HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

Khoa Công nghệ thông tin

*****📖*****

Báo cáo môn học

Chuyên đề công nghệ phần mềm

Giảng viên : TS. Nguyễn Duy Phương

Sinh viên : Nguyễn Đình Tiến

Mã sinh viên : B16DCCN353

Nhóm : 01

Hà Nội, 07/2020

Mục lục

1. Tổng quan
   - Nhóm 1
     - Thuật toán Brute-Force
     - Thuật toán Karp-Rabin
     - Thuật toán Morris-Pratt
     - Thuật toán Knuth-Morris-Pratt
     - Thuật toán Not So Naive
   - Nhóm 2
     - Thuật toán Boyer-Moore
     - Thuật toán Berry – Ravindran
     - Thuật toán Turbo Boyer-Moore
   - Nhóm 3
     - Thuật toán Colussi
     - Thuật toán Skip Search
   - Nhóm 4
     - Thuật toán Horspool
     - Thuật toán Quick Search
     - Thuật toán Raita
2. Tìm kiếm mẫu từ trái sang phải

   2.1 Thuật toán Brute-Force
   - Trình bày thuật toán
     - Thuật toán Brute Force kiểm tra ở tất cả các vị trí trong văn bản giữa 0 và n-m, bất kể có xuất hiện một pattern hay không. Mỗi bước so khớp dịch một vị trí sang phải.
     - 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(m.n).
   - Đặc điểm
     - Không có giai đoạn tiền xử lý
     - Cần thêm không gian liên tục
     - Luôn di chuyển của sổ so sanh 1 vị trí sang phải
     - So sánh có thể hoàn thành trong thứ tự bất kỳ
     - Độ phức tạp thời gian trong giai đoạn tìm kiếm là O(m.n)
     - So sánh khoảng 2n ký tự
   - Kiểm nghiệm

First attempt

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

Second attempt

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

Third attempt

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

Fourth attempt

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

Fifth attempt

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

Sixth attempt

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

Seventh attempt

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

Eighth attempt

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

Ninth attempt

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

Tenth attempt

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

Eleventh attempt

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

Twelfth attempt

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

Thirteenth attempt

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

Fourteenth attempt

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

Fifteenth attempt

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

Sixteenth attempt

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

Seventeenth attempt

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

- Lập trình

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);

```
void BruteForce(char *x, int m, char *y, int n) {
  int i, j;
  /* Searching */
  for (j = 0; j <= n - m; ++j) {
    for (i = 0; i < m && x[i] == y[i + j]; ++i);
    if (i >= m)
      cout<<(j); // Output
  }
}
```

2.2 Thuật toán Karp-Rabin

- Trình bày thuật toán
  - Hàm băm cung cấp phương pháp đơn giản để tránh việc so sánh ký tự trùng nhau trong tình huống thực tế nhất. Thay vì kiểm tra tại mỗi vị trí của văn bản nếu có pattern, cách hiệu quả hơn để kiểm tra là chỉ khi các nội dung nội dung của cửa sổ trông giống như pattern. Để kiểm tra sự giống nhau giữa 2 đoạn này, hàm băm được sử dụng.
  - Ta có hàm băm :

$$hash(w[0\ ..\ m\text{-}1]) = (w[0]*2^{m-1} + w[1]*2^{m-2} + \cdots + w[m\text{-}1]*2^{0}) \bmod q$$

$$rehash(a,b,h) = ((h - a*2^{m-1})*2 + b) \bmod q$$

- Pha tiền xử lý có độ phức tạp O(m). Tuy nhiên thời gian tìm kiếm lại tỷ lệ với O(m.n) vì có nhiều trường hợp hàm băm bị lừa và không phát huy tác dụng. Tuy nhiên đó là trường hợp đặc biệt, trong thực tế thời gian tính toán thường tỉ lệ thuận với O(m+n). Hơn nữa thuật toán Karp Rabin có thể dễ dàng mở rộng cho các mẫu, văn bản dạng 2 chiều, hữu ích cho các thuật toán xử lý ảnh.

- Đặc điểm
  - Sử dụng hàm băm
  - Giai đoạn tiền xử lý có độ phức tạp thời gian O(m) và độ phức tạp không gian hằng số
  - Giai đoạn tìm kiếm có độ phức tạp thời gian O(m.n)
  - Thời gian chạy dự đoán là O(m+n)
- Kiểm nghiệm

First attempt

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

G C A G A G A G

hash(*y*[0 .. 7]) = 17819

Second attempt

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

G C A G A G A G

hash(*y*[1 .. 8]) = 17533

Third attempt

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

G C A G A G A G

hash(*y*[2 .. 9]) = 17979

Fourth attempt

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

G C A G A G A G

hash(*y*[3 .. 10]) = 19389

Fifth attempt

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

G C A G A G A G

hash(*y*[4 .. 11]) = 17339

Sixth attempt

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

hash(*y*[5 .. 12]) = 17597

Seventh attempt

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

G C A G A G A G

hash(*y*[6 .. 13]) = 17102

Eighth attempt

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

G C A G A G A G

hash(*y*[7 .. 14]) = 17117

Ninth attempt

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

G C A G A G A G

hash(*y*[8 .. 15]) = 17678

Tenth attempt

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

G C A G A G A G

hash(*y*[9 .. 16]) = 17245

Eleventh attempt

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

G C A G A G A G

hash(*y*[10 .. 17]) = 17917

Twelfth attempt

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

G C A G A G A G

hash(*y*[11 .. 18]) = 17723

Thirteenth attempt

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

G C A G A G A G

hash(*y*[12 .. 19]) = 18877

Fourteenth attempt

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

G C A G A G A G

hash(*y*[13 .. 20]) = 19662

Fifteenth attempt

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

G C A G A G A G

hash(*y*[14 .. 21]) = 17885

Sixteenth attempt

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

G C A G A G A G

hash(*y*[15 .. 22]) = 19197

Seventeenth attempt

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

G C A G A G A G

hash(*y*[16 .. 23]) = 16961

- Lập trình

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.

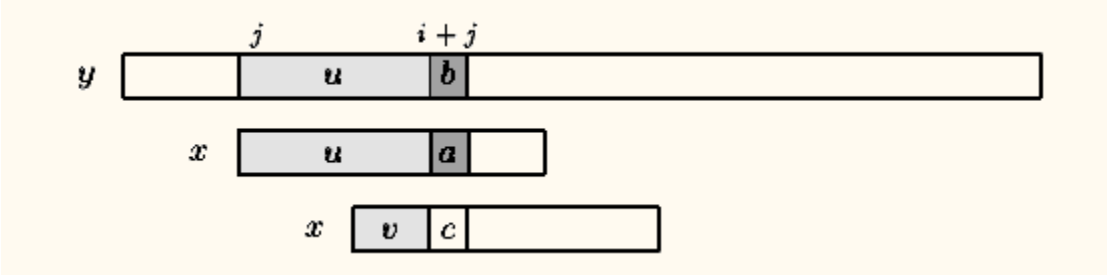
Formats:

```
Brute-Force(X, m, Y, n);


#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)
                        cout<<(j);
                hy = REHASH(y[j], y[j + m], hy);
                ++j;
        }
}
```

2.3 Thuật toán Morris-Pratt

- Trình bày thuật toán
  - Thuật toán MP cải tiến thuật toán Brute Force, thay vì dịch chuyển từng bước một, phí công các ký tự đã so sánh trước đó, ta tìm cách dịch x đi một đoạn xa hơn.



Hình 3.1 : Dịch chuyển trong thuật toán Morris-Pratt

- Giả sử tại bước so sánh bất kỳ, ta có một pattern "*u*" trùng nhau giữa x và y, tại x[i] != y[j+i] (*a* != *b*), thay vì dịch chuyển 1 bước sang phải, ta cố gắng dịch chuyển dài hơn sao cho một tiền tố (prefix) *v* của x trùng với hậu tố (suffix) của u.
- Ta có mảng mpNext[] để tính trước độ dài trùng nhau lớn nhất giữa tiền tố và hậu tố trong x, khi so sánh với y tại vị trí thứ i, x sẽ trượt một khoảng = i – mpNext[i].
- Việc tính toán mảng mpNext[] có độ phức tạp thời gian và không gian là O(n). Giai đoạn tìm kiếm sau đó có độ phức tạp thời gian là O(m+n).

- Đặc điểm
  - Thực hiện việc so sanh từ trái qua phải
  - Pha tiền xử lý có độ phức tạp không gian và thời gian là O(m)
  - Pha tiền xử lý có độ phức tạp thời gian là O(m+n)
  - Thực thi 2n-1 thông tin thu thập được trong quá trình quét văn bản
  - Độ trễ m (số lượng tối đa các lần so sánh ký tự đơn)
- Kiểm nghiệm
  - Kiểm nghiệm pha tiền xử lý(thuật toán preMp)
  - x[] = GCAGAGAG

ghi chú	**mpNext[i]**	**j**	**i**	**x[i]**	**x[j]**
		-1	0		
=>mpNext[1] =0	-1	-1	0	G	
	0	0 -1	1	C	G
	0	0 -1	2	A	G

	0	0	3	G	G
	1	1 0 -1	4	A A	C G
	0	0	5	G	G
	1	1 0 -1	6	A A	C G
	0	0	7	G	G
	1	1	8		

Ta được bảng mpNext[]

i	0	1	2	3	4	5	6	7	8
x[i]	G	C	A	G	A	G	A	G	
mpNext[i]	-1	0	0	0	1	0	1	0	1

First attempt

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

Shift by: 3 (*i*-*mpNext*[*i*]=3-0)

Second attempt

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

Shift by: 1 (*i-mpNext*[*i*]=0- -1)

Third attempt

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

Shift by: 1 (*i-mpNext*[*i*]=0- -1)

Fourth attempt

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

Shift by: 7 (*i-mpNext*[*i*]=8-1)

Fifth attempt

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

Shift by: 1 (*i-mpNext*[*i*]=1-0)

Sixth attempt

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

Shift by: 1 (*i-mpNext*[*i*]=0- -1)

Seventh attempt

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

Shift by: 1 (*i-mpNext*[*i*]=0- -1)

Eighth attempt

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

Shift by: 1 (*i-mpNext*[*i*]=0- -1)

Ninth attempt

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

Shift by: 1 (*i-mpNext*[*i*]=0- -1)

- Lập trình

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <iostream>
using namespace std;
#define MAX 12
int mpNext[MAX];
void Init()
{
	for(int i = 0; i < MAX; i++)
		mpNext[i] = 999;
}
void preMp(char *x, int m) {
	int i, j;
	i = 0;			//mang mpNext the hien do dai trung nhau lon
	j = mpNext[0] = -1;		//nhat giua tien to va hau to
	while (i < m) {
		while (j > -1 && x[i] != x[j])
		{
			j = mpNext[j];  //chay nguoc xet xem do dai lon nhat cua
			//vi tri giong voi x[i]
		}
		i++;
		j++;
		mpNext[i] = j;
		int a = 2;
	}
}
void MP(char *x, int m, char *y, int n) {
	int i, j;// mpNext[m];
	//int mpNext[8];
	/* Preprocessing */
```

```
        Init();
        preMp(x, m);
        for(int k =0;k<m;k++){
            cout<<x[k]<<" "<<mpNext[k]<<endl;
        }
        /* Searching */
        i = j = 0;
        while (j < n) {
            while (i > -1 && x[i] != y[j])
                i = mpNext[i];
            i++;
            j++;
            if (i >= m) {
                cout<<j - i;
                i = mpNext[i];
            }
        }
    }
    void main()
    {
        char *x = "GCAGAGAG";           //"ATCACATCATCA ";
        int m = strlen(x);
        char *y = "GCATCGCAGAGAGTATACAGTACG";
    //"AGTATCATCACATCATCAGA";
        int n = strlen(y);
        MP(x, m, y, n);
    }
```

2.4 Thuật toán Knuth-Morris-Pratt

- Trình bày thuật toán

- Knuth Morris Partt là một phát triển chặt chẽ hơn của thuật toán Morris partt. Hãy nhìn lại Morris Partt, nó có thể tăng được số bước dịch chuyển. Quan tâm tới bên trái vị trí j trong y khi cửa sổ đang là vị trí của đoạn y[j…j+m-1]. Giả sử rằng vị trí đầu tiên mà chúng không giống nhau là x[i] và y[i+j] với 0<j<m. và ***x***[0..***i***-1] = ***y***[***j*** .. ***i***+***j***-1] = ***u*** and ***a*** = x[***i***] ***y***[***i***+***j***]=***b***.Group TTV Page 31 Khi dịch, nó là cơ sở để chắc chắn rằng tiền tố v của mẫu trùng với một hậu tố của u trong văn bản. Tuy nhiên, nếu chúng ta muốn tránh những mẫu không khớp ngay lập tức, kí tự ngay sau tiền tố v trong mẫu phải khác a. Độ dài lớn nhất của tiền tố v được gọi là nhãn biên của u(nó là kí tự xuất hiện ở 2 đầu của u theo sau bởi kí tự khác nhau trong x). kmpNext[i] là độ dài dài nhất của biên x[i…i-1] nếu theo sau là kí tự c khác với x[i] và bằng -1 nếu nhãn biên tồn tại với 0<i<=m. Bên cạnh đó sau mỗi bước dịch, ta tiếp tục so sánh kí tự c= x[mpNext[i]] và y[i+j]= b trong trường hợp không tồn tại kí tự nào của x trong y và tránh bị backtrack. Giá trị của kmpNext[0] được thiết lập bằng -1.

- Đặc điểm
  - Thực hiện so sánh từ trái sang phải
  - 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 O(m+n)
  - Độ trễ log**Φ**(*m*) với **Φ** là tỷ lệ vàng ($\Phi = \frac{1+\sqrt{(5)}}{2}$)
- Kiểm nghiệm
  - Kiểm nghiệm pha tiền xử lý (thuật toán preKMP)
  - x[] = GCAGAGAG

ghi chú	mpNext[i]	j	i	x[i]	x[j]
		-1	0		
=>mpNext[1] =0	-1	-1	0	G	\0
	0	0 -1	1	C	G
	0	0 -1	2	A	G
kmpNext[i] = kmpNext[j];	-1	0	3	G	G

	1	1 0 -1	4	A A	C G
kmpNext[i] = kmpNext[j];	-1	0	5	G	G
	1	1 0 -1	6	A A	C G
kmpNext[i] = kmpNext[j];	-1	0	7	G	G
	1	1	8	\0	C

Ta được bảng kmpNext[]

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	

First attempt

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

Shift by: 4 (*i*-*kmpNext*[*i*]=3- -1)

Second attempt

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

Shift by: 1 (*i-kmpNext*[*i*]=0- -1)

Third attempt

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											

Shift by: 7 (*i-kmpNext*[*i*]=8-1)

Fourth attempt

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			

Shift by: 1 (*i-kmpNext*[*i*]=1-0)

Fifth attempt

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			

Shift by: 1 (*i-kmpNext*[*i*]=0- -1)

Sixth attempt

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		

Shift by: 1 (*i-kmpNext*[*i*]=0- -1)

Seventh attempt

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

Shift by: 1 (*i-kmpNext*[*i*]=0- -1)

Eighth attempt

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

Shift by: 1 (*i-kmpNext*[*i*]=0- -1)

- Lập trình

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
using namespace std;
#define MAX 9
int kmpNext[MAX + 1];
void Init()
{
        for(int i = 0; i < MAX + 1; i++)
                kmpNext[i] = 999;
}
```

```
void preKMP(char *x, int m)
{
	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;
	Init();
	/*Preprocessing*/
	preKMP(x, m);

	/*Searching*/
	i = j = 0;
	while (j < n)
	{
```

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

void main()
{
        char *y = "GCATCGCAGAGAGTATACAGTACG";
//"ABABDABACDABABCABAB";
        int n = strlen(y);
        char *x = "GCAGAGAG";           //"ABABCABAB";
        int m = strlen(x);
        KMP(x, m, y, n);
}
```

2.5 Thuật toán Not So Naïve

- Trình bày thuật toán
  - Trong quá trình thực thi của Not So Naïve các kí tự so sánh với các vị trí của mẫu theo thứ tự 1,2,3…m-2, m-1, 0. Group TTV Page 41 Với mỗi trường hợp, trên cửa sổ dịch chuyển là các vị trí của xâu văn bản y[j…j+m-1]: nếu x[0] = x[1] và x[1] != y[j+1] hoặc nếu x[1] != x[0] và y[1] = y[j+1] thì số bước dịch là 2 so với vị trí trước đó. Và dịch 1 bước

với các trường hợp còn lại. Do vậy, pha tiền xử lí có thể xác định được độ phức tạp thuật toán và không gian nhớ.

- 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);
- Kiểm nghiệm
  - Pha tiền xử lý

    $k=1$ and $\ell=2$

  - Pha tìm kiếm

First attempt

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: 2 (l)

Second attempt

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

Shift by: 1 (k)

Third attempt

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

Shift by: 2 (*l*)

Fourth attempt

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

Shift by: 2 (*l*)

Fifth attempt

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

Shift by: 1 (*k*)

Sixth attempt

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

Shift by: 1 (*k*)

Seventh attempt

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

Shift by: 1 (*k*)

Eighth attempt

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

Shift by: 1 (*k*)

Ninth attempt

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

Shift by: 1 (*k*)

Tenth attempt

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

Shift by: 1 (*k*)

Eleventh attempt

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

Shift by: 1 (*k*)

Twelfth attempt

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

Shift by: 1 (*k*)

Thirteenth attempt

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

Shift by: 1 (*k*)

Fourteenth attempt

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

Shift by: 2 (*l*)

- Lập trình

```
void NSN(char *x, int m, char *y, int n) {
  int j, k, ell;
  /* Preprocessing */
  if (x[0] == x[1]) {
```

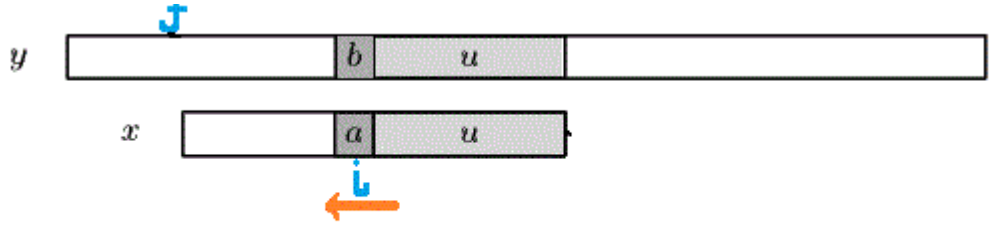
```
    k = 2;
    ell = 1;
  }
  else {
    k = 1;
    ell = 2;
  }
  /* Searching */
  j = 0;
  while (j <= n - m)
    if (x[1] != y[j + 1])
      j += k;
    else {
      if (memcmp(x + 2, y + j + 2, m - 2) == 0 &&
          x[0] == y[j])
        OUTPUT(j);
      j += ell;
    }
}
```

3. Tìm kiếm mẫu từ phải sang trái

3.1 Thuật toán Boyer-Moore

- Trình bày thuật toán
  - Thuật toán Boyer-Moore được coi là thuật toán hiệu quả nhất trong vấn đề tìm kiếm chuỗi trong các ứng dụng thường gặp. Các biến thể của nó được dùng trong các bộ soạn thảo cho các lệnh như <<search> và <<subtitle>>.
  - 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 (tìm được đoạn khớp với mẫu), nó sẽ dùng 2 hàm được tính toán trước để dịch cửa sổ sang phải. Hai hàm dịch chuyển này là good-suffix shift (dịch chuyển khớp) và bad-character shift (dịch chuyển xuất hiện).

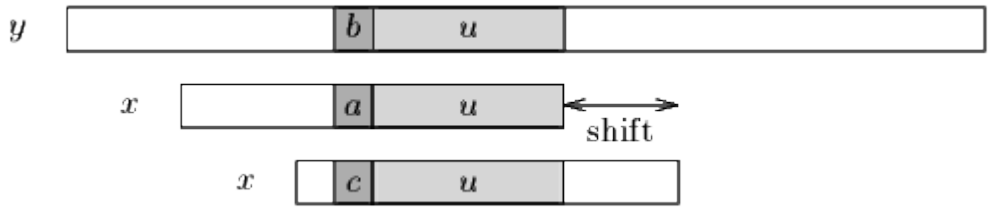
- Giả sử trong quá trình so sánh ta gặp mis-match tại x[i] = a và y[j+i] = b khi khớp tại vị trí thứ j



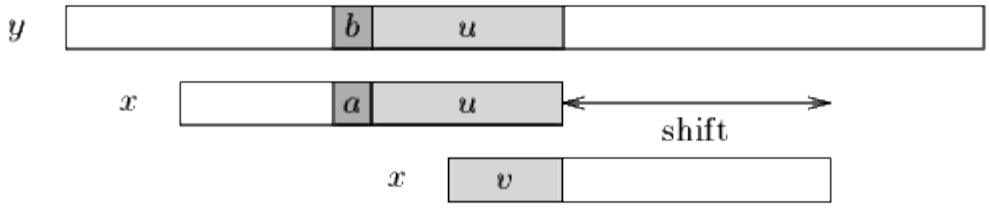
- Ta có các trường hợp dịch chuyển như sau:

a) Good-suffix shift

- Đoạn khớp *u* lại xuất hiện ở một vị trí khác trong x, dịch chuyển sang phải cho tới khi gặp ký tự khác với x[i], (c != a)

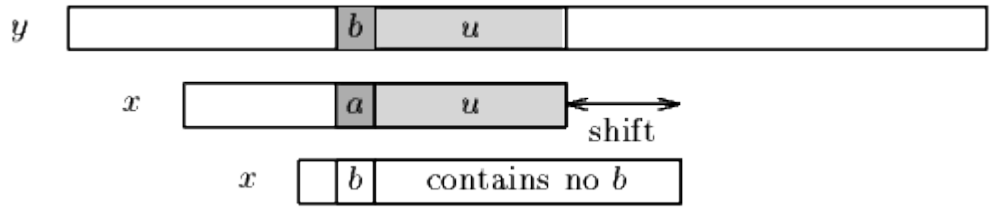


- Đoạn *u* không xuất hiện lại trong x, mà chỉ có phần hậu tố của u khớp với phần tiền tố của x, ta dịch một đạn sao cho phần hậu tố dài nhất v khớp với phần tiền tố của x.

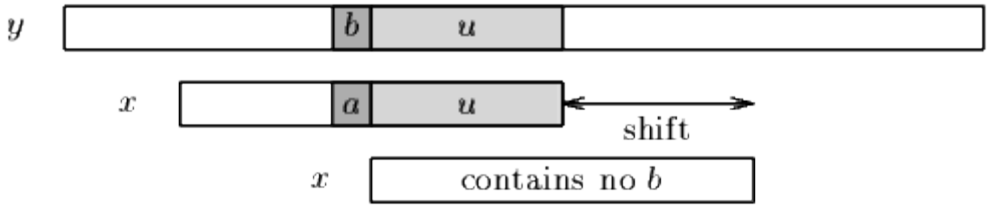


b) Bad-character shift

- Trong trường hợp không thể dịch chuyển theo Good-suffix shift. Ta dịch chuyển theo Bad-character shift như sau
- Nếu ký tự y[j+i] = b xuất hiện trong x, ta dịch chuyển sao cho y[j+i] khớp với ký tự bên phải nhất trong đoạn x[0 … m-2]



- Nếu ký tự y[j+i] = b không xuất hiện trong x, ta dịch chuyển hết x sang ngay sau y[j+1]



- Thuật toán Boyer-Moore sẽ chọn đoạn dịch chuyên dài nhất trong 2 hàm dịch chuyển Good-suffix shift và Bad-character shift. Hai hàm này tạo ra 2 bảng bmGs và bmBc
- Bảng bmGs và bmBc được tính toán trong thời gian $\boldsymbol{O}(m+\sigma)$ trước khi thực hiện việc tìm kiếm và cần 1 không gian phụ là $\boldsymbol{O}(m+\sigma)$. Giai đoạn tìm kiếm có độ phức tạp thời gian bậc hai nhưng lại chỉ có 3n phép so sánh khi tìm kiếm 1 chuỗi không có chu kì. Đối với việc tìm kiếm trong 1 khối lượng lớn các chữ cái thuật toán thực hiện với tốc độ nhanh "khủng khiếp". Khi tìm kiếm chuỗi $a^{m-1}b$ trong b^n chuỗi thuật toán chỉ sử dụng O(n/m) phép so sánh, đây được coi là "hình mẫu" cho bất cứ một thuật toán tìm kiếm chuỗi nào mà mẫu đã được xử lý trước.
  - Đặc điểm
    - Thực hiện so sánh từ phải sang trái
    - Pha tiền xử lý có độ phức tạp không gian và thời gian $\boldsymbol{O}(m+\sigma)$
    - Pha tìm kiếm có độ phức tạp thời gian O(m.n)
    - Trường hợp xấu nhất thực hiện so sánh 3n ký tự khi tìm kiếm mẫu không chu kỳ
    - Hiệu suất tốt nhất O(n/m)
  - Kiểm nghiệm
    - x[] = GCAGAGAG, m = 8

a) Giai đoạn tiền xử lý tạo bảng bmBc với thuật toán preBmBC

c	A	C	G	T

bmBC[c]	1	6	2	8

b) Giai đoạn tiền xử lý tạo bảng bmGs với thuật toán preBmGs

- Thuật toán suffixes tạo bảng suff[] chứa giá trị xâu hậu tố dài nhất

suff[i]	*f*	*g*	*suff[i + m -1 –f] < (i-g) ?*	*i>g ?*	*i*
8	0	7			7
0	6	6		False	6
4	5	1		False	5
0	5	1	True	True	4
2	3	1	False	True	3
0	1	1		False	2
1	0	0		False	1

Ta được mảng suff[] như sau

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

- Thuật toán preBmGs tạo bảng bmGs

Giai đoạn đầu

bmGs[i] = m-1-i	*j*	*j < m-1-i ?*	*bmGs[j]=m ?*	*Suff[i]= i+1 ?*	*i*
9	0	False		True	7
8	0			False	6
8	0			False	5
8	0			False	4
8	0			False	3
8	0			False	2
8	0			False	1
7	0	True	True	True	0
7	1	True	True		0
7	2	True	True		0
7	3	True	True		0
7	4	True	True		0
7	5	True	True		0
7	6	True	True		0
8	7	False	True		0

Giai đoạn sau

bmGs[m-1-suff[i])	*i<m-2 ?*	*suff[i]*	*i*

bmGs[6] = 7	True	1	0
bmGs[7] = 6	True	0	1
bmGs[7] = 5	True	0	2
bmGs[5] = 4	True	2	3
bmGs[7] = 3	True	0	4
bmGs[3] = 2	True	4	5
bmGs[7] = 1	True	0	6
	False		7

Ta được mảng bmGs[] như sau

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

c) Giai đoạn Tìm kiếm

First attempt

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

d) Shift by: 1 = max(*bmGs*[7]=*bmBc*[A]-7+7)

Second attempt

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

3 2 1

G C A G A G A G

e) Shift by: 4 = max (*bmGs*[5]=*bmBc*[C]-7+5

Third attempt

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 7 6 5 4 3 2 1

G C A G A G A G

f) Shift by: 7 (*bmGs*[0])

Fourth attempt

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

3 2 1

G C A G A G A G

g) Shift by: 4 = max (*bmGs*[5]=*bmBc*[C]-7+5)

Fifth attempt

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

Shift by: 7 (*bmGs*[6])

- Lập trình

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
using namespace std;
#define ASIZE 255
#define XSIZE 9
int bmBc[ASIZE];  //Mảng bmBc[ ]  là mảng chứa vị trí xuất hiện cuối cùng của các ký tự trong xâu x
```

```
int bmGs[XSIZE];  //Mảng chứa vị trí có hậu tố trùng nhau hoặc có phần chung giữa 2 xâu
int suff[XSIZE];  //Mảng chứa giá trị xâu hậu tố dài nhất
void Init()
{
	for(int i = 0; i < XSIZE; i++)
	{
		bmGs[i] = 999;
		suff[i] = 999;
	}
}

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) // mang bmBC chua vi tri xuat hien cuoi cung cua cac ky tu trong xau X, khong duyet Ky tu ben phai nhat (i < m-1)
    bmBc[x[i]] = m - i - 1;  // duyet xau tu trai qua phai, vi tri ky tu tinh' tu` phai qua trai. ( AGCG => A = 3; G = 2; C = 1)
}
void suffixes(char *x, int m/*, int *suff*/) { //xac dinh hau to chung dai nhat giua xau x[0 ... x] va x[0 ... m-1]
  int f, g, i;
  suff[m - 1] = m;
  g = m - 1;
  for (i = m - 2; i >= 0; --i) { //duyet tu cuoi xau ve dau xau
    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]) //neu trung nhau thi dua do dai trung vao
        --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]*/;
  Init();
  /* 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) {
                    cout<<j;
      j += bmGs[0];
    }
    else
          {
                int a = bmGs[i];
                int b =  bmBc[y[i + j]] - m + 1 + i;
                j += max(a, b);
          }
  }
}
void main()
{
        char *y = "GCATCGCAGAGAGTATACAGTACG";
//"ABABDABACDABABCABAB";
        int n = strlen(y);
        char *x = "GCAGAGAG";           //"ABABCABAB";
        int m = strlen(x);
        BM(x, m, y, n);
}
```

3.2 Thuật toán Berry – Ravindran

- Trình bày thuật toán

- Berry và Ravindran đã thiết kế ra thuật toán thực hiện bước dịch dựa vào tư tưởng bad-character. Nhưng ở đây lấy 2 kí tự liên tiếp ngoài cùng bên phải của sổ để xác định bước dịch. Quá trình chuẩn bị của thuật toán bao gồm việc xác định với mỗi cặp (a,b) vị trị ngoài cũng bên phải gần nhất bắt đầu xuất hiện ab trong x.

$$brBc[a,b] = \min \begin{cases} 1 & \text{if } x[m-1] = a \ , \\ m-i & \text{if } x[i]x[i+1] = ab \ , \\ m+1 & \text{if } x[0] = b \ , \\ m+2 & \text{otherwise} \ . \end{cases}$$

- Sau mỗi lần thử mẫu khi cửa sổ đang ở vị trí tương ứng y[j .. j+m-1] bước dịch tiếp theo sẽ là brBc[y[j+m], y[j+m+1]].

- Đặc điểm
  - 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 $O(m \times n)$
- Kiểm nghiệm
  - Tiền xử lý

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

  - Tìm kiếm

First attempt

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

Shift by: 1 (*brBc*[G][A])

Second attempt

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

Shift by: 2 (*brBc*[A][G])

Third attempt

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

Shift by: 2 (*brBc*[A][G])

Fourth attempt

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

Shift by: 10 (*brBc*[T][A])

Fifth attempt

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

Shift by: 7 (*brBc*[G][0])

Sixth attempt

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

Shift by: 10 (*brBc*[0][0])

- Lập trình

```
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)
```

```
        cout<<(j);
      j += brBc[y[j + m]][y[j + m + 1]];
    }
  }
```

3.3 Thuật toán Turbo Boyer-Moore

- Trình bày thuật toán
  - 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 là khác nhau vì chúng ta giả định rằng sự thay đổ itrước 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, 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 giới hạn bởi 2n.

- Đặc điểm
  - 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 vớithuật 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);
- Kiểm nghiệm

- Pha 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

- Pha tìm kiếm

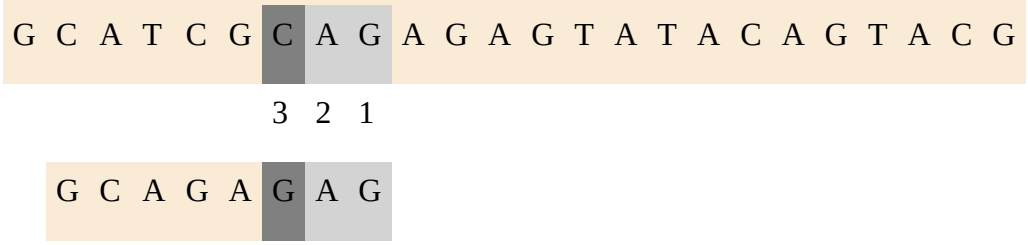
First attempt

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

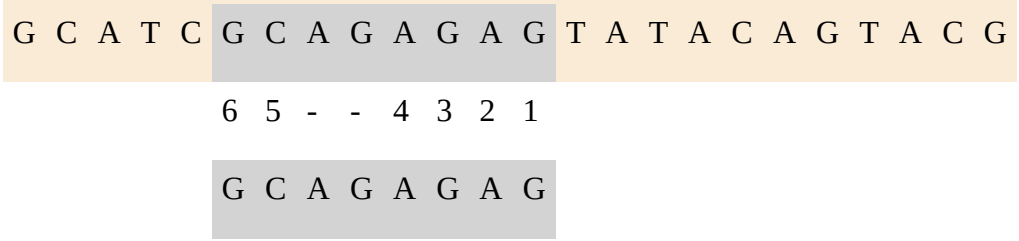
Shift by: 1 (*bmGs*[7]=*bmBc*[A]-8+8)

Second attempt



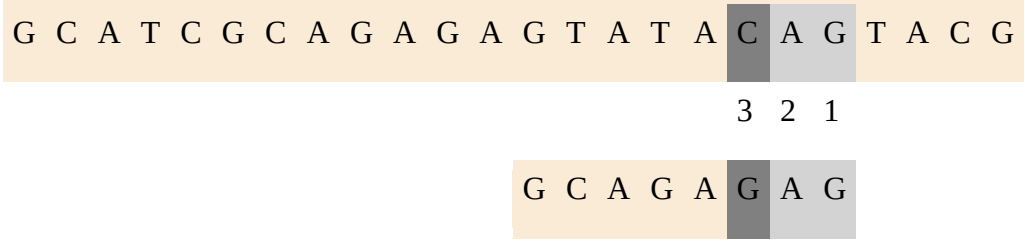
Shift by: 4 (*bmGs*[5]=*bmBc*[C]-8+6)

Third attempt



Shift by: 7 (*bmGs*[0])

Fourth attempt



Shift by: 4 (*bmGs*[5]=*bmBc*[C]-8+6)

Fifth attempt

G	C	A	G	A	G	A	G

Shift by: 7 (*bmGs*[6])

- Lập trình

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>

using namespace std;
#define ASIZE 255
#define XSIZE 9
int bmBc[ASIZE];  //Mảng bmBc[ ]  là mảng chứa vị trí xuất hiện cuối cùng của các ký tự trong xâu x
int bmGs[XSIZE];  //Mảng chứa vị trí có hậu tố trùng nhau hoặc có phần chung giữa 2 xâu
int suff[XSIZE];  //Mảng chứa giá trị xâu hậu tố dài nhất
void Init()
{
	for(int i = 0; i < XSIZE; i++)
	{
		bmGs[i] = 999;
		suff[i] = 999;
	}
}
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) // mang bmBC chua vi tri xuat hien cuoi cung cua cac ky tu trong xau X, khong duyet Ky tu ben phai nhat (i < m-1)
    bmBc[x[i]] = m - i - 1;  // duyet xau tu trai qua phai, vi tri ky tu tinh' tu` phai qua trai. ( AGCG => A = 3; G = 2; C = 1)
}
void suffixes(char *x, int m/*, int *suff*/) { //xac dinh hau to chung dai nhat giua xau x[0 ... x] va x[0 ... m-1]
  int f, g, i;
  suff[m - 1] = m;
  g = m - 1;
  for (i = m - 2; i >= 0; --i) { //duyet tu cuoi xau ve dau xau
    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]) //neu trung nhau thi dua do dai trung vao
        --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 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) {
        cout<<(j);
        shift = bmGs[0];
        u = m - shift;
      }
      else {
        v = m - 1 - i;
```

```
            turboShift = u - v;
            bcShift = bmBc[y[i + j]] - m + 1 + i;
            shift = max(turboShift, bcShift);
            shift = max(shift, bmGs[i]);
            if (shift == bmGs[i])
              u = min(m - shift, v);
            else {
              if (turboShift < bcShift)
                shift = MAX(shift, u + 1);
              u = 0;
            }
          }
          j += shift;
        }
}
void main()
{
        char *y = "GCATCGCAGAGAGTATACAGTACG";
//"ABABDABACDABABCABAB";
        int n = strlen(y);
        char *x = "GCAGAGAG";          //"ABABCABAB";
        int m = strlen(x);

        TBM (x, m, y, n);
}
```

4. Tìm kiếm mẫu từ vị trí xác định

   4.1 Thuật toán Colussi

   - Trình bày thuật toán

- 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.
- Định nghĩa thuật toán:
    - For $0 \leqslant i \leqslant m-1$:

$$\text{kmin}[i] = \begin{cases} d > 0 \text{ nếu } x[0 \dots i-1-d] = x[d \dots i-1] \text{ và } x[i-d] \neq x[i] \\ 0 \text{ trong trường hợp khác} \end{cases}$$

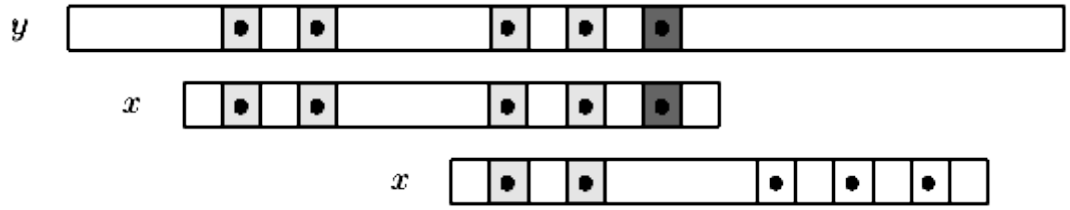
    - Khi kmin $\neq 0$ một chu kỳ kết thúc tại vị trí i trong x.
    - For $0 < i < m$:

$$\text{nếu kmin}[i] \begin{cases} \neq 0 \text{ thì i là một nohole} \\ \text{giá trị khác thì i là một hole} \end{cases}$$

    - Lấy nd+1 là số lượng của noholes trong x.
    - Bảng h chứa nd+1 noholes đầu tiên theo thứ tự tăng dần và tiếp đó là m-nd-1 holes theo thứ tự giảm dần:
        - for $0 \leqslant i \leqslant nd$, h[i] là một nohole và h[i] < h[i+1] với $0 \leqslant i < nd$;
        - for nd < i < m, h[i] là một hole và h[i] > h[i+1] với nd < i < m-1
- Nếu i là một hole thì rmin[i] là chu kỳ nhỏ nhất của x lớn hơn i
- Giá trị của first[u] là số nguyên nhỏ nhất v mà $u \leqslant h[v]$
- Tiếp theo, giả sử rằng x phù hợp với y[j .. j+m-1]. Nếu x[h[k]] = y[j+h[k]] với $0 \leqslant k < r < nd$ và $x[h[r]] \neq y[j+h[r]]$. Lấy j' = j+kmin[h[r]]. Tiếp

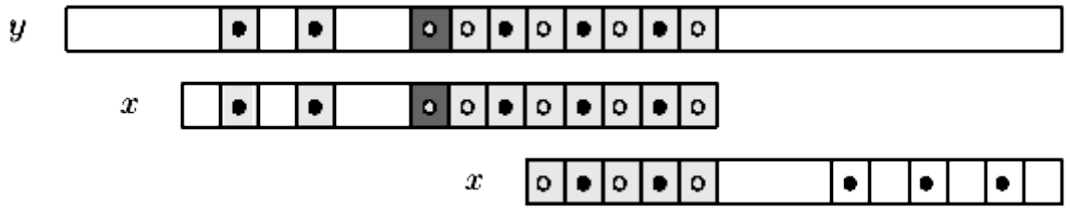
đó không có sự xuất hiện của x bắt đầu trong y[j .. j'] và x có thể được dịch kmin[h[r]] vị trí sang phải .

- Ngoài ra x[h[k]]=y[j'+h[k]] với 0 ⩽ k < first[h[r]-kmin[h[r]]] có nghĩa rằng việc so sánh có thể được tiếp tục với x[h[first[h[r] kmin[h[r]]]]] và y[j'+h[first[h[r]-kmin[h[r]]]]].



Hình 1: Không phù hợp với một nohole. Noholes là các vòng tròn màu đen và được so sánh từ trái qua phải

- Nếu x[h[k]]=y[j+h[k]] với 0 ⩽ k < r và x[h[r]] ≠ y[j+h[r]] với nd ⩽ r < m. Lấy j'=j+rmin[h[r]]. Tiếp đó, không có sự xuất hiện nào của x bắt đầu trong y[j .. j'] và x có thể được dịch kmin[h[r]] vị trí sang phải.
- Ngoài ra x[0 .. m-1-rmin[h[r]]]=y[j' .. j+m-1] có nghĩa rằng việc so sánh có thể được tiếp tục với x[h[first[m-1-rmin[h[r]]]]] và y[j'+h[first[m-1-rmin[h[r]]]]].



Hình 2: Không phù hợp với một hole. Noholes là các vòng tròn màu đen và được so sánh từ trái qua phải, trong khi holes là các vòng tròn màu trắng và được so sánh từ phải qua trái

- Để tính toán giá trị kmin, một bảng hmax được sử dụng và được định nghĩa như sau:
  - hmax[k] thỏamãn x[k .. hmax[k]1]=x[0.. hmax[k]k1] và x[hmax[k]] ≠ x[hmax[k]-k].
- Giá trị của nhd0[i] là số lượng các noholes chắc chắn nhỏ hơn i
- Chúng ta định nghĩa 2 hàm shift và next như sau:
  - shift[i]=kmin[h[i]] và next[i]=nhd0[h[i]-kmin[h[i]]] với i < nd;
  - shift[i]=rmin[h[i]] và next[i]=nhd0[m-rmin[h[i]]] với nd ⩽ i < m;

- shift[m]=rmin[0] và next[m]=nhd0[m-rmin[h[m-1]]].

- Do đó, trong suốt một lần thử, khi cửa sổ được đặt ở vị trí trên nhân tố văn bản y[j .. j+m-1], khi một không phù hợp xuất hiện giữa x[h[r]] và y[j+h[r]] cửa số phải được dịch đi shift[r] và những so sánh có thể được tiếp tục với mẫu ởi vị trí h[next[r]].

  Pha tiền xử lý có thể hoàn thành trong một O(m) không gian và thời gian. Pha tìm kiếm có thể hoàn thành trong O(n) độ phức tạp thời gian và hơn nữa tối đa $\frac{3}{2}$n lần việc so sánh ký tự văn bản được thực hiện trong suốt pha tìm kiếm.

  - 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
  - Kiểm nghiệm
- 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	

$nd = 3$

- Tìm kiếm

First attempt

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])

Second attempt

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

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

Third attempt

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 7 3 6 4 5

G C A G A G A G

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

Fourth attempt

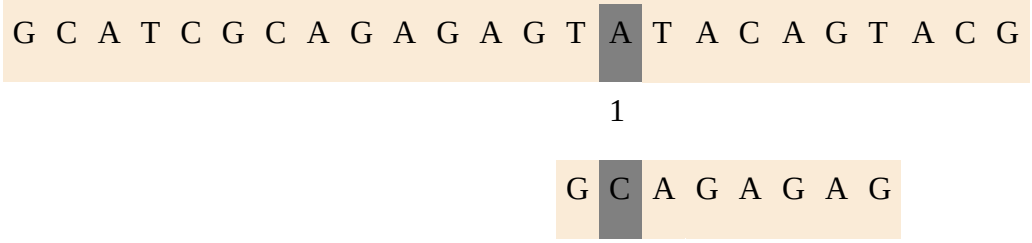
G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

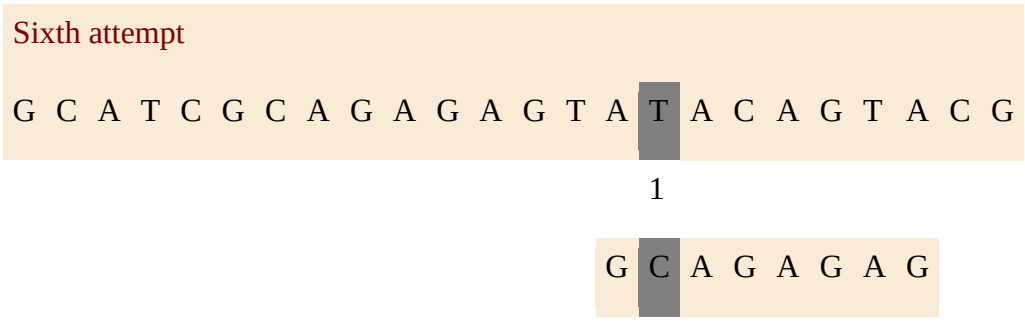
Shift by: 1 (*shift*[0])

Fifth attempt



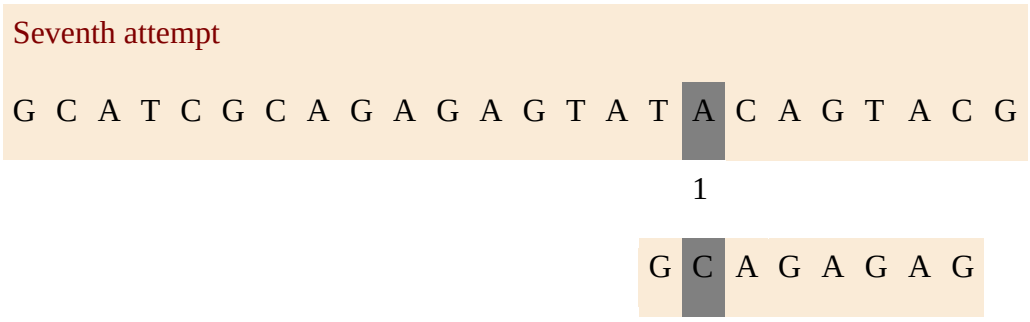
Shift by: 1 (*shift*[0])

Sixth attempt



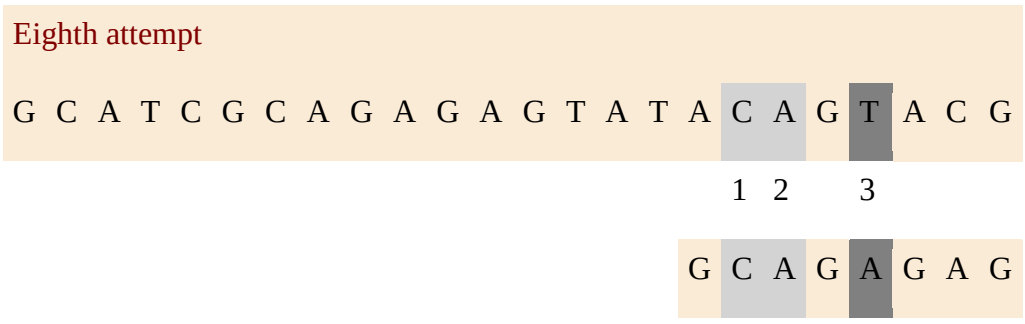
Shift by: 1 (*shift*[0])

Seventh attempt



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

Eighth attempt



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

- Lập trình

```
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];
  }
}
```

4.2 Thuật toán Skip Search

- Trình bày thuật toán

- Với mỗi kí tự trong bảng chữ cái, một thùng chứa (bucket) sẽ chứa tất cả các vị trí xuất hiện của kí tự đó trong xâu mẫu x. khi một kí tự xuất hiện k lần trong mẫu. bucket sẽ lưu k vị trí của kí tự đó. Khi mà xâu y chứa it kí tự hơn trong bản chữ cái thì sẽ có nhiều bucket rỗng. Quá trình xử lý của thuật toán Skip Search bao gồm việc tính các buckets cho tất cả các kí tự trong bảng chữ cái. for ***c*** in ***z***[***c***] = {***i***: 0 ***I m***-1 and ***x***[***i***] = ***c***} Thuật toán Skip Search có độ phức tạp bình phương trong trường hợp tồi nhất. nhưng cũng có trường hợp là O(n).

- Đặc điểm
  - Sử dụng thùng chứa (bucket) các vị trí xuất hiện của kí tự trong xâu mẫu. Pha xử lý có độ phực tạp thời gian và không gian chứa O(m+ ∂) Pha tìm kiếm có độ phức tạp thời gian O(mn)
- Kiểm nghiệm

  X = “GCAGAGAG”

  Y = ”GCATCGCAGAGAGTATACAGTACG”

  Pha tiền xử lý xác định:

c	$z[c]$
A	$(6, 4, 2)$
C	(1)
G	$(7, 5, 3, 0)$
T	$\emptyset$

  Pha tìm kiếm:

First attempt

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

Shift by: 8

Second attempt

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

Shift by: 8

Third attempt

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

- Lập trình

```
#include<iostream>
#include<algorithm>
#include<iomanip>
#include<cstdio>
#include<cstring>
#include<list>
using namespace std;
char x[100001],y[100001];
int ASIZE = 256, m, n;
void nhap(){
        printf("Nhap x: "); gets(x); m = strlen(x);
        printf("Nhap y: "); gets(y); n = strlen(y);
}
void SKIP(char *x, int m, char *y, int n){
        int i,j;
        list<int> z[ASIZE];
        list<int>::iterator it;
        /*preprocessing */

        for(i = m-1 ; i >=0 ;i--){
                z[x[i]].push_back(i);
        }
        /* search */
        for(j = m-1; j < n ; j +=m){
                for(it = z[y[j]].begin() ; it!= z[y[j]].end(); it++){
                        if(memcmp(x,y+j-(*it),m) == 0)
                        printf("position is %d\n",j- (*it));
                }
        }
```

```
}
main(){
        nhap();
        SKIP(x,m,y,n);
}
```

5. Tìm kiếm mẫu từ vị trí bất kì

5.1 Thuật toán Horspool

- Trình bày thuật toán
  - Phiên bản đơn giản của Boyer-More, chỉ dịch chuyển bad-character shift dựa theo ký tự cuối cùng bên phải của cửa sổ y[m-1]
- Đặc điểm
  - Phiên bản đơn giản của Boyer-Moore
  - Dễ dàng thực hiện
  - Pha tiền xử lý có độ phức tạp thời gian O(m+ σ) và đọ phức tạp không gian O(σ)
  - Pha tìm kiếm có độ phức tạp thời gian O(mn)
  - Giá trị trung bình của việc so sanh 1 ký tự trong khoảng 1/ σ và 2/(σ +1)
- Kiểm nghiệm

- Bảng bad-character shift bmBc

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

- Searching

Dịch theo bmBC[y[m-1]] , y[m-1] = ký tự cuối cùng bên phải nhất của cửa sổ

First attempt

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**

Shift by: 1 (*bmBc*[A])

Second attempt

```
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
```

Shift by: 2 (*bmBc*[G])

Third attempt

```
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
```

Shift by: 2 (*bmBc*[G])

Fourth attempt

```
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 3 4 5 6 7 8 1
          G C A G A G A G
```

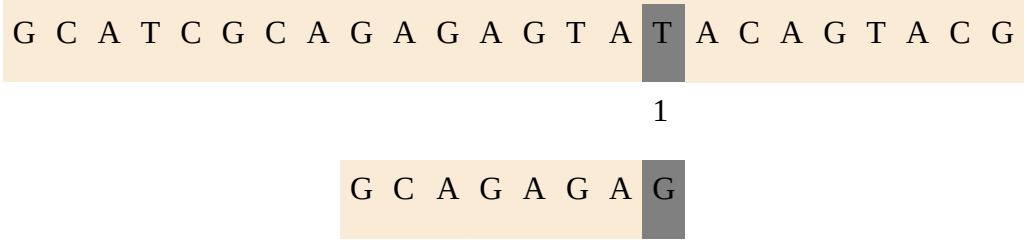
Shift by: 2 (*bmBc*[G])

Fifth attempt

```
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
```

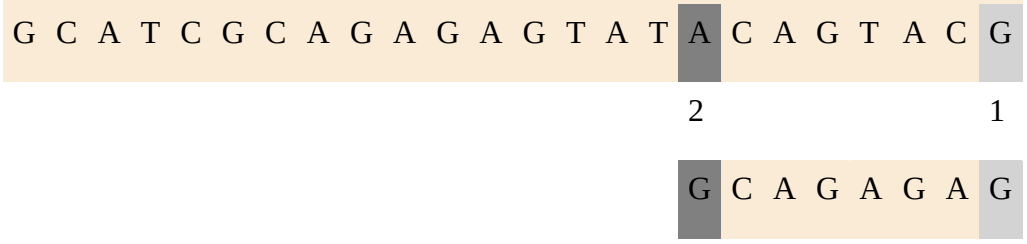
Shift by: 1 (*bmBc*[A])

Sixth attempt



Shift by: 8 (*bmBc*[T])

Seventh attempt



Shift by: 2 (*bmBc*[G])

- Lập trình

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
using namespace std;
#define ASIZE 255
#define XSIZE 9
int bmBc[ASIZE];  //Mảng bmBc[ ]  là mảng chứa vị trí xuất hiện cuối cùng của các ký tự trong xâu x
int bmGs[XSIZE];  //Mảng chứa vị trí có hậu tố trùng nhau hoặc có phần chung giữa 2 xâu
int suff[XSIZE];  //Mảng chứa giá trị xâu hậu tố dài nhất
```

```
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) // mang bmBC chua vi tri xuat hien cuoi cung cua cac ky tu trong xau X, khong duyet Ky tu ben phai nhat (i < m-1)
		bmBc[x[i]] = m - i - 1;  // duyet xau tu trai qua phai, vi tri ky tu tinh' tu` phai qua trai. ( AGCG => A = 3; G = 2; C = 1)
}
void HORSPOOL(char *x, int m, char *y, int n) {
	int j/*, bmBc[ASIZE]*/;
	char c;
	/* Preprocessing */
	preBmBc(x, m/*, bmBc*/);		//tien xu ly
	/* Searching */
	j = 0;
	while (j <= n - m) {
		c = y[j + m - 1];
		char * buff = y + j;
		if (x[m - 1] == c && memcmp(x, y + j, m - 1) == 0)
			cout<<j;
		j += bmBc[c];
	}
}
void main()
{
	char y[] = "GCATCGCAGAGAGTATACAGTACG"; //"ABABDABACDABAACABABE";
	int n = strlen(y);
	char *x =  "GCAGAGAG";		//"ABAACABAB";
	int m = strlen(x);
```

HORSPOOL(x, m, y, n);

}

5.2 Thuật toán Quick Search

- Trình bày thuật toán
  - Thuật toán Quick Search chỉ sử dụng bảng bad charater. Giả sử trên cửa sổ dịch chuyển đang là đoạn văn bản y[j….j+m-1], độ dài bước dịch đồng

    Group TTV Page 101 đều là 1 bước. Vì vậy, vị trí tiếp theo có liên quan mật thiết tới số bước dịch chuyển. Quick Search sẽ quan tâm tới vị trí y[j+m]. ***qsBc***[***c***]=***min***{i: 0 < ***i m*** and ***x***[***m-i***]=***c***} if ***c*** có trong ***x*** hoặc bằng ***m***+1 với trường hợp còn lại.
- Đặc điểm
  - Phiên bản đơn giản của Boyer-Moore
  - Chỉ sử dụng Bad-Character shift
  - Dễ thực thi
  - Giai đoạn tiền xử lý có độ phức tạp thời gian và không gian O(m+σ)
  - Giai đoạn tìm kiếm có độ phức tạp thời gian O(mn)
  - Rất nhanh trong thực thi với mẫu ngắn và bảng chữ cái lớn
- Kiểm nghiệm

a	A	C	G	T
qsBC[a]	2	7	1	9

First attempt

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

Shift by: 1 (*qsBc*[G])

Second attempt

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

Shift by: 2 (*qsBc*[A])

Third attempt

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

Shift by: 2 (*qsBc*[A])

Fourth attempt

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

Shift by: 9 (*qsBc*[T])

Fifth attempt

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

Shift by: 7 (*qsBc*[C])

- Lập trình

```
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)
      cout<<(j);
    j += qsBc[y[j + m]];          /* shift */
  }
}
```

5.3 Thuật toán Raita

- Trình bày thuật toán
  - 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 bởi 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.

- Đặc điểm
  - 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
  - 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(m.n).
- Kiểm nghiệm
  - Bảng bad-character shift bmBc

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

  - Searching

First attempt

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

Shift by: 1 (*bmBc*[A])

Second attempt

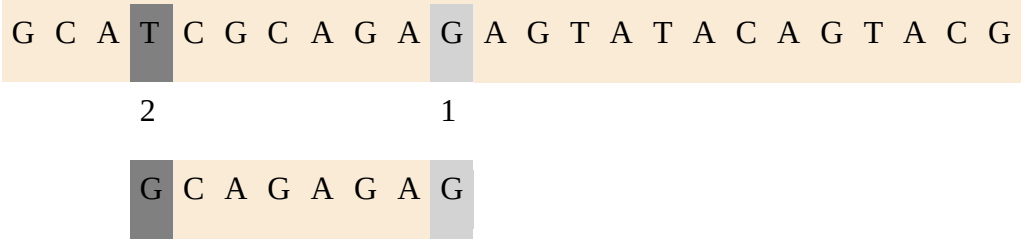
G C A T C G C A G A G A G T A T A C A G T A C G

2 1

G C A G A G A G

Shift by: 2 (*bmBc*[G])

Third attempt

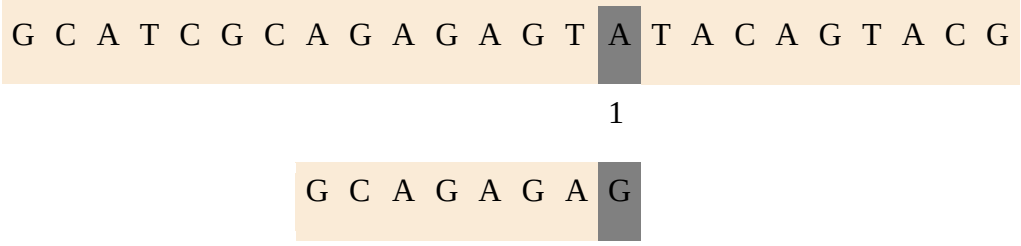


Shift by: 2 (*bmBc*[G])

Fourth attempt

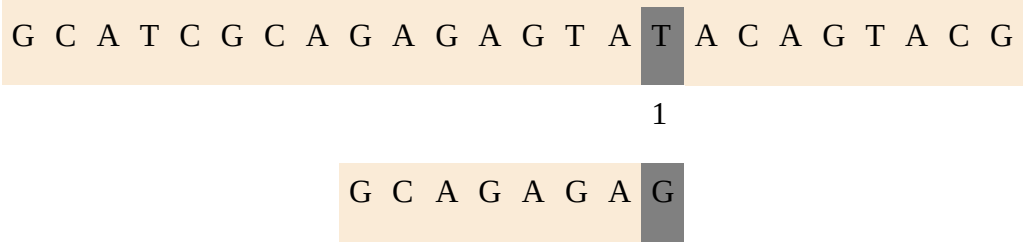
Shift by: 2 (*bmBc*[G])

Fifth attempt



Shift by: 1 (*bmBc*[A])

Sixth attempt



Shift by: 8 (*bmBc*[T])

Seventh attempt

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

Shift by: 2 (*bmBc*[G])

- Lập trình

```
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) // mang bmBC chua vi tri xuat hien cuoi cung cua cac ky tu trong xau X, khong duyet Ky tu ben phai nhat (i < m-1)
                bmBc[x[i]] = m - i - 1;  // duyet xau tu trai qua phai, vi tri ky tu tinh' tu` phai qua trai. ( AGCG => A = 3; G = 2; C = 1)
}
void RAITA(char *x, int m, char *y, int n) {
  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 && middleCh == y[j + m/2] &&
          firstCh == y[j] &&
          memcmp(secondCh, y + j + 1, m - 2) == 0)
        OUTPUT(j);
      j += bmBc[c];
    }
}
```