 0	kmpNext[6]=1
i=7	('G'=='G'):Yes	Len = 1	kmpNext[7]=0
i=8			kmpNext[8]=1

Lần 1

G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3 4

G C A G A G A G

Dịch: 3 (*i-mpNext*[*i*]=3-0)

Lần 2

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch: 1 (*i-mpNext*[*i*]=0- -1)

Lần 3

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch: 1 (*i*-*mpNext*[*i*]=0- -1)

Lần 4

G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3 4 5 6 7 8

G C A G A G A G

Dịch: 7 (*i*-*mpNext*[*i*]=8-1)

Lần 5

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch: 1 (*i*-*mpNext*[*i*]=1-0)

Lần 6

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch: 1 (*i*-*mpNext*[*i*]=0- -1)

Lần 7

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch: 1 (*i*-*mpNext*[*i*]=0- -1)

Lần 8

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch : 1 (*i-mpNext*[*i*]=0- -1)

Lần 9

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch: 1 (*i-mpNext*[*i*]=0- -1)

1.5. Thuật toán Knuth-Morris-Pratt

a.Trình bày thuật toán

Ý tưởng chính của phương pháp này như sau : trong quá trình tìm kiếm vị trí của mẫu P trong xâu gốc T, nếu tìm thấy một vị trí sai ta chuyển sang vị trí tìm kiếm tiếp theo và quá trình tìm kiếm sau này sẽ được tận dụng thông tin từ quá trình tìm kiếm trước để không phải xét các trường hợp không cần thiết.

b.Đặc điểm

- Thực hiện so sánh từ trái qua phải
- Độ phức tạp thời gian và không gian tiền xử lý *O(m)*
- Độ phức tạp thời gian và không gian xử lý tìm kiếm *O(m+n)*
- Xử lý gần 2n-1 kí tự chữ trong quá trình tìm kiếm

c.Mã hóa thuật toán

Bước tiền xử lý

Input:

Xâu mẫu X = (x_0, x_1,…, x_m) có độ dài m

Output:

Mảng giá trị kmpNext[]

Khởi tạo: PreKMP(X, m, kmpNext);

```
    i = 1; kmpNext[0] = 0; len = 0;
Thực hiện:    while(i<m){
                  if(X[i] == X[len]) { len++; kmpNext[i] = len; i++;}
                  else {
                        if(len!=0) len = kmpNext[len-1];
                        else kmpNext[i] = 0;
                        i++; }
```

}

Bước xử lý

Input :

- Xâu mẫu X =(x0, x1,..,xm), độ dài m.
- Văn bản Y =(y0, y1,..,xn), độ dài n.

Output:

- Tất cả vị trí xuất hiện X trong Y.

Thực hiện:

Bước 1 (Tiền xử lý):

preKmp(x, m, kmpNext); //Tiền xử lý với độ phức tạp O(m)

Bước 2 (Lặp):

```
i = 0; j = 0;
    while (i < n) {
        if ( X[j] == Y[i] ) { i++; j ++; }
        if ( i == m ) {
            < Tìm thấy mẫu ở vị trí i-j>;
            j = kmpNext[j-1]; }
        else if (i < n && X[j]!=Y[i]){
            if(j!=0) j = kmpNext[j-1];
            else i = i +1;
        }
    }
```

Cài đặt thuật toán trên C :

```
void preKmp(char *x, int m, int kmpNext[]) {
   int i, j;

   i = 0;
   j = kmpNext[0] = -1;
   while (i < m) {
      while (j > -1 && x[i] != x[j])
         j = kmpNext[j];
      i++;
      j++;
      if (x[i] == x[j])
         kmpNext[i] = kmpNext[j];
      else
         kmpNext[i] = j;
   }
}


void KMP(char *x, int m, char *y, int n) {
   int i, j, kmpNext[XSIZE];
```

```
   /* Preprocessing */
   preKmp(x, m, kmpNext);

   /* Searching */
   i = j = 0;
   while (j < n) {
      while (i > -1 && x[i] != y[j])
         i = kmpNext[i];
      i++;
      j++;
      if (i >= m) {
         OUTPUT(j - i);
         i = kmpNext[i];
      }
   }
}
```

d.Kiểm nghiệm thuật toán

Bước tiền xử lý

- Với X[] = "ABABCABAB", m = 9

i=?	(X[i]== X[len])?	Len =?	kmpNext[i]=?
		Len =0	kmpNext[0]=0
i=1	('B'=='A'): No	Len =0	kmpNext[1]=0
i=2	('A'=='A'): Yes	Len =1	kmpNext[2]=1
i=3	('B'=='B'): Yes	Len=2	kmpNext[3]=2
i=4	('C'=='A'): No	Len=0	kmpNext[4]=0
i=5	('A'=='A'): Yes	Len=1	kmpNext[5]=1
i=6	('B'=='B'): Yes	Len=2	kmpNext[6]=2
i=7	('A'=='A'): Yes	Len=3	kmpNext[6]=3
i=8	('B'=='B'): Yes	Len=4	kmpNext[6]=4
Kết luận: kmpNext[] = {0, 0, 1, 2, 0, 1, 2, 3, 4}.			

Bước xử lý

Knuth-Moriss-Patt (X, m, Y, n)

• X[] = "ABABCABAB", m = 9.

• Y[] = "ABABDABACDABABCABAB", n = 19

Bước 1 (Tiền xử lý).

Thực hiện Prekmp(X, m, kmpNext) ta nhận được: kmpNext[] = { 0, 0, 1, 2, 0, 1, 2, 3, 4}

Bước 2 (Lặp):

(X[j]==Y[i])?	(J ==9)?	I =? J = ?	
(X[0]==Y[0]): Yes	No	i=1, j=1	
(X[1]==Y[1]): Yes	No	i=2, j=2	
(X[2]==Y[2]): Yes	No	i=3, j=3	
(X[3]==Y[3]): Yes	No	i=4, j=4	
(X[4]==Y[4]): No	No	i=4, j=2	
(X[2]==Y[4]): No	No	i=4, j=0	
(X[0]==Y[4]): No	No	i=5, j=0	
(X[0]==Y[5]): Yes	No	i=6, j=1	
(X[1]==Y[6]): Yes	No	i=7, j=2	
.....			

1.6. Thuật toán Not So Naive

a.Đặc điểm:

- Thực hiện từ trái sang phải.
- Pha tiền xử lý có độ phức tạp hằng số.
- Độ phức tạp về không gian là hằng số
- Pha tìm kiếm có độ phức tạp thuật toán là O(n.m);

b.Thuật toán Not So Naive:

Input :

- Xâu mẫu X =(x0, x1,..,xm), độ dài m.
- Văn bàn Y=(y1, y2,..,yn) độ dài n.

Output:

- Đưa ra mọi vị trí xuất hiện của X trong Y.

Formats: k = NSN(X, m, Y, n);

Actions:

Bước 1 (Tiền xử lý): Xác định số bước dịch chuyển SubY tren Y

- Gọ i:

- SubY: chuỗi con gồm m phần tử (y[j.. j+m-1]) của Y
- k: số bước dịch khi x[1] != SubY[1]
- l: số bước dịch khi x[1] = SubY[1]

- Xác định số dịch chuyển chuỗi con SubY khi chưa thỏa mãn:
  - Nếu (x[0] = x[1] && x[1] != y[j+1]) --> x[0] != y[j+1] --> SubY dịch chuyển 2 đơn vị j = j+2 --> k =2
  - Nếu (x[0] != x[1] && x[1] = y[j+1]) --> x[0] != y[j+1] --> SubY dịch chuyển 2 đơn vị j = j+2 --> l =2
  - Còn lại SubY dịch chuyển 1 đơn vị j = j+1

```
/* Preprocessing */
        if (x[0] == x[1]) {
                k = 2; // dịch chuyển 2 nếu x[1] != suby[1]
                l = 1; // ngược lại dịch chuyển 1
        }
        else {
                k = 1; // dịch chuyển 1 nếu x[1] != suby[1]
                l = 2; // ngược lại dịch chuyển 2
        }
```

Bước 2 (Tìm kiếm):

- Ký tự của SubY được so sánh với chuỗi mẫu X theo thứ tự 1, 2, ... , m-2, m-1, 0
- Nếu SubY = chuỗi mẫu thì in ra j (vị trí bắt đầu của SubY trên Y)
- Ngược lại di chuyển SubY trên Y

EndActions.

c.Cài đặt thuật toán

```
void NSN(char *x, int m, char *y, int n) {
   int j, k, ell;

   /* Preprocessing */
   if (x[0] == x[1]) {
      k = 2;
      ell = 1;
   }
   else {
      k = 1;
      ell = 2;
   }

   /* Searching */
   j = 0;
   while (j <= n - m)
      if (x[1] != y[j + 1])
         j += k;
      else {
         if (memcmp(x + 2, y + j + 2, m - 2) == 0 &&
             x[0] == y[j])
            OUTPUT(j);
         j += ell;
      }
}
```

d.Kiểm nghiệm thuật toán

+ Pha tiền xử lý

X[0] = 'G'

X[1] = 'C'

X[0] != X[1] nên k = 1, l =2

+ Pha tìm kiếm

J = 0, k =1, l =2

Lần 1

G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3

G C A G A G A G

Do X[1] = SubY[1], số bước dịch chuyển: 2 (*l*)

Lần 2

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Do X[1] != SubY[1], số bước dịch chuyển: 1 (*k*)

Lần 3

G C A T C G C A G A G A G T A T A C A G T A C G

1 2

G C A G A G A G

Dịch chuyển: 2 (*l*)

Lần 4

G C A T C G C A G A G A G T A T A C A G T A C G

8 1 2 3 4 5 6 7

G C A G A G A G

In ra vị trí chuỗi SubY thỏa mãn: 5 và dịch chuyển: 2 (*l*)

Lần 5

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch chuyển: 1 (*k*)

Lần 6

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch chuyển: 1 (*k*)

Lần 7

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch chuyển: 1 (*k*)

Lần 8

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch chuyển: 1 (*k*)

Lần 9

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch chuyển: 1 (*k*)

Lần 10

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch chuyển: 1 (*k*)

Lần 11

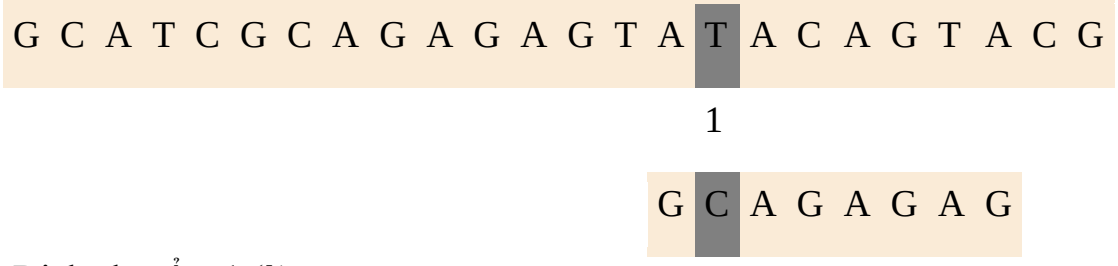
G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

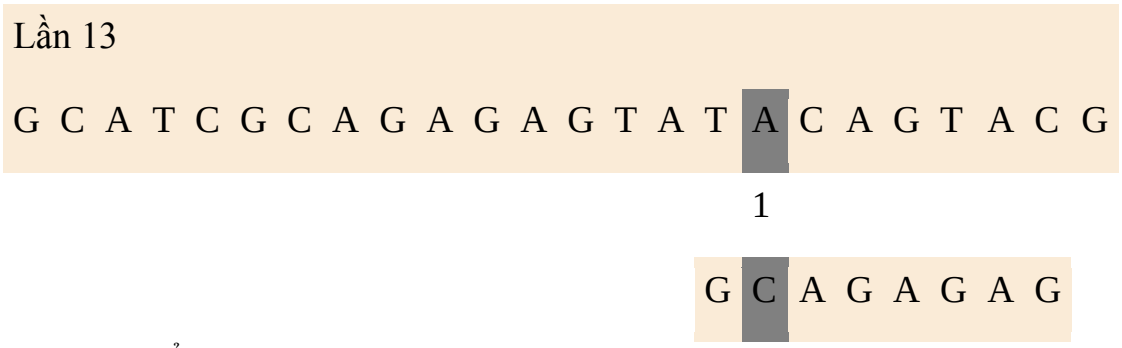
Dịch chuyển: 1 (*k*)

Lần 12



Dịch chuyển: 1 (*k*)

Lần 13



Dịch chuyển: 1 (*k*)

Lần 14

Dịch chuyển: 2 (*l*)

Thuật toán dừng. Kết quả in ra vị trí bắt đầu của chuỗi con: Y[5].

1.7. Thuật toán Apostolico - Crochemore

a.Đặc điểm:

- Thực hiện từ trái sang phải
- Pha tiền xử lý có độ phức tạp về không gian và thời gian là $O(m)$
- Pha tìm kiếm có độ phức tạp về thời gian là $O(n)$
- Trong trường hợp xấu nhất, thuật toán thực hiện so sánh $\frac{3}{2}n$ ký tự

b.Mô tả thuật toán Apostolico-Crochemore

Input:

- Xâu mẫu X = (x_0, x_1,…, x_m), độ dài m.

- Văn bản Y = $(y_1, y_2, \ldots, y_n)$, độ dài n

Output:

- Đưa ra mọi vị trí xuất hiện của X trong Y.

Formats: Apostolico-Crochemore(X, m, Y, n);

Actions:

Bước 1 (Tiền xử lý):

- Xây dựng bảng dịch để tính toán bước dịch: **preKMP(x, m, kmpNext)**. //sử dụng thuật toán xây dựng bảng dịch của phương pháp Knuth-Morris-Pratt.
- Tìm vị trí ký tự khác nhau đầu tiên trong xâu mẫu X, ký hiệu là ***l***.
    - ***l*** = 0 nếu x chỉ chứa 1 loại ký tự, vd: aaaaaa.
    - ***l*** = vị trí của ký tự đầu tiền của X khác với ký tự x0.

Bước 2 (Tìm kiếm):

- Việc so sánh được tính toán trên mô hình các vị trí theo thứ tự sau:

$$l, l+1, \ldots, m-2, m-1, 0, 1, \ldots, l-1.$$

- Xét bộ ba ***(i, j, k)*** thỏa mãn:
    - Cửa sổ chạy được xác định bởi factor $[y_j, y_{j+1}, \ldots, y_{j+m-1}]$
    - $0 \le k \le l$ và $[x_0, x_1, \ldots, x_{k-1}] = [y_j, y_{j+1}, \ldots, y_{j+k-1}]$
    - $l \le i \le m$ và $[x_l, x_{l+1}, \ldots, x_i] = [y_{j+l}, y_{j+l+1}, \ldots, y_{i+j-1}]$
- Cách tính toán bộ ba ***(i, j, k)*** tiếp theo:

If ($i = l$) {
    If($x_i = y_{i+j}$){
        Bộ 3 tiếp theo là $(i + 1, j, k)$
    }else{
        Bộ 3 tiếp theo là $(l, j+1, \max\{0, k-1\})$
    }
}else if($l < i < m$) {
    If($x_i = y_{i+j}$){

Bộ 3 tiếp theo là **(*i* + 1, *j*, *k*)**

}else{

If(**kmpNext[i]** ≤ ***l*)**{

Bộ 3 tiếp theo là **(*l*, *i* + *j* – *kmpNext[i]*, max{0, *kmpNext[i]*, }))**

}else{

Bộ 3 tiếp theo là **(*kmpNext[i]*, *i* + *j* – *kmpNext[i]*, *l*)**

}

}

}else if(i = m){

If ($k \leq l$ && $x_k = y_{k+j}$){

Bộ 3 tiếp theo là **(*i*, *j*, *k* + 1)**

}else{

If (k = l){

<tìm thấy mẫu ở vị trí j – i >

}

Tính bộ 3 theo trường hợp ***l* < *i* < *m*.**

}

}

c.Code triển khai

```
void preKmp(char *x, int m, int kmpNext[]) {
   int i, j;

   i = 0;
   j = kmpNext[0] = -1;
   while (i < m) {
      while (j > -1 && x[i] != x[j])
         j = kmpNext[j];
      i++;
      j++;
      if (x[i] == x[j])
         kmpNext[i] = kmpNext[j];
      else
         kmpNext[i] = j;
   }
}

void AXAMAC(char *x, int m, char *y, int n) {
   int i, j, k, ell, kmpNext[XSIZE];

   /* Preprocessing */
```

```
   preKmp(x, m, kmpNext);
   for (ell = 1; x[ell - 1] == x[ell]; ell++);
   if (ell == m)
      ell = 0;

   /* Searching */
   i = ell;
   j = k = 0;
   while (j <= n - m) {
      while (i < m && x[i] == y[i + j])
         ++i;
      if (i >= m) {
         while (k < ell && x[k] == y[j + k])
            ++k;
         if (k >= ell)
            OUTPUT(j);
      }
      j += (i - kmpNext[i]);
      if (i == ell)
         k = MAX(0, k - 1);
      else
         if (kmpNext[i] <= ell) {
            k = MAX(0, kmpNext[i]);
            i = ell;
         }
         else {
            k = ell;
            i = kmpNext[i];
         }
   }
}
```

d.Kiểm nghiệm thuật toán

i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$kmpNext[i]$	−1	0	0	−1	1	−1	1	−1	1

$\ell = 1$

Vòng lặp thứ nhất

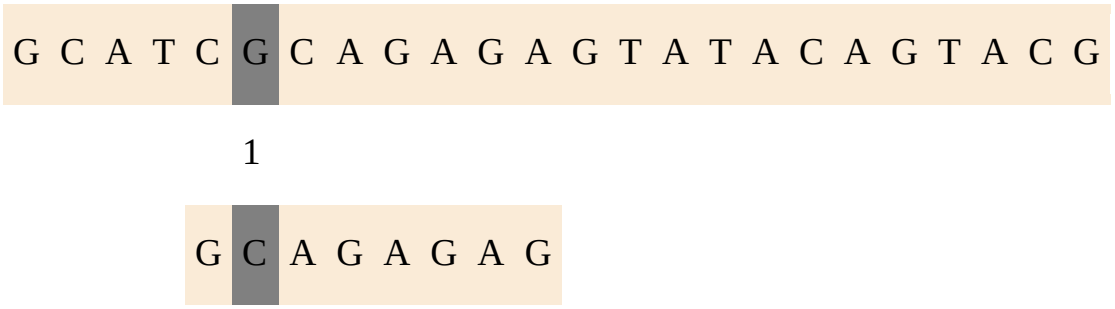
G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3

G C A G A G A G

Số bước dịch chuyển: 4 (*i-kmpNext*[*i*]=3- -1)

Vòng lặp thứ hai

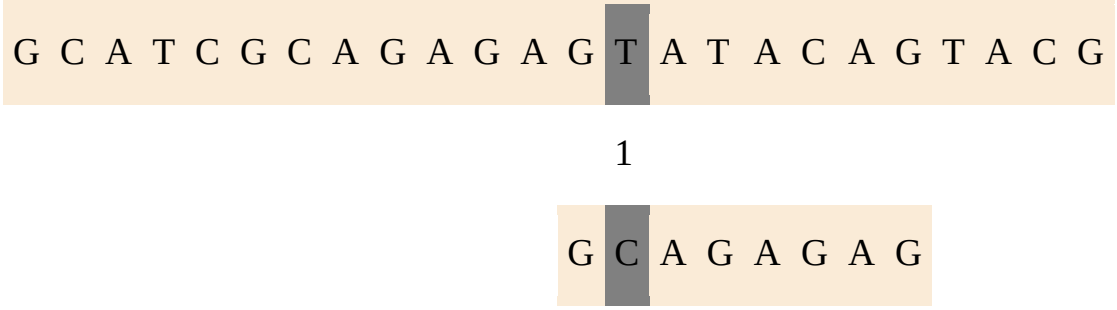


Số bước dịch chuyển: 1 (*i-kmpNext*[*i*]=1-0)

Vòng lặp thứ ba

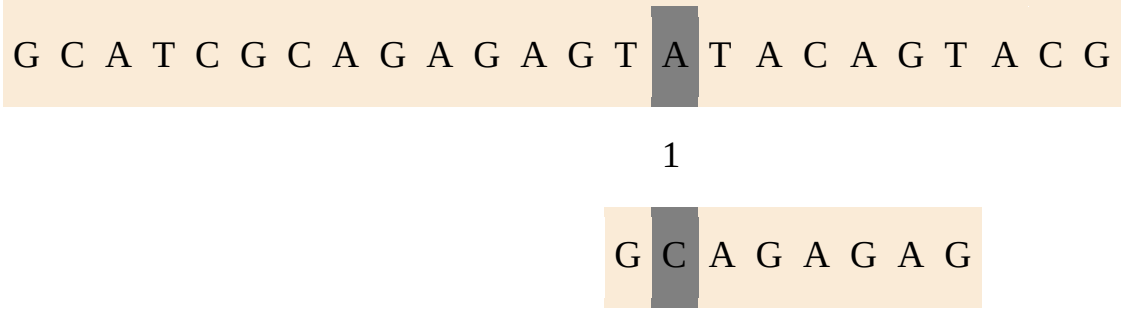
Số bước dịch chuyển: 7 (*i-kmpNext*[*i*]=8-1)

Vòng lặp thứ tư



Số bước dịch chuyển: 1 (*i-kmpNext*[*i*]=1-0)

Vòng lặp thứ năm



Số bước dịch chuyển: 1 (*i-kmpNext*[*i*]=1-0)

Vòng lặp thứ sáu

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Số bước dịch chuyển: 1 (*i*-*kmpNext*[*i*]=1-0)

Vòng lặp thứ bảy

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Số bước dịch chuyển: 1 (*i*-*kmpNext*[*i*]=1-0)

Vòng lặp thứ bảy

G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3 4

G C A G A G A G

2. Các thuật toán tìm kiếm từ phải qua trái

2.1. Thuật toán Colussi

a.Đặc điểm

- Sàng lọc lại thuật toán Knutt-Morris-Pratt;
- Phân vùng tập các vị trí mẫu thành 2 tập con rời nhau; các vị trí trong tập đầu tiên được từ trái qua phải và khi không có sự phù hợp xảy ra các vị trí trong tập con thứ 2 sẽ được quét từ phải qua trái;
- Pha tiền xử lý có độ phức tạp không gian và thời gian là O(m);
- Pha tìm kiếm có độ phức tạp thời gian là O(n);
- Trong trường hợp xấu nhất phải thực hiện $\frac{3}{2}n$ so sánh ký tự văn bản

b.Mô tả

Việc thiết kế thuật toán Colussi tuân theo một phân tích có tính chặt chẽ của thuật toán Knutt-Morris-Pratt

Tập các vị trí mẫu được phân chia thành 2 tập con rời nhau. Sau đó, mỗi mẫu thử bao gồm 2 pha:

- Trong pha đầu tiên, các so sánh được thực hiện từ trái qua phải với các ký tự văn bản phù hợp với vị trí mẫu mà giá trị của hàm kmpNext hoàn toàn lớn hơn -1. Những vị trí đó được gọi là ***noholes;***
- Pha thứ 2 bao gồm việc so sánh các vị trí còn lại (được gọi là ***holes***) từ phải qua trái.

Chiến lược này có 2 ưu điểm:

- Khi một không phù hợp xảy ra trong pha đầu tiên, sau khi dịch chuyển thích hợp không cần thiết phải so sánh ký tự văn bản phù hợp với noholes được so sánh trong suốt mẫu thử trước;
- Khi một không phù hợp xảy ra trong pha thứ 2 điều đó có nghĩa là một hậu tố của mẫu thử phù hợp với một nhân tố của văn bản, sau khi dịch chuyển tương ứng một tiền tố của mẫu thử cũng sẽ vẫn phù hợp với một nhân tố của văn bản, do đó không cần thiết phải so sánh lại với nhân tố đó nữa.

c.Mã hóa thuật toán

```
int preColussi(char *x, int m, int h[], int next[],
               int shift[]) {
   int i, k, nd, q, r, s;
```

```
   int hmax[XSIZE], kmin[XSIZE], nhd0[XSIZE],
rmin[XSIZE];

   /* Computation of hmax */
   i = k = 1;
   do {
      while (x[i] == x[i - k])
         i++;
      hmax[k] = i;
      q = k + 1;
      while (hmax[q - k] + k < i) {
         hmax[q] = hmax[q - k] + k;
         q++;
      }
      k = q;
      if (k == i + 1)
         i = k;
   } while (k <= m);

   /* Computation of kmin */
   memset(kmin, 0, m*sizeof(int));
   for (i = m; i >= 1; --i)
      if (hmax[i] < m)
         kmin[hmax[i]] = i;

   /* Computation of rmin */
   for (i = m - 1; i >= 0; --i) {
      if (hmax[i + 1] == m)
         r = i + 1;
```

```
   if (kmin[i] == 0)
      rmin[i] = r;
   else
      rmin[i] = 0;
}
/* Computation of h */
s = -1;
r = m;
for (i = 0; i < m; ++i)
   if (kmin[i] == 0)
      h[--r] = i;
   else
      h[++s] = i;
nd = s;

/* Computation of shift */
for (i = 0; i <= nd; ++i)
   shift[i] = kmin[h[i]];
for (i = nd + 1; i < m; ++i)
   shift[i] = rmin[h[i]];
shift[m] = rmin[0];

/* Computation of nhd0 */
s = 0;
for (i = 0; i < m; ++i) {
   nhd0[i] = s;
   if (kmin[i] > 0)
      ++s;
}
```

```
   /* Computation of next */
   for (i = 0; i <= nd; ++i)
      next[i] = nhd0[h[i] - kmin[h[i]]];
   for (i = nd + 1; i < m; ++i)
      next[i] = nhd0[m - rmin[h[i]]];
   next[m] = nhd0[m - rmin[h[m - 1]]];

   return(nd);
}

void COLUSSI(char *x, int m, char *y, int n) {
   int i, j, last, nd,
       h[XSIZE], next[XSIZE], shift[XSIZE];

   /* Processing */
   nd = preColussi(x, m, h, next, shift);

   /* Searching */
   i = j = 0;
   last = -1;
   while (j <= n - m) {
      while (i < m && last < j + h[i] &&
                      x[h[i]] == y[j + h[i]])
         i++;
      if (i >= m || last >= j + h[i]) {
         OUTPUT(j);
         i = m;
```

```
        }
        if (i > nd)
            last = j + m - 1;
        j += shift[i];
        i = next[i];
    }
}
```

d.Kiểm nghiệm thuật toán

Văn bản Y: GCATCGCAGAGAGTATACAGTACG
Mẫu X: GCAGAGAG

Bước tiền xử lý:

i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$kmpNext[i]$	-1	0	0	-1	1	-1	1	-1	1
$kmin[i]$	0	1	2	0	3	0	5	0	
$h[i]$	1	2	4	6	7	5	3	0	
$next[i]$	0	0	0	0	0	0	0	0	0
$shift[i]$	1	2	3	5	8	7	7	7	7
$hmax[i]$	0	1	2	4	4	6	6	8	8
$rmin[i]$	7	0	0	7	0	7	0	8	
$ndh0[i]$	0	0	1	2	2	3	3	4	

$nd = 3$

Bước tìm kiếm:

Lần thử đầu tiên

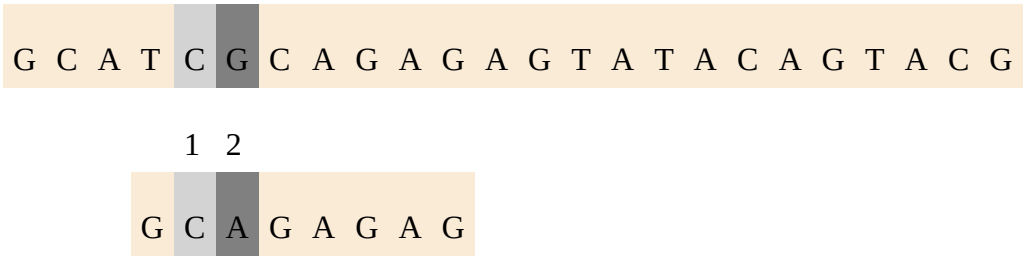
G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3

G C A G A G A G

Shift by: 3 (*shift*[2])

Lần thử thứ 2

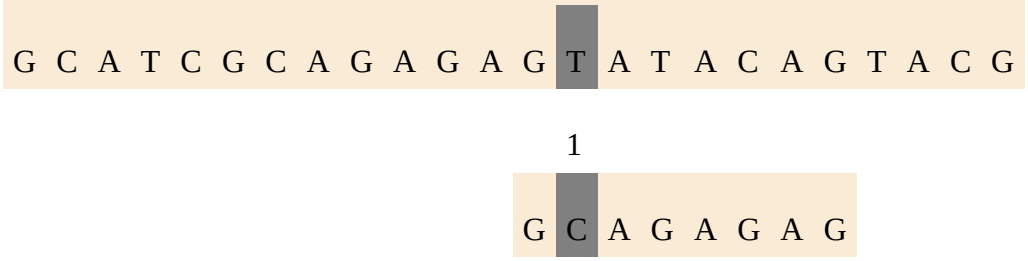


Shift by: 2 (*shift*[1])

Lần thử thứ ba

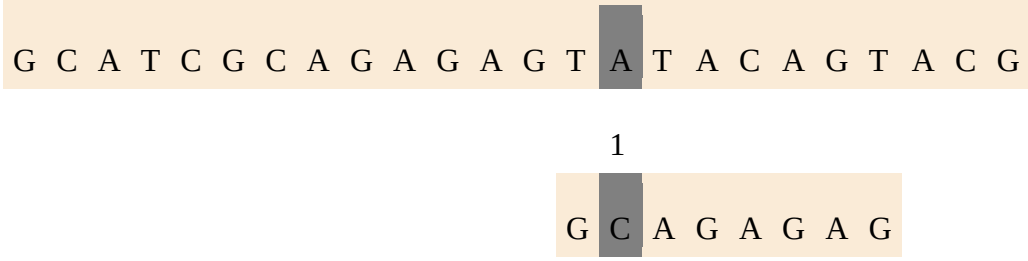
Shift by: 7 (*shift*[8])

Lần thử thứ tư



Shift by: 1 (*shift*[0])

Lần thử thứ năm

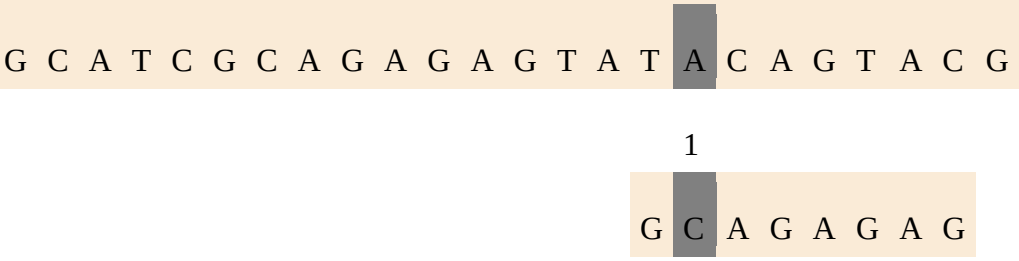


Shift by: 1 (*shift*[0])

Lần thử thứ sáu

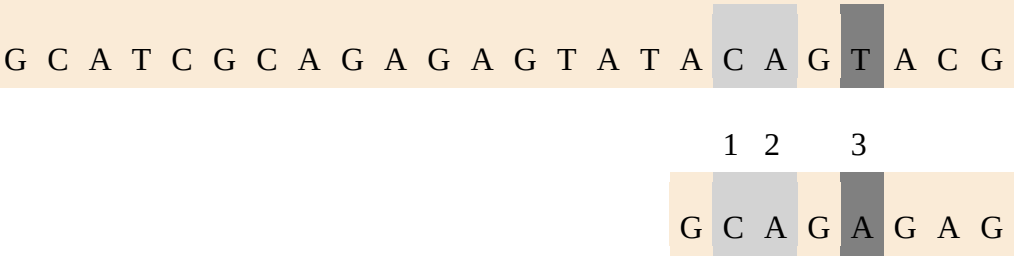
Shift by: 1 (*shift*[0])

Lần thử thứ bảy



Shift by: 1 (*shift*[0])

Lần thử thứ tám



Shift by: 3 (*shift*[2])

Thuật toán Colussi thực hiện so sánh 20 ký tự trong ví dụ nêu trên.

2.2. Thuật toán Boyer – Moore

a.Phát biểu thuật toán

- Thực hiện việc so sánh từ phải sang trái.
- Giai đoạn tiền xử lý (preprocessing) có độ phức tạp thời gian và không gian là O(m+σ).
- Giai đoạn tìm kiếm có độ phức tạp O(m*n).
- So sánh tối đa 3n ký tự trong trường hợp xấu nhất đối với mẫu không có chu kỳ (non periodic pattern).
- Độ phức tạp O(n/m) trong trường hợp tốt nhất.
- Thuật toán Boyer-Moore được coi là thuật toán hiệu quả nhất trong vấn đề tìm kiếm chuỗi (string-matching) trong các ứng dụng thường gặp. Các biến thể của nó thường được dùng trong các bộ soạn thảo cho các lệnh như <<search>> và <<subtitute>>.
- Thuật toán sẽ quét các ký tự của mẫu (pattern) từ phải sang trái bắt đầu ở phần tử cuối cùng.
- Trong trường hợp mis-match (hoặc là trường hợp đã tìm được 1 đoạn khớp với mẫu), nó sẽ dùng 2 hàm được tính toán trước để dịch cửa sổ sang bên phải. Hai hàm dịch chuyển này được gọi là good-suffix shift (còn được biết với cái tên phép dịch chuyển khớp) và bad-character shift (còn được biết với cái tên phép dịch chuyển xuất hiện).

b.Mã hóa thuật toán

```
void preBmBc(char *x, int m, int bmBc[]) {
   int i;

   for (i = 0; i < ASIZE; ++i)
      bmBc[i] = m;
   for (i = 0; i < m - 1; ++i)
      bmBc[x[i]] = m - i - 1;
}


void suffixes(char *x, int m, int *suff) {
   int f, g, i;

   suff[m - 1] = m;
   g = m - 1;
   for (i = m - 2; i >= 0; --i) {
      if (i > g && suff[i + m - 1 - f] < i - g)
         suff[i] = suff[i + m - 1 - f];
      else {
         if (i < g)
            g = i;
```

```
            f = i;
            while (g >= 0 && x[g] == x[g + m - 1 - f])
               --g;
            suff[i] = f - g;
         }
      }
   }

void preBmGs(char *x, int m, int bmGs[]) {
   int i, j, suff[XSIZE];

   suffixes(x, m, suff);

   for (i = 0; i < m; ++i)
      bmGs[i] = m;
   j = 0;
   for (i = m - 1; i >= 0; --i)
      if (suff[i] == i + 1)
         for (; j < m - 1 - i; ++j)
            if (bmGs[j] == m)
               bmGs[j] = m - 1 - i;
   for (i = 0; i <= m - 2; ++i)
      bmGs[m - 1 - suff[i]] = m - 1 - i;
}


void BM(char *x, int m, char *y, int n) {
   int i, j, bmGs[XSIZE], bmBc[ASIZE];

   /* Preprocessing */
   preBmGs(x, m, bmGs);
   preBmBc(x, m, bmBc);

   /* Searching */
   j = 0;
```

```
    while (j <= n - m) {
       for (i = m - 1; i >= 0 && x[i] == y[i + j]; --i);
       if (i < 0) {
          OUTPUT(j);
          j += bmGs[0];
       }
       else
          j += MAX(bmGs[i], bmBc[y[i + j]] - m + 1 + i);
    }
}
```

c.Kiểm nghiệm thuật toán

- Bước tiền xử lý

c	A	C	G	T
$bmBc[c]$	1	6	2	8

i	0	1	2	3	4	5	6	7
$x[i]$	G	C	A	G	A	G	A	G
$suff[i]$	1	0	0	2	0	4	0	8
$bmGs[i]$	7	7	7	2	7	4	7	1

Bảng bmBc và *bmGs* được sử dụng bởi thuật toán Boyer – Moore

- Bước tìm kiếm

Bước 1:

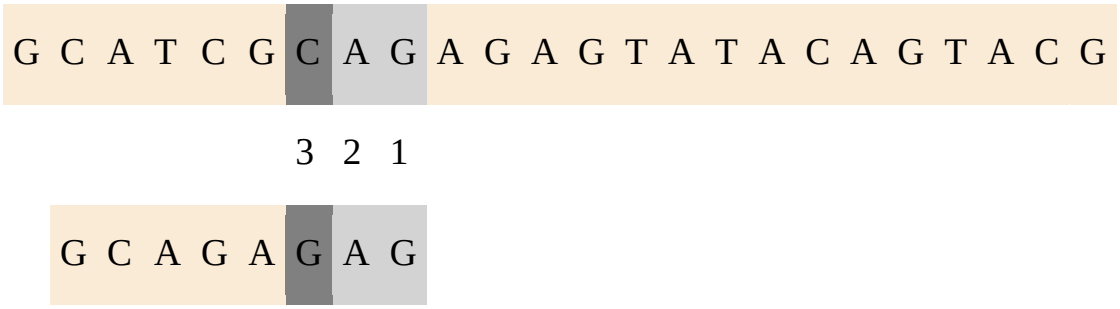
G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

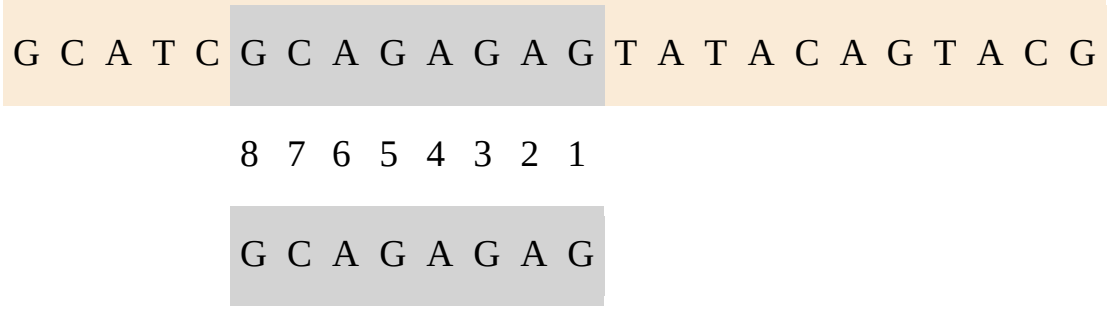
Dịch: 1 (*bmGs*[7]=*bmBc*[A]-8+8)

Bước 2:



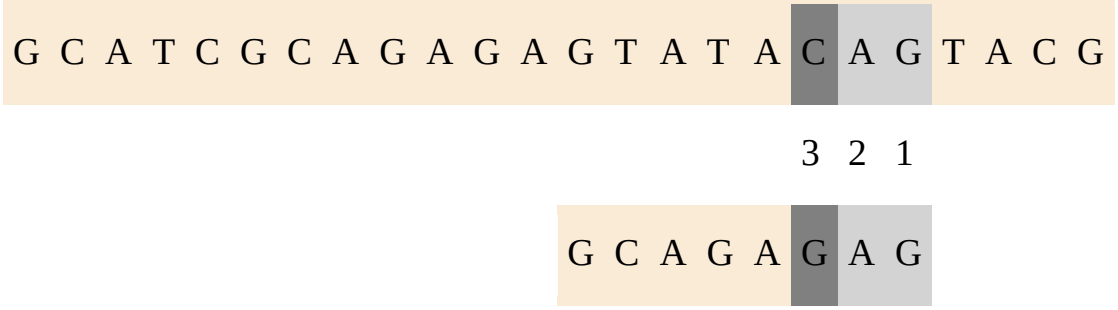
Dịch: 4 (*bmGs*[5]=*bmBc*[C]-8+6

Bước 3:



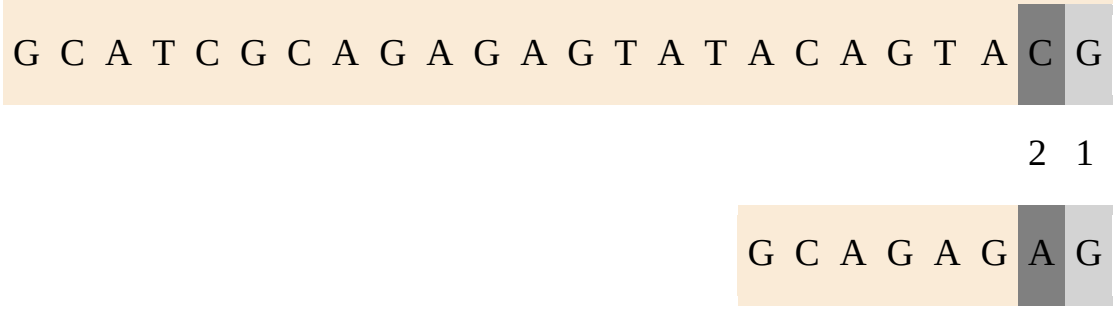
Dịch: 7 (*bmGs*[0])

Bước 4:



Dịch: 4 (*bmGs*[5]=*bmBc*[C]-8+6)

Bước 5:



Dịch: 7 (*bmGs*[6])

2.3. Thuật toán Turbo-BM

a.Đặc điểm của thuật toán

- Đây là 1 biến thể của thuật toán Boyer-Moore
- Không yêu cầu pha tiền xử lý như thuật toán Boyer-Moore
- Cần không gian nhớ phụ như với thuật toán Boyer-Moore;
- Pha tiền xử lý có độ phức tạp O(m+σ);
- Pha tìm kiếm có độ phức tạp thuật toán là O(n);

b.Giải thích thuật toán Turbo-BM:

Thuật toán Turbo-Bm là 1 thuật toán được cải thiện từ thuật toán Boyer-Moore. Thuật toán này không cần pha tiền xử lý mà chỉ cần cung cấp không gian nhớ phụ . Nó bao gồm việc ghi nhớ các thành phần của các kí tự khớp với hậu tố của các mẫu trong lần thử cuối cùng (và chỉ thực hiện nếu có sự dịch hậu tố tốt được thực hiện).

Kỹ thuật này có 2 lợi ích:

- Có thể nhảy qua thành phần

- Có thể thực hiện dịch chuyển nhanh

Một dịch chuyển nhanh có thể thực hiện ra nếu trong quá trình xử lý hiện tại của các hậu tố của mẫu phù hợp với các kí tự ngắn hơn các xử lý trước đó. Trong trường hợp này chúng ta hãy gọi u là yếu tố nhớ và v là các hậu tố xuất hiện trong các xử lý hiện tại như vậy uzv là một hậu tố của x. Hãy để a và b là các phần tử không phù hợp trong các xử lý hiện tại trong các mẫu và các kĩ tự tương ứng. Sau đó, av là một hậu tố của x. Hai kí tự a, b xảy ra tại khoảng cách p trong chuỗi kí tự, và các hậu tố của x có chiều dài |uzv| có một đọ dài thời gian là p = |ZV| kể từ u là một biên giới của uzv, do đó nó không thể chồng lên nhau cả hai lần xuất hiện của hai nhân vật a và b khác nhau, tại khoảng cách p trong chuỗi kí tự. Sự dịch chuyển nhỏ nhất có thể có chiều dài |u| - |v|, mà chúng ta gọi là dịch chuyển nhanh (turbo-shift)

Tuy nhiên trong trường hợp nơi | v | <| u | nếu chiều dài của sự thay đổi phần tử-

tồi lớn hơn độ dài của sự dịch chuyển của hậu tố - tốt và độ dài của dịch chuyển nhanh sau đó chiều dài của sự chuyển đổi thực tế phải lớn hơn hoặc bằng |u| + 1.

Thật vậy (xem hình 15.2), trong trường hợp này hai phần tử c và d là khác nhau vì chúng ta giả định rằng sự thay đổi trước đó là một sự dịch chuyển hậu tố tốt. Sau đó, một sự thay đổi lớn hơn dịch chuyển nhanh nhưng nhỏ hơn |u| + 1 sẽ sắp xếp c và d, với một kí tự tương tự trong v. Vì vậy, nếu trường hợp này chiều dài của sự thay đổi thực tế phải có ít nhất bằng |u| + 1.
Giai đoạn tiền xử lý có thể được thực hiện trong O (m+ σ) thời gian. Giai đoạn tìm kiếm là O (n). Các số so sánh phần tử trong đoạn mã được thực hiện bởi các thuật toán Turbo-BM được bao bọc bởi 2n.

c.Cài đặt thuật toán C

```
void TBM(char *x, int m, char *y, int n) {
     int bcShift, i, j, shift, u, v, turboShift,
     bmGs[XSIZE], bmBc[ASIZE];
     /* Preprocessing */
     preBmGs(x, m, bmGs);
     preBmBc(x, m, bmBc);
     /* Searching */
     j = u = 0;
     shift = m;
     while (j <= n - m) {
     i = m - 1;
          while (i >= 0 && x[i] == y[i + j]) {
          --i;
          if (u != 0 && i == m - 1 - shift)
          i -= u;
          }
     if (i < 0) {
     OUTPUT(j) ;
```

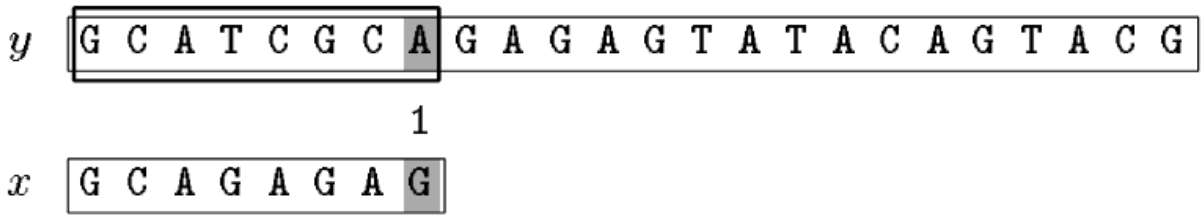
```
            shift = bmGs[0];
            u = m - shift;
            }
            else {
                  v = m - 1 - i;
                  turboShift = u - v;
                  bcShift = bmBc[y[i +j]] -m+l+i;
                  shift = MAX(turboShift, bcShift);
                  shift = MAX(shift, bmGs[i]);
                  if (shift == bmGs[i])
                        u = MIIF(m - shift, v);
                        else {
                              if (turboShift < bcShift)
                              shift = MAX(shift, u + 1);
                              u = 0;
                        }
                  }
                  j += shift;
            }
      }
```

d.Kiểm nghiệm thuật toán

a	*A C G T*
bmBc[a]	*1 6 2 8*

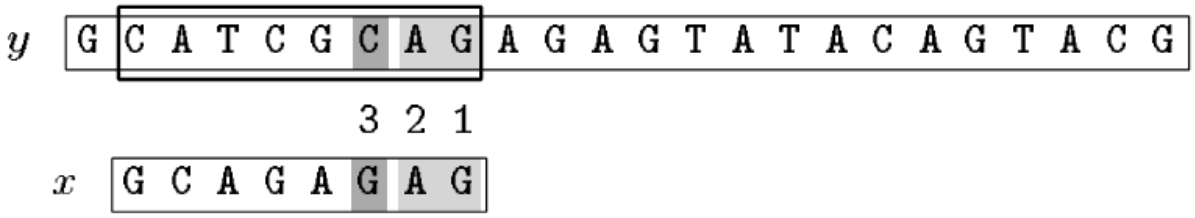
i	0	1	2	3	4	5	6	7
$x[i]$	G	C	A	G	A	G	A	G
$suff[i]$	1	0	0	2	0	4	0	8
$bmGs[i]$	7	7	7	2	7	4	7	1

- Thực hiện lần thứ nhất



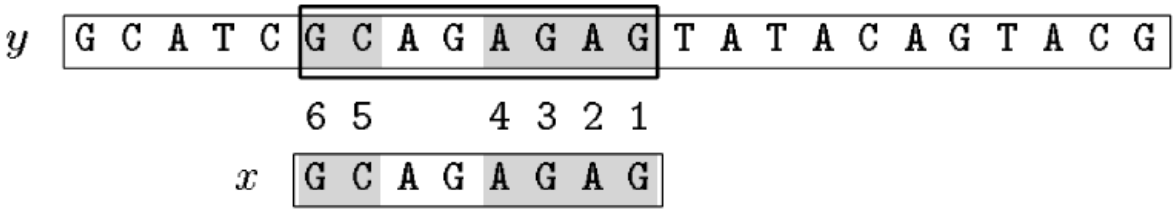
Dịch chuyển 1 (bmGs[7]=bmBc[A]-7+7)

- Thực hiện lần thứ 2



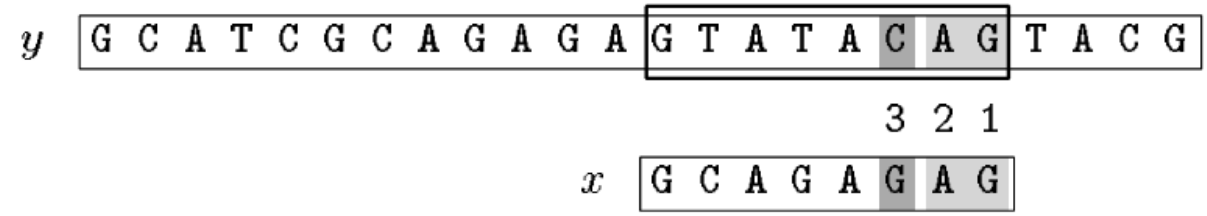
Dịch chuyển 4 (bmGs[5]=bmBc[C]-7+5)

- Thực hiện lần thứ 3



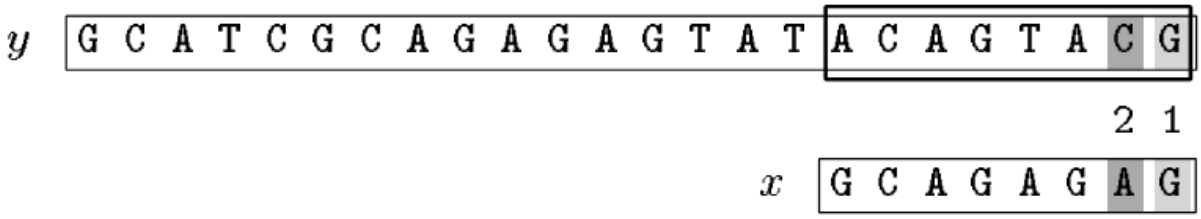
Dịch chuyển 7 (bmGs[0])

- Thực hiện lần thứ 4



Dịch chuyển 4 (bmGs[5]=bmBc[C]-7+5)

- Thực hiện lần thứ 5



Dịch chuyển 7 (bmGs[6])

2.4. Thuật toán Colussi nghịch đảo

a.Đặc điểm :

- Được tinh chỉnh lại từ thuật toán Boyer-Moore
- Phân chia tập các vị trí mẫu thành hai tập nhỏ hơn tách rời nhau
- Pha tiền xử lý có độ phức tạp O(m^2) về thời gian và O(m x σ) về không gian
- Pha tìm kiếm có độ phức tạp thuật toán là O(n)

b.Thuật toán Colussi nghịch đảo

Input:

- Xâu mẫu X = (x_0, x_1,…,x_m), độ dài m.
- Văn bản Y = (y_0,y_1,…x_n), độ dài n

Output:

- Tất cả vị trí xuất hiện X trong Y

Format: k = RC(X, m, Y, n)

Actions:

Bước 1 (tiền xử lý):

preRC(x, m, h, rcBc, rcGs)// rcBc là tập ký tự tồi, rcGs tập hậu tố tốt

Bước 2 tìm kiếm

```
void RC(char *x, int m, char *y, int n){
    int i, j, s, rcBc[ASIZE][XSIZE], rcGs[XSIZE], h[XSIZE];
    /*Preprocessing*/
preRc (x, m, h, rcBc, rcGs);// Tính rcBc thông qua locc, tính rcGs[i], h[i] thông qua hmin, kmin, rmin
    /*Searching*/
    j = 0;
    s = m;
    while (j <= n - m){
        while (j <= n-m && x[m - 1] != y[j + m - 1]){
            s = rcBc[y[j + m - 1]][s];
            j += s;
        }
```

```
            for (i = 1; i < m && x[h[i]] == y[j + h[i]]; ++i);
            if (i >= m)
                OUTPUT(j);
            s = rcGs[i];
            j += s;
        }
}
```

c.Bước kiểm nghiệm

Tính locc[a]

a	A	C	G	T
locc[*a*]	6	1	5	-1

Tính rcBc thông qua locc

rcBc	1	2	3	4	5	6	7	8
A	8	5	5	3	3	3	1	1
C	8	6	6	6	6	6	6	6
G	2	2	2	4	4	2	2	2
T	8	8	8	8	8	8	8	8

Tính rcGs[i], h[i] thông qua hmin, kmin, rmin

i	0	1	2	3	4	5	6	7	8
x[*i*]	G	C	A	G	A	G	A	G	
link[*i*]	-1	-1	-1	-1	0	2	3	4	
hmin[*i*]	0	7	3	7	5	5	7	6	7
kmin[*i*]	0	0	0	2	0	4	7	1	0
rmin[*i*]	7	7	7	7	7	7	7	8	0
rcGs[*i*]	0	2	4	7	7	7	7	7	7
h[*i*]		3	5	6	0	1	2	4	

Pha tìm kiếm

Lần thử thứ nhất

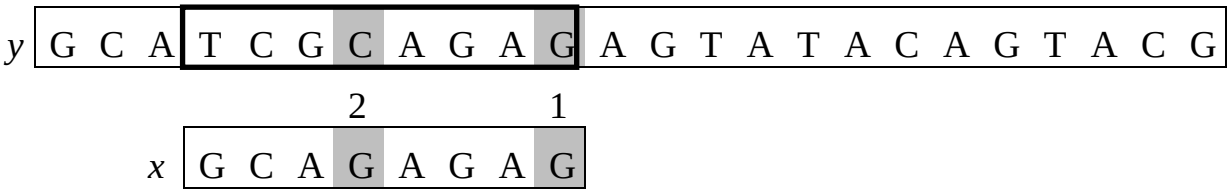
Lần thử thứ nhất

Dịch chuyển sang phải 1 bước (rcBc[A][8])

Lần thử thứ 2

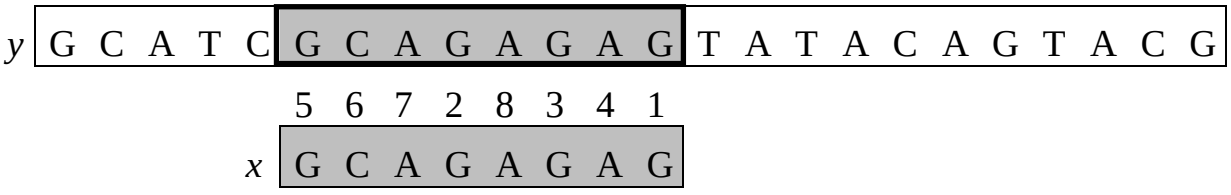
Dịch sang phải 2 bước (rcGs[1])

Lần thử thứ 3



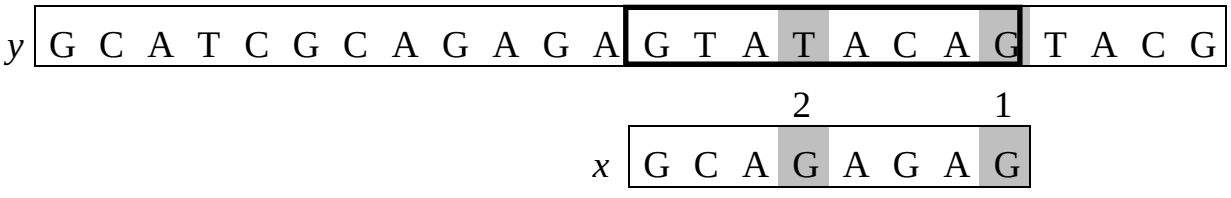
Dịch sang phải 2 bước (rcGs[1])

Lần thử thứ 4



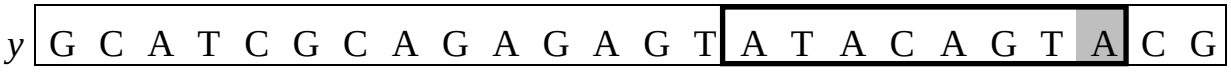
Dịch sang phải 7 bước (rcGs[8])

Lần thử thứ 5



Dịch sang phải 2 bước (rcGs[1])

Lần thử thứ 6



Dịch chuyển sang phải 5 bước (rcBc[A][2]) → vượt ra ngoài văn bản → kết thúc

2.5. Thuật toán Quick Search

a.Đặc điểm:

- Thuật toán là sự đơn giản hóa của thuật toán Boyer-Moore
- Sử dụng duy nhất bad-character shift
- Dế dàng thực hiện
- Pha tiền xử lý có độ phức tạp về thời gian là O(m+σ) và không gian O(σ);
- Pha tìm kiếm có độ phức tạp thuật toán là O(mn);

b.Mô tả thuật toán Quick Search

Input:

- Xâu mẫu X = (x_0, x_1,…, x_m), độ dài m.
- Văn bản Y = (y_1, y_2,…, y_n), độ dài n

Output:

- Đưa ra mọi vị trí xuất hiện của X trong Y.

Formats: Quick Search (x,m,qsBc);

Thuật toán Quick Search sử dụng duy nhất bad-character shift. Sau khi dịch chuyển cửa sổ trượt của xâu con trên xâu gốc y[j .. j+m-1] thì độ dài của dịch chuyển vị trí tối thiểu là 1.

bad-character shift của thuật toán có sự thay đổi nhỏ để đưa vào giá trị cuối cùng của X như sau : for c in Σ, *qsBc*[c]=*min*{i : 0 < i ≤ m and x[m-i]=c} if c occurs in x, m+1 otherwise

Trong suốt quá trình tìm kiếm xâu con với sự so sánh với xâu gốc, quá trình tìm kiếm sẽ không phân biệt theo thứ tự.

c.Code triển khai

```
void preQsBc(char *x, int m, int qsBc[]) {
   int i;

   for (i = 0; i < ASIZE; ++i)
      qsBc[i] = m + 1;
   for (i = 0; i < m; ++i)
      qsBc[x[i]] = m - i;
}
```

```
void QS(char *x, int m, char *y, int n) {
   int j, qsBc[ASIZE];

   /* Preprocessing */
   preQsBc(x, m, qsBc);

   /* Searching */
   j = 0;
   while (j <= n - m) {
      if (memcmp(x, y + j, m) == 0)
         OUTPUT(j);
      j += qsBc[y[j + m]];                /* shift */
   }
}
```

d.Kiểm nghiệm

Pha tiền xử lý :

a	A	C	G	T
$qsBc[a]$	2	7	1	9

Vòng lặp thứ 1

G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3 4

G C A G A G A G

Số bước dịch chuyển: 1 (*qsBc*[G])

Vòng lặp thứ 2

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Số bước dịch chuyển: 2 (*qsBc*[A])

Vòng lặp thứ 3

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Số bước dịch chuyển: 2 (*qsBc*[A])

Vòng lặp thứ 4

G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3 4 5 6 7 8

G C A G A G A G

Số bước dịch chuyển : 9 (*qsBc*[T])

Vòng lặp thứ 5

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Số bước dịch chuyển : 7 (*qsBc*[C])

2.6. Thuật toán Tuned Boyer-Moore :

Input :

- Xâu mẫu X =(x0, x1,..,xm), độ dài m.
- Văn bản Y =(y0, y1,..,xn), độ dài n.

Output:

- Tất cả vị trí xuất hiện X trong Y.

Formats:

TUNEDBM (char *x, int m, char *y, int n)

Actions:

```
void TUNEDBM(char *x, int m, char *y, int n) {
   int j, k, shift, bmBc[ASIZE];

   /* Preprocessing */
   preBmBc(x, m, bmBc);
   shift = bmBc[x[m - 1]];
   bmBc[x[m - 1]] = 0;
  memset(y + n, x[m - 1], m);

/* Searching */
   j = 0;
   while (j < n) {
      k = bmBc[y[j + m -1]];
      while (k !=  0) {
         j += k; k = bmBc[y[j + m -1]];
         j += k; k = bmBc[y[j + m -1]];
         j += k; k = bmBc[y[j + m -1]];
      }
      if (memcmp(x, y + j, m - 1) == 0 && j < n)
         OUTPUT(j);
      j += shift;                          /* shift */
   }
}
```

EndActions.

Kiểm nghiệm:

Y[]="ABABDABACDABAACABABE" n=20

X[]="ABAACABAB" m=9

Giai đoạn đầu

shift=BmBc[x[m - 1]]=BmBc[x[8]]= BmBc[B];

BmBc[x[m - 1]]=BmBc[x[8]]= BmBc[B]=0;

Có lại mảng BmBc[]

C	**B**	**A**	
4	**0**	**1**	**bmbc**

So sánh mảng x và y từ vị trí j với m-1phần tử và j>n?	k = bmBc[y[j + m -1]]	J+=k;	Nếu k!=0 lặp	k = bmBc[y[j + m - 1]]	J+=shift <n=20?
			Lặp	k=bmBc[Y[8]]= bmBc[C]=4	0(yes)
	k = bmBc[y[12]]= bmBc[A]=1	J=0+4 =4			
	k = bmBc[y[13]]= bmBc[B]=0	J=4+1 =5			
	k = bmBc[y[13]]= bmBc[B]=0	J=5+0 =5			
“ABAACABA” vs “ABACDABA” (no) J=5<n=20 (yes)			Thoát lặp		
	k = bmBc[y[16]]= bmBc[B]=0	J=7+1 =8	Lặp	k=bmBc[Y[15]]= bmBc[A]=1	7(yes)
	k = bmBc[y[16]]= bmBc[B]=0	J=8+0 =8			
	k = bmBc[y[16]]= bmBc[B]=0	J=8+0 =8			
“ABAACABA” vs “CDABAACA” (no)			Thoát lặp		

J=8<n=20 (yes)					
“ABAACABA” vs “ABAACABA” (yes) **J=10<n=20 (yes)**			Không lặp	k=bmBc[Y[18]]= bmBc[B]=0	10(yes)
	k = bmBc[y[29]]= bmBc[B]=0	J=12+ 9=21	Lặp	k=bmBc[Y[20]]= bmBc[B]=9	12(yes)
	k = bmBc[y[29]]= bmBc[B]=0	J=21+ 0=21			
	k = bmBc[y[29]]= bmBc[B]=0	J=21+ 0=21			
J=21>n=20 (yes)			Thoắt lặp		
					23(no)

Code:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
#include<iostream>
using namespace std;
#define ASIZE 256
void OUTPUT(int j){
     cout<<"\n\n "<<j;
}

void preBmBc(char *x, int m, int bmBc[]) {
     int i;
//  gán mang bmBc tu x[0]->x[m-1]  = i con lai la m
     for (i = 0; i < ASIZE; ++i)
      bmBc[i] = m;
     cout<<" \n";
     for (i = 0; i < m - 1; ++i){
     bmBc[x[i]] = m - i - 1;
          cout<<" x[i]: "<<x[i]<<" - bmBc[x[i]]:
"<<bmBc[x[i]]<<"\n";
```

```
     }
cout<<"\n ";
}
void TUNEDBM(char *x, int m, char *y, int n) {
   int j, k, shift, bmBc[ASIZE];
   /* Preprocessing */
   preBmBc(x, m, bmBc);
   shift = bmBc[x[m - 1]];
   bmBc[x[m - 1]] = 0;
   memset(y + n, x[m - 1], m);

     /* Searching */
   j = 0;
   while (j < n) {
      k = bmBc[y[j + m -1]];
      while (k !=  0) {
         j += k; k = bmBc[y[j + m -1]];
         j += k; k = bmBc[y[j + m -1]];
         j += k; k = bmBc[y[j + m -1]];
      }
      if (memcmp(x, y + j, m - 1) == 0 && j < n)
         OUTPUT(j);
      j += shift;                          /* shift */
   }
}

main()
{
     char *txt= "ABABDABACDABABCABABE";
    char *pat = "ABABCABAB";

     cout<<txt<< "      "<<strlen(txt);
     cout<<"\n"<<pat <<"      "<<strlen(pat);
     cout<<"\n";
```

```
    TUNEDBM(pat,strlen(pat), txt,strlen(txt));
    getch();
    return 0;
}
```

2.7. Thuật toán Zhu-Takaoka:

a.Đặc điểm:

- Biến thể của thuật toán Boyer - moore
- Sử dụng 2 ký tự văn bản liên tiếp để tính toán sự thay đổi ký tự xấu
- Giai đoạn tiền xử lý trong $0(m + a^2)$ thời gian và phức tạp không gian
- Tìm kiếm trong giai đoạn 0 (m x n) thời gian phức tạp

b.Mô tả:

Zhu and takaoka thiết kế một thuật toán thực hiện sự thay đổi bằng cách xem xét các thay đổi xấu ký tự (xem trang 14) cho 2 ký tự văn bản liên tiếp. Trong giai đoạn tìm kiếm so sánh được thực hiện từ phải sang trái và khi cửa sổ đặt trên nhân tố văn bản Y[j1] và một ghép đôi không xứng xảy ra giữa *x[m — k]* và *y[j + m — k]* khi đó *x[m — k + I . . m — 1] = y[j + m — k + I . . j + m — 1]* sự thay đổi là thực hiện với thay đổi ký tự xấu cho ký tự văn bản *y[j + m — 2]* và *y[j + m — 1]*. Bảng thay đổi good- sufEx cũng được dùng để tính toán sự thay đổi

Giai đoạn tiền xử lý của thuật toán bao gồm trong tính toán cho mỗi cặp ký tự (a,b) với a,b € ∑ xảy ra ở đầu bên phải của ab trong x[0..m-2]

For $a, b \in \Sigma$:

$$ztBc[a,b] = k \Leftrightarrow \begin{cases} k < m-2 & \text{and } x[m-k\,..\,m-k+1] = ab \\ & \text{and } ab \text{ does not occur} \\ & \text{in } x[m-k+2\,..\,m-2]\ , \\ \text{or} & \\ k = m-1 & x[0] = b \text{ and } ab \text{ does not occur} \\ & \text{in } x[0\,..\,m-2]\ , \\ \text{or} & \\ k = m & x[0] \neq b \text{ and } ab \text{ does not occur} \\ & \text{in } x[0\,..\,m-2]\ . \end{cases}$$

Nó cũng bao gồm việc tính toán bang bmGs. Giai đoạn tiền xử lý là trong *0(m + σ 2) thời gian và phức tạp không gian*

Giai đoạn tìm kiếm có một trường hợp xấu bậc 2

c.C code:

```
void preZtBc(char *x, int m, int ztBc[ASIZE][ASIZE]) {
    int i, j ;
    for (i = 0; i < ASIZE; ++i)
for (j = 0; j < ASIZE; ++j) ztBc [i] [j] = m;
    for (i = 0; i < ASIZE; ++i) ztBc [i] [x [0] ] = m - 1;
                for (i = 1; i < m - 1; ++i)
                        ztBc[x[i - l]][x[i]] = m - 1 - i;
    }
void ZT(char *x, int m, char *y, int n) {
  int i, j, ztBc[ASIZE] [ASIZE], bmGs[XSIZE];
  /* Preprocessing */ preZtBc(x, m,
  ztBc); preBmGs(x, m, bmGs);
  /* Searching */
  j = 0;
  while (j <= n - m) {
        i = m - 1;
        while (i < m && x[i] == y[i + j])
        --i;
    if (i < 0) {
      OUTPUT(j); j += bmGs[0];
    }
    else
      j += MAX(bmGs [i],
        ztBc [y [j + m - 2]][y[j + m - 1]]);
    }
```

d.Kiểm nghiệm thuật toán

ztBc	A	C	G	T
A	8	8	2	8
C	5	8	7	8
G	1	6	7	8
T	8	8	7	8

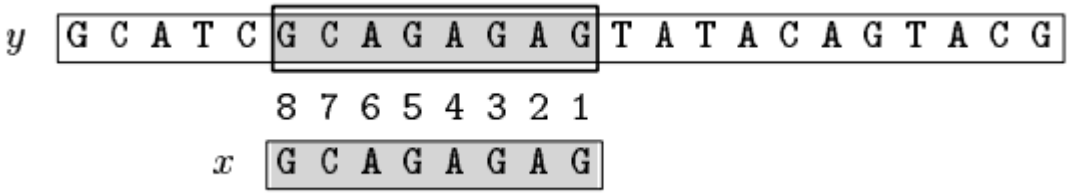
i	0	1	2	3	4	5	6	7
$x[i]$	G	C	A	G	A	G	A	G
$bmGs[i]$	7	7	7	2	7	4	7	1

Giai đoạn tìm kiếm:

Bước 1:

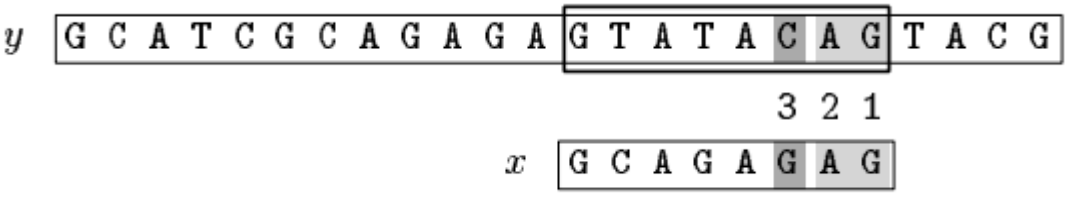
Dịch chuyển 5 (ztBc[C][A])

Bước 2



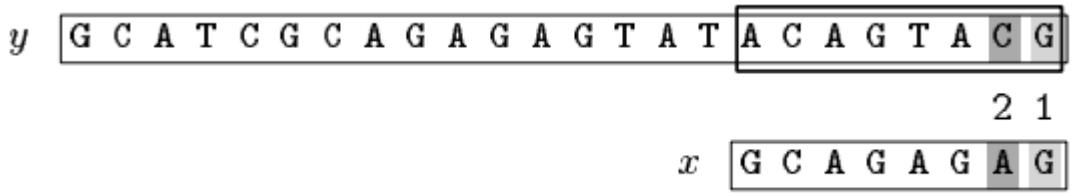
Dịch chuyển 7 (bmGs[0])

Bước 3



Dịch chuyển 4 (bmGs[6])

Bước 4



Dịch chuyển 7 *(bmGs[*7] = *zt.Bc*[C][G])

Các thuật toán Zhu-Takaoka thực hiện 14 so sánh nhân vật văn bản trên ví dụ.

2.8. Thuật toán Berry – Ravindran

a.Phát biểu thuật toán

- Sự pha trộn của thuật toán Quick Search và thuật toán Zhu-Takaoka
- Giai đoạn tiền xử lý độ phức tạp $O(m + \sigma^2)$
- Giai đoạn tìm kiếm độ phức tạp $O(m \times n)$

b.Mã hóa thuật toán

```
void preBrBc(char *x, int m, int
brBc[ASIZE][ASIZE]) {
   int a, b, i;

   for (a = 0; a < ASIZE; ++a)
      for (b = 0; b < ASIZE; ++b)
         brBc[a][b] = m + 2;
   for (a = 0; a < ASIZE; ++a)
      brBc[a][x[0]] = m + 1;
   for (i = 0; i < m - 1; ++i)
      brBc[x[i]][x[i + 1]] = m - i;
   for (a = 0; a < ASIZE; ++a)
      brBc[x[m - 1]][a] = 1;
}

void BR(char *x, int m, char *y, int n) {
   int j, brBc[ASIZE][ASIZE];

   /* Preprocessing */
   preBrBc(x, m, brBc);

   /* Searching */
   y[n + 1] = '\0';
   j = 0;
   while (j <= n - m) {
```

```
            if (memcmp(x, y + j, m) == 0)
                OUTPUT(j);
            j += brBc[y[j + m]][y[j + m + 1]];
        }
    }
```

c.Kiểm nghiệm

brBc	A	C	G	T	*
A	10	10	2	10	10
C	7	10	9	10	10
G	1	1	1	1	1
T	10	10	9	10	10
*	10	10	9	10	10

Bảng ký tự tồi. Ký tự (*) biểu thị cho các ký tự còn lại

Bước 1:

G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3 4

G C A G A G A G

Dịch: 1 (*brBc*[G][A])

Bước 2:

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch: 2 (*brBc*[A][G])

Bước 3:

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch: 2 (*brBc*[A][G])

Bước 4:

G C A T C G C A G A G A G T A T A C A G T A C G

1 2 3 4 5 6 7 8

G C A G A G A G

Dịch: 10 (*brBc*[T][A])

Bước 5:

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch: 7 (*brBc*[G][0])

Bước 6:

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch: 10 (*brBc*[0][0]). Kết thúc thuật toán

3. Thuật toán tìm kiếm mẫu từ vị trí cụ thể

3.1. Thuật toán Galil-Giancarlo

a.Các đặc điểm chính

- Cần hoàn chỉnh các thuật toán Colussi;
- Pha tiền xử lý có độ phức tạp không gian và thời gian là O(m);
- Pha tìm kiếm có độ phức tạp thời gian là O(n);
- Trong trường hợp xấu nhất phải thực hiện $\frac{4}{3}n$ so sánh ký tự văn bản

b.Mô tả thuật toán

Các thuật toán Galil-Giancarlo là một biến thể của thuật toán Colussi (xem chương 9). Sự thay đổi can thiệp vào pha tìm kiếm. Phương pháp này được áp dụng khi x không phải là một lũy thừa của một ký tự đơn. Như vậy x ≠ c^m với c € Σ. Lấy ℓ là chỉ số cuối cùng trong patern sao cho 0 ≤ i ≤ ℓ, x [0] = [i] x và x [0] ≠ x [ℓ + 1]. Giả sử trong lần thử trước đó tất cả các noholes đã được so khớp và một hậu tố của pattern đã được so khớp nghĩa là sau sự thay đổi tương ứng một tiền tố của pattern sẽ vẫn khớp với một phần của văn bản. Do đó các ô được đặt ở vị trí trên các nhân tố văn bản *y*[j ... j + m -1] và phân ra *y*[j ... cuối cùng] khớp với x [0 ... cuối cùng - j]. Sau đó lần thử tiếp theo thuật toán sẽ quét các ký tự văn bản bắt đầu với y [last + 1] cho đến khi hoặc kết thúc của văn bản là đạt hoặc một ký tự x[0] ≠ y [j + k] được tìm thấy. Trong trường hợp sau này hai subcases có thể phát sinh:

x [ℓ + 1] ≠ y [j + k] hoặc quá ít x [0] đã được tìm thấy (k ≤ ℓ) sau đó các ô được chuyển và định vị trên các nhân tố văn bản y [k + 1 ... k + m), quá trình quét của văn bản được khôi phục lại (như trong thuật toán Colussi) với nohole đầu tiên và tiền tố ghi nhớ của pattern là những từ rỗng.

x [ℓ + 1] = y [j + k] và đầy đủ của x [0] đã được tìm thấy (k> ℓ) sau đó các ô được chuyển và định vị trên các nhân tố văn bản y [k - ℓ -1 ... k - ℓ + m - 2], quá trình quét của văn bản được khôi phục lại (như trong thuật toán Colussi) với nohole thứ hai (x [ℓ + 1] là đầu tiên) và tiền tố ghi nhớ của pattern là x [0 ... ℓ + 1].

Pha tiền xử lý là chính xác giống như trong thuật toán Colussi (chương 9) và có thể được thực hiện trong O(m) không gian và thời gian. Pha tìm kiếm sau đó có thể thực hiện độ phức tạp về thời gian O(n) và hơn nữa tối đa là 4/3n so sánh ký tự văn bản được thực hiện trong giai đoạn tìm kiếm.

c.Mã hóa thuật toán

```
void GG(char *x, int m, char *y, int n) {
int i, j, k, ell, last, nd;
int h[XSIZE], next[XSIZE], shift[XSIZE];
```

```
char heavy;

for (ell = 0; x[ell] == x[ell + 1]; ell++);
if (ell == m - 1)
        /* Tìm kiếm lũy thừa của một ký tự đơn */
        for (j = ell = 0); j < n; ++j)
            if (x[0] == y[j] {
                ++ell;
                if (ell >= m)
                    OUTPUT (j - m + 1);
            }
            else
                ell = 0;
else {
        /* Tiền xử lý */
        nd = preCOLUSSI (x, m, h, next, shift);

        /* Tìm kiếm */
        i = j = heavy = 0;
        last = -1;
        while (j <= n - m) {
            if (heavy && I == 0) {
                k = last - j + 1;
                while (x[0] == y[j + k])
                    k++;
                if (k <= ell || x[ell + 1] != y[j +
k]) {
                    i = 0;
                    j += (k + 1);
                    last = j - 1;
                }
                else {
```

```
                    i = 1;
                    last = j – k; j= last - (ell +
1);
                }
                heavy = 0;
            }
else {
                while (i < m && last < j + h[i] &&
x[h[i]] == y[j + h[i]])
                    ++i;
                if (I >= m || last >= j + h[i]) {
                    OUTPUT(j);
                    i = m;
                }
if (i > nd)
                    last = j + m – n;
j += shift [i];
i = next[i];
}
Heavy = (j > last ? 0 : 1);
            }
        }
}
```

d.Kiểm nghiệm thuật toán

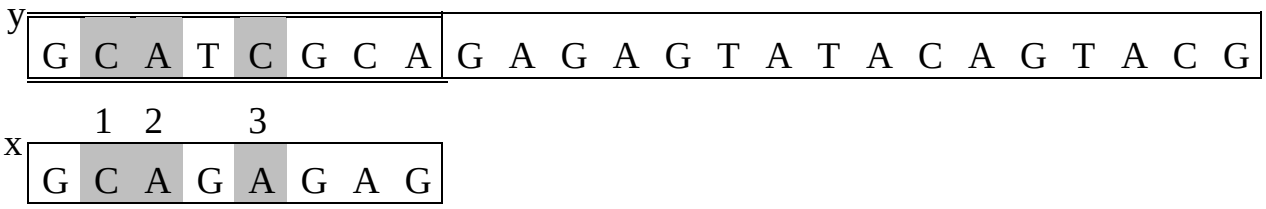
i	0	1	2	3	4	5	6	7	8
x[i]	G	C	A	G	A	G	A	G	
kmpNext[i]	-1	0	0	-1	1	-1	1	-1	1
kmin[i]	0	1	2	0	3	0	5	0	
h[i]	1	2	4	6	7	5	3	0	
next[i]	0	0	0	0	0	0	0	0	0
shift[i]	1	2	3	5	8	7	7	7	7

hmax[i]	0	1	2	4	4	6	6	8	8
rmin[i]	7	0	0	7	0	7	0	8	
ndh0[i]	0	0	1	2	2	3	3	4	

nd = 3 and l = 0

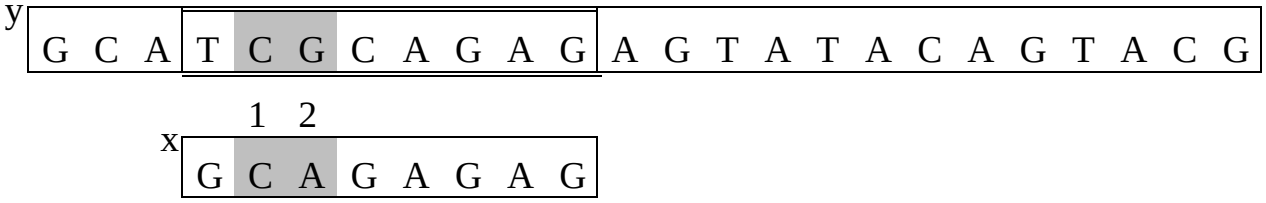
Pha tìm kiếm

Lần thử đầu tiên:



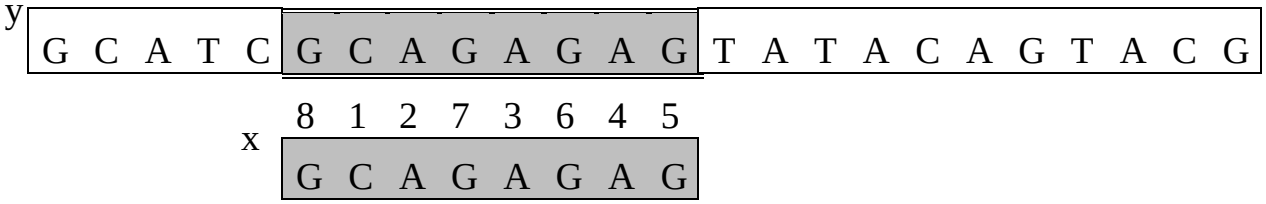
Shift by 3 (shift[2])

Lần thử thứ 2:



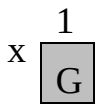
Shift by 2 (shift[1])

Lần thử thứ 3:



Shift by 7 (shift[8])

Lần thử thứ 4:



Shift by 2

Lần thử thứ 5:

Shift by 1 (shift[0])

Lần thử thứ 6:

Shift by 1 (shift[0])

Lần thử thứ 7:

Shift by 3 (shift[2])

Các thuật toán Galil-Giancarlo thực hiện 19 so sánh ký tự văn bản trên ví dụ.

3.2. Thuật toán Skip Search

a.Mô tả thuật toán

- Sử dụng nhóm các vị trí cho mỗi ký tự trong bảng
- Gồm ***O***(*m*+σ) bươcs
- Độ phức tạp thuật toán : ***O***(*mn*);
- Có ***O***(*n*) bước so sánh

b.Cài đặt thuật toán

```
void SKIP(char *x, int m, char *y, int n) {
   int i, j;
   List ptr, z[ASIZE];

   /* Preprocessing */
   memset(z, NULL, ASIZE*sizeof(List));
   for (i = 0; i < m; ++i) {
      ptr = (List)malloc(sizeof(struct _cell));
      if (ptr == NULL)
         error("SKIP");
      ptr->element = i;
      ptr->next = z[x[i]];
      z[x[i]] = ptr;
   }

   /* Searching */
   for (j = m - 1; j < n; j += m)
      for (ptr = z[y[j]]; ptr != NULL; ptr = ptr->next)
         if (memcmp(x, y + j - ptr->element, m) == 0) {
            if (j - ptr->element <= n - m)
               OUTPUT(j - ptr->element);
         }
         else
            break;
}
```

c.Kiểm nghiệm thuật toán

Lần đầu

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
	2						1																
	G	C	A	G	A	G	A	G															

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
			1				-																
			G	C	A	G	A	G	A	G													

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
					1	2	3	4	5	6	7	8											
					G	C	A	G	A	G	A	G											

Dịch : 8

Lần 2

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
													1										

Dịch : 8

Lần 3

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
																2							1
																G	C	A	G	A	G	A	G

Thuật toán tiến hành so sánh 14 ký tự trong ví dụ này

4. Thuật toán tìm kiếm mẫu từ bất kỳ

4.1. Thuật toán Horspool

a.Phát biểu thuật toán

- Là dạng đơn giản của thuật toán Boyer – Moore
- Chỉ dùng để thay đổi các ký tự xấu
- Dễ thực hiện
- Giai đoạn tiền xử lý độ phức tạp $O(m+\sigma)$ thời gian và $O(\sigma)$ không gian
- Giai đoạn tìm kiếm phức tạp $O(m \times n)$ thời gian
- Chỉ số trung bình của sự so sánh cho 1 ký tự text là giữa $1/\sigma$ và $2/(\sigma+1)$

b.Mã hóa thuật toán

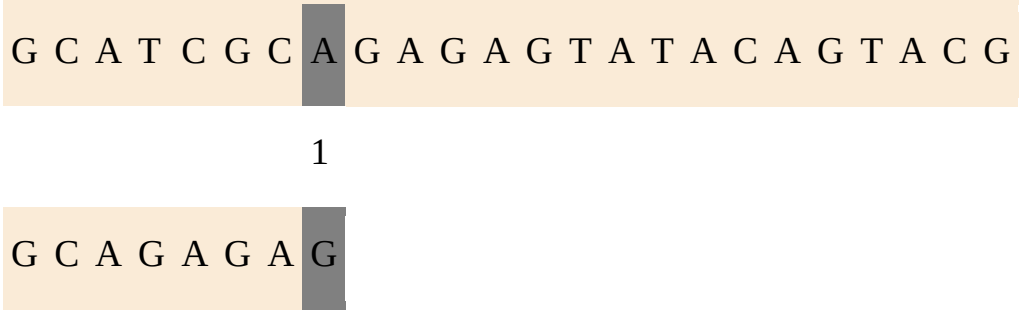
```
void HORSPOOL(char *x, int m, char *y, int n) {
   int j, bmBc[ASIZE];
   char c;

   /* Preprocessing */
   preBmBc(x, m, bmBc);
   /* Searching */
   j = 0;
   while (j <= n - m) {
      c = y[j + m - 1];
      if (x[m - 1] == c && memcmp(x, y + j, m - 1) ==
0)
         OUTPUT(j);
      j += bmBc[c];
   }
}
```

c.Kiểm nghiệm

a	A	C	G	T
$bmBc[a]$	1	6	2	8

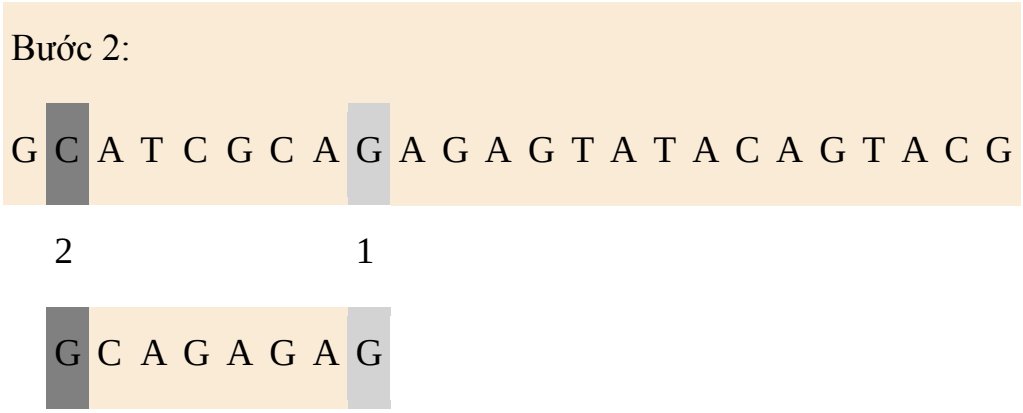
Bảng ký tự tồi

Bước 1:



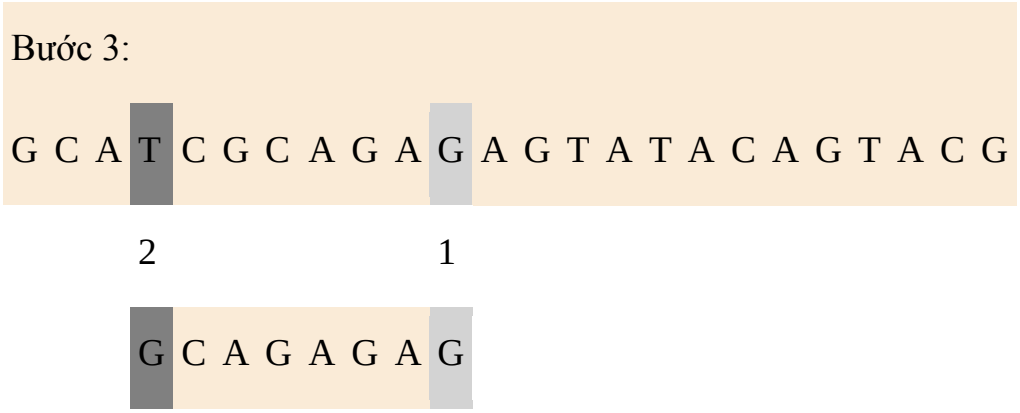
Dịch : 1 (*bmBc*[A])

Bước 2:



Dịch: 2 (*bmBc*[G])

Bước 3:



Dịch: 2 (*bmBc*[G])

Bước 4:

G C A G A G A G

Dịch: 2 (*bmBc*[G])

Bước 5:

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch: 1 (*bmBc*[A])

Bước 6:

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Dịch: 8 (*bmBc*[T])

Bước 7:

G C A T C G C A G A G A G T A T A C A G T A C G

2 1

G C A G A G A G

Dịch: 2 (*bmBc*[G])

Kết thúc thuật toán.

4.2. Thuật toán Raita

a.Các tính năng chính

- Trước tiên so sánh các ký tự của các mẫu cuối cùng, sau đó là mẫu đầu tiên và cuối cùng là một trong những so sánh khác nhau ở hiện tại.
- Thực hiện các thay đổi như ở thuật toán Horspool (Trang 117)
- Thời gian đưa vào từng bước xử lý trước là O(m + a) và độ phức tạp không gian O(a)
- Độ phức tạp của thời gian tìm kiếm là O(mxn).

b.Mô tả

Raita thiết kế một thuật toán mà tại mỗi nỗ lực đầu tiên so sánh các ký tự cuối cùng của mô hình với các ký tự bên phải của cửa sổ, sau đó nếu chúng phù hợp với các so sánh các ký tự đầu tiên của mô hình với các ký tự văn bản ngoài cùng bên trái của cửa sổ, rồi nếu chúng phù hợp với so sánh ký tự văn bản giữa các mô hình với giữa các ký tự của cửa sổ. Và cuối cùng của chúng phù hợp với các ký tự của mô hình khác từ thứ hai đến cuối, nhưng một, có thể so sánh một lần nữa giữa các ký tự này.

Raita quan sát thấy rằng thuật toán của nó đã có một tác động tốt trong thực tế khi tìm kiếm mô hình trong các văn bản tiếng Anh và các tác động đến sự tồn tại của các ký tự phụ thuộc. Smith thực hiện một số thí nghiệm và kết luận rằng hiện tượng này có thể thay thế bỡi các kết quả của trình biên dịch khác.

Giai đoạn tiền xử lý của thuật toán Raita bao gồm trong tính toán các chức năng chuyển đổi xấu ký tự (xem chương 14). Nó có thể được thực hiện trong thời gian O (m + a) và sự phức tạp không gian O (a)

Giai đoạn tìm kiếm của thuật toán Raita có một trường hợp xấu là nhất bậc hai về sự phức tạp thời gian.

c.Mã chương trình bằng ngôn ngữ C

```
Void RAITA(char *x, int m, char *y, int m) {
Int j, bmBc[ASIZE];
Char c, firstCh, *SecondCh, middleCh, lastCh;

/*preprocessing*/
preBmBc(x, m, bmBc);
firstCh = x[0];
secondCh = x + 1
```

```
middleCh = x[m/2];
lastCh = x[m-1];

/*Searching*/
J = 0;
While (j <= n - m){
C = y[ j + m - 1 ];
If (lastCh == c && firstCh == y[j]  &&middleCh ==
y[ j + m/2 ] && memcmp (secondCh, y + j +1, m -
2) == 0
OUTPUT(j);
J += bmBc[c];
}
}
```

d.Kiểm nghiệm thuật toán

a	A	C	G	T
bmBc[a]	1	6	2	8

Searching phase

First attempt

y	G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
								1																
x	G	C	A	G	A	G	A	G																

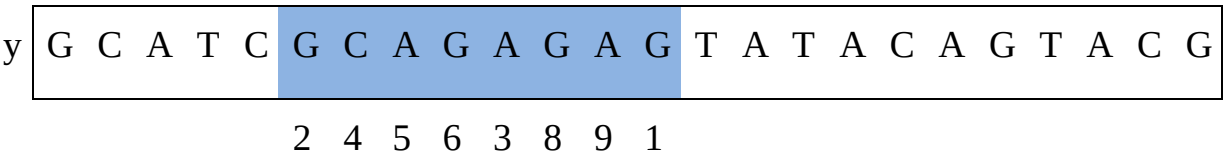
Shift by 1 (bmBc[a])

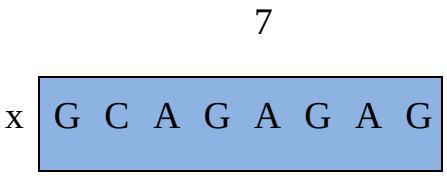
Second attempt

y	G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G

Shift by 2 (bmBc[G])

Fourth attempt





Shirt by 2 (bmBc[G])

Firth attempt

Sifth by 1 (bmBc[A])

Sixth attempt

Shift by 8 (bmBc[T])

Seventh attempt:

Shift by 2 (bmBc[G])

Thuật toán Raita thực hiện 18 so sánh đặc tính văn bản trên ví dụ.

4.3. Thuật toán String Matching on Ordered

a.Đặc điểm

- Không có giai đoạn tiền xử lý;
- Đòi hỏi phải có một bảng chữ cái đúng;
- Độ phức tạp trong giai đoạn tìm kiếm O (n);
- Thực hiện 6n + 5 so sánh đối tượng văn bản trong trường hợp xấu nhất.

b.Thuật toán

Giải thuật

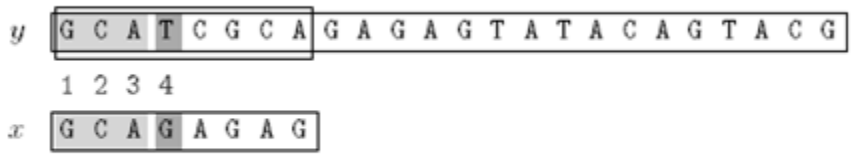
```
/* Compute the next maximal suffix. */
void nextMaximalSuffix(char *x, int m,
                       int *i, int *j, int *k, int *p) {
   char a, b;

   while (*j + *k < m) {
      a = x[*i + *k];
      b = x[*j + *k];
      if (a == b)
         if (*k == *p) {
            (*j) += *p;
            *k = 1;
         }
         else
            ++(*k);
      else
         if (a > b) {
            (*j) += *k;
            *k = 1;
```

```
            *p = *j - *i;
         }
         else {
            *i = *j;
            ++(*j);
            *k = *p = 1;
         }
   }
}


/* String matching on ordered alphabets algorithm. */
void SMOA(char *x, int m, char *y, int n) {
   int i, ip, j, jp, k, p;

   /* Searching */
   ip = -1;
   i = j = jp = 0;
   k = p = 1;
   while (j <= n - m) {
      while (i + j < n && i < m && x[i] == y[i + j])
         ++i;
      if (i == 0) {
         ++j;
         ip = -1;
         jp = 0;
         k = p = 1;
      }
      else {
         if (i >= m)
            OUTPUT(j);
         nextMaximalSuffix(y + j, i+1, &ip, &jp, &k, &p);
         if (ip < 0 ||
             (ip < p &&
              memcmp(y + j, y + j + p, ip + 1) == 0)) {
```

```
            j += p;
            i -= p;
            if (i < 0)
               i = 0;
            if (jp - ip > p)
               jp -= p;
            else {
               ip = -1;
               jp = 0;
               k = p = 1;
            }
         }
         else {
            j += (MAX(ip + 1,
                      MIN(i - ip - 1, jp + 1)) + 1);
            i = jp = 0;
            ip = -1;
            k = p = 1;
         }
      }
   }
}
```

c.Kiểm nghiệm

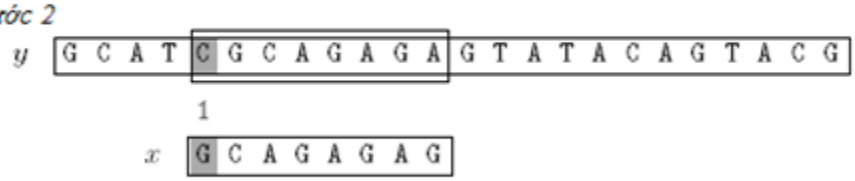
Bước 1



Sau khi gọi hàm `nextMaximalSuffix`: $\texttt{ip} = 2, \texttt{jp} = 3, \texttt{k} = 1, \texttt{p} = 1.$
Nó thực hiện so sánh 6 kí tự chữ.

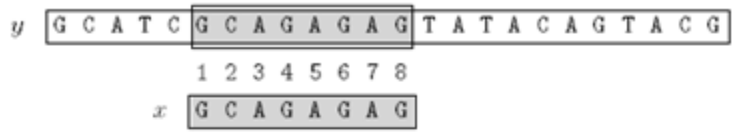
Chuyển thành 4

Bước 2



Chuyển thành 1

Bước 3



Sau khi gọi hàm `nextMaximalSuffix`: $\texttt{ip} = 7, \texttt{jp} = 8, \texttt{k} = 1, \texttt{p} = 1$
Nó thực hiện so sánh 1 kĩ tự chữ

Chuyển thành 9

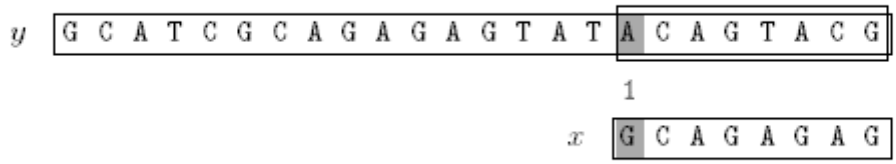
Bước 4

Chuyển thành 1

Bước 5

Chuyển thành 1

Bước 6



Chuyển thành 1

Sau khi thực hiện. Thuật toán so sánh 15 kí tự chữ so khớp xâu theo thứ tự bảng chữ cái