HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

KHOA CÔNG NGHỆ THÔNG TIN 1

BÁO CÁO

MÔN: CHUYÊN ĐỀ CÔNG NGHỆ PHẦN MỀM

CHỦ ĐỀ: PATTERN SEARCHING

Giảng viên: Nguyễn Duy Phương

Sinh viên: Đỗ Quốc Phong
Mã SV: B20DCCN492

Hà Nội, 5/20/2024

Mục lục

I. Tìm kiếm mẫu từ trái qua phải

1. Thuật toán Brute Force

- **Đặc điểm**
  - + Không có giai đoạn tiền xử lý
  - + Bộ nhớ cần dùng cố định
  - + Luôn luôn dịch 1 bước sang phải
  - + Việc so sánh có thể phải dùng trong các trường hợp
  - + Độ phức tạp pha thực thi là O(m x n)
  - + So sánh khoảng 2n ký tự
- **Trình bày thuật toán**
  - + Thuật toán Brute Force kiểm tra ở tất cả các vị trí trong đoạn văn bản giữa 0 và n-m, không cần quan tâm liệu mẫu này có tồn tại ở vị trí đó hay không. Sau đó, sau mỗi lần kiểm tra mẫu sẽ dịch sang phải một vị trí.
  - + 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).
- **Code**

```
void BruteForce(string &x, string &y) {
        int m = x.length();
        int n = y.length();
        for(int i = 0; i <= n - m; i++) {
            for(int j = 0; j < m && x[j] == y[i + j]; j++);
                if(j == m) {
                    cout << "FOUND AT " << i << endl;
                }
            }
}
```

– **Kiểm nghiệm thuật toán**

Xâu X=”AB”

Xâu Y=”ABDAAB”

1	**Y**	A	B	D	A	A	B
	X	A (1)	B (2)				
2	**Y**	A	B	D	A	A	B
	X		A	B			
3	**Y**	A	B	D	A	A	B
	X			A			
4	**Y**	A	B	D	A	A	B
	X				A (1)	B	
5	**Y**	A	B	D	A	A	B
	X					A (1)	B (2)

2. Thuật toán Knuth-Morris-Pratt

– **Đặc điểm**

+ Thực hiện từ trái qua phải
+ Pha tiền xử lý PreKMP có độ phức tạp không gian và thời gian là O(m)
+ Pha tìm kiếm có độ phức tạp thời gian O(m+n)

– **Trình bày thuật toán**

+ Thuật toán là bản đơn giản và xử lý tương tự như thuật toán Morris-Pratt khi cố gắng dịch chuyển một đoạn dài nhất sao cho một tiền tố (prefix) *v* của x trùng với hậu tố (suffix) của u
+ Điểm khác nhau là KMP sẽ thực hiện thêm so sánh *c* và *b*, có nghĩa KMP sẽ thực hiện một pha dòm trước ký tự bên phải đoạn đang so khớp. Do đó mỗi bước KMP sẽ dịch chuyển thêm một bước sang phải so với MP nếu *c != b*

– **Code**

```
void PreKMP(const string &X, vector<int> &kmpNext) {
  int m = X.length();
  int len = 0;
  kmpNext[0] = 0;
```

```
    int i = 1;

    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++;
            }
        }
    }
}

void KMP(const string &X, int m, const string &Y, int n) {
    vector<int> kmpNext(m);
    PreKMP(X, kmpNext);

    int i = 0, j = 0;
    while (i < n) {
        if (X[j] == Y[i]) {
            i++;
            j++;
        }

        if (j == m) {
            cout << "FOUND AT " << i - j << endl;
            j = kmpNext[j - 1];
        } else if (i < n && X[j] != Y[i]) {
            if (j != 0) {
                j = kmpNext[j - 1];
```

```
            } else {
               i++;
            }
         }
      }
   }
```

- **Kiểm nghiệm thuật toán**

  + Input:
    - xâu mẫu X=”ABABCABAB” độ dài m=9
    - Xâu văn bản Y=”ABADABABCABAB” độ dài n=13

B1: PreKMP(X,kmpNext[])

i	**len**	**X[i]=X[len]**	**kmpNext[]**
	0		0
1	0	B!=A	0,0
2	0	A=A	0,0,1
3	1	B=B	0,0,1,2
4	2	C!=A	0,0,1,2
4	0	C!=A	0,0,1,2,0
5	0	A=A	0,0,1,2,0,1
6	1	B=B	0,0,1,2,0,1,2
7	2	A=A	0,0,1,2,0,1,2,3
8	3	B=B	0,0,1,2,0,1,2,3,4

→ kmpNext[]={0,0,1,2,0,1,2,3,4}

B2:KMP(X,m,Y,n,kmpNext[])

STT	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
X	A	B	A	D	A	B	A	B	C	A	B	A	B	C	A	B	A	B
Y	A	B	A	**B**	C	A	B	A	B									
I		J=1																
j		I=3																
X	A	B	A	**D**	A	B	A	B	C	A	B	A	B	C	A	B	A	B
Y			A	**B**	A	**B**	C	A	B	A	B							
I																		
j																		

X	A	B	A	**D**	A	B	A	B	C	A	B	A	B	C	A	B	A	B
Y				A	**B**	A	**B**	C	A	B	A	B						
I																		
j																		
X	A	B	A	D	**A**	**B**	**A**	**B**	**C**	**A**	**B**	**A**	**B**	C	A	B	A	B
Y					**A**	**B**	**A**	**B**	**C**	**A**	**B**	**A**	**B**					
I																		
j																		
X	A	B	A	**D**	A	B	A	B	C	A	B	A	B	C	A	B	A	B
Y										**A**	**B**	**A**	**B**	**C**	**A**	**B**	**A**	**B**
I																		
j																		

3. Thuật toán Karp- Rabin

- **Đặc điểm**
  - + Biểu diễn xâu kí tự bằng số nguyên
  - + Sử dụng hàm băm
  - + Độ phức tạp thuật toán O((n-m+1)*m)
- **Trình bày thuật toán**
  - + Hàm băm cung cấp phương thức đơn giản để tránh những con số phức tạp trong việc so sánh những kí tự trong hầu hết các trường hợp thực tế.
  - + Thay cho việc kiểm tra từng vị trí trong văn bản nếu như có mẫu xuất hiện, nó chỉ phải kiểm tra những đoạn "gần giống" xâu mẫu.
  - + Để kiểm tra sự giống nhau giữa 2 từ sử dụng hàm băm.
  - + Giúp cho việc đối chiếu xâu, hàm băm hash:
    - Có khả năng tính toán được
    - Đánh giá xâu mức cao.
    - Hash(y[j+1…j+m]) được tính toán dễ hơn dựa trên hash(y[j…j+m-1]) và hash(y[j+m]):
      - hash(y[j+1 .. j+m])= rehash(y[j], y[j+m], hash(y[j .. j+m-1]).

        Với từ w có độ dài m có hash(w) là:

      - hash(w[0 .. m-1])=(w[0]*2m-1+ w[1]*2m-2+⋯+ w[m-1]*20) mod q
        Với q là một số lớn.

- Sau đó rehash(a,b,h)= ((h-a*2m-1)*2+b) mod q

+ Pha chuẩn bị của Karp- Rabin có hàm hash(x) có thể tính toán được. nó được dùng lại không gian nhớ và có độ phức tạp O(m)
+ Trong quá trình thực thi nó so sánh hash(x) với hash([j..j+m-1]) với 0<= j<=n-m. nếu so sánh đúng, nó phải kiểm tra lại xem các kí tự trong x và y có đúng bằng nhau hay không x=y[j…j+m-1]

– **Code**

```
void RK(const string &x, int m, const string &y, int n, int prime) {
   int hashX = 0;
   int hashY = 0;
   for (int i = 0; i < m; i++) {
      hashX += x[i] * pow(prime, i);
      hashY += y[i] * pow(prime, i);
   }

   int i = 0;
   while (i <= n - m) {
      if (hashY == hashX) {
         if (y.substr(i, m) == x) {
            cout << "FOUND AT " << i << endl;
         }
      }
      if (i < n - m) {
         hashY = (hashY - y[i]) / prime + y[i + m] * pow(prime, m - 1);
      }
      i++;
   }
}
```

– **Kiểm nghiệm thuật toán**

+ Input:
  - X=”ABC” m=3;
  - Y=”EABABCACD” n=9
  - Bảng định nghĩa các kí tự:

A	B	C	D	E

65	66	67	68	69

- Prime=3

	0	1	2	3	4	5	6	7	8
Y	E	A	B	A	B	C	A	C	D

+ Tiền xử lý:

Hash(ABC)= 65*prime^0 + 66*prime^1 + 67*prime^2= 866

Hash(EAB) = 69 + 65*prime + 66*prime^2=858

i	Substring	Hash(y)	== hash(ABC)?
0	**EAB**ABCACD	Hash(EAB)=858	No
1	E**ABA**BCACD	Hash(ABA)= (858-E)/prime + A*prime^2=848	No
2	EA**BAB**CACD	Hash(BAB)= (858-A)/prime + B*prime^2=855	No
3	EAB**ABC**ACD	Hash(ABC)= (858-B)/prime + C*prime^2=866	YES, OUT(3)
4	EABA**BCA**CD	Hash(BCA)= (858-A)/prime + A*prime^2=852	NO
5	EABAB**CAC**D	Hash(CAC)= (858-B)/prime + B*prime^2=865	NO
6	EABABC**ACD**	Hash(ACD)= (858-C)/prime + D*prime^2=878	NO

4. Thuật toán Morris-Pratt

- **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)
- **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.
+ 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).

- **Code**

```
#include <iostream>
#include <string>

using namespace std;

#define MAX 12
int mpNext[MAX];

void Init() {
   for(int i = 0; i < MAX; i++)
      mpNext[i] = 999;
}

void preMp(const string &x, int m) {
   int i = 0;
   int j = mpNext[0] = -1;

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

```
        i++;
        j++;
        mpNext[i] = j;
    }
}

void MP(const string &x, int m, const string &y, int n) {
    int i, j;
    Init();
    preMp(x, m);

    for(int k = 0; k < m; k++) {
        cout << x[k] << " " << mpNext[k] << endl;
    }

    i = j = 0;
    while (j < n) {
        while (i > -1 && x[i] != y[j])
            i = mpNext[i];
        i++;
        j++;
        if (i >= m) {
            cout << "FOUND AT " << j - i << endl;
            i = mpNext[i];
        }
    }
}

int main() {
    string x = "GCAGAGAG";
    int m = x.length();
    string y = "GCATCGCAGAGAGTATACAGTACG";
    int n = y.length();
    MP(x, m, y, n);
    return 0;
```

}

- **Kiểm nghiệm thuật toán**

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

5. Thuật toán Search with an automaton

- **Đặc điểm**
  - + yêu cầu xây dựng automation đơn định (DFA)
  - + pha xử lý có độ phức tạp tính toán là O(n∂)
  - + quá trình tím kiếm có độ phức tạp là O(n)
  - + trường hợp DFA đươc xây dựng bằng cây cân bằng thì độ phức tạp là O(nlog(∂))
- **Trình bày thuật toán**
  - + Trong thuật toán này, quá trình tìm kiếm được đưa về một quá trình biến đổi trạng thái automat. Hệ thống automat trong thuật toán DFA sẽ được xây dựng dựa trên xâu mẫu. Mỗi trạng thái (nút) của automat lúc sẽ đại diện cho số ký tự đang khớp của mẫu với văn bản. Các ký tự của văn bản sẽ làm thay đổi các trạng thái. Và khi đạt được trạng cuối cùng có nghĩa là đã tìm được một vị trí xuất hiện ở mẫu.
  - + Thuật toán này có phần giống thuật toán Knuth-Morris-Pratt trong việc nhảy về trạng thái trước khi gặp một ký tự không khớp, nhưng thuật toán DFA có sự đánh giá chính xác hơn vì việc xác định vị trí nhảy về dựa trên ký tự không khớp của văn bản (trong khi thuật toán KMP lùi về chỉ dựa trên vị trí không khớp).
  - + Việc xây dựng hệ automat khá đơn giản khi được cài đặt trên ma trận kề. Khi đó thuật toán có thời gian xử lý là O(n) và thời gian và bộ nhớ để tạo ra hệ automat là O(m*d) (tùy cách cài đặt) . Nhưng ta nhận thấy rằng trong DFA chỉ có nhiều nhất m cung thuận và m cung nghịch, vì vậy việc lưu trữ các cung không cần thiết phải lưu trên ma trận kề mà có thể dùng cấu trúc danh sách kề Forward Star để lưu trữ. Như vậy thời gian chuẩn bị và lượng bộ nhớ chỉ là O(m). Tuy nhiên thời gian tìm kiếm có thể tăng lên một chút so với cách lưu ma trận kề.
- **Code**

```
#include <iostream>

#include <algorithm>

#include <iomanip>
```

```
#include <set>
#include <cstring>

#define For(i,a,b) for(long i = a; i <= b; i++)
typedef int Graph[10001][256];

using namespace std;

Graph aut;
char x[10001], y[100001];
int m, n, ASIZE;
string s = "";

void nhap() {
    cout << "Nhap x: ";
    cin.getline(x, 10001);
    m = strlen(x);

    cout << "Nhap y: ";
    cin.getline(y, 100001);
    n = strlen(y);

    ASIZE = 0;
```

```
    set<char> se;

    for(int i = 0; i < m; i++) {
        if(se.find(x[i]) == se.end()) {
            se.insert(x[i]);
            s += x[i];
            ASIZE++;
        }
    }

    for(int i = 0; i < n; i++) {
        if(se.find(y[i]) == se.end()) {
            se.insert(y[i]);
            s += y[i];
            ASIZE++;
        }
    }
}

void preAut(char *x, int m, Graph aut) {
    memset(aut, 0, sizeof(aut));
    aut[0][x[0]] = 1;
    aut[1][x[0]] = 1;
```

```
    For(i, 2, m) {
        int vt = aut[i-1][x[i-1]];
        for(int j = 0; j < ASIZE; j++) {
            aut[i][s[j]] = aut[vt][s[j]];
        }
        aut[i-1][x[i-1]] = i;
    }
}

void AUT() {
    int state = 0;
    for(int i = 0; i < n; i++) {
        state = aut[state][y[i]];
        if(state == m)
            cout << "position is " << i - m + 1 << endl;
    }
}

int main() {
    nhap();
    preAut(x, m, aut);
    AUT();
```

return 0;

}

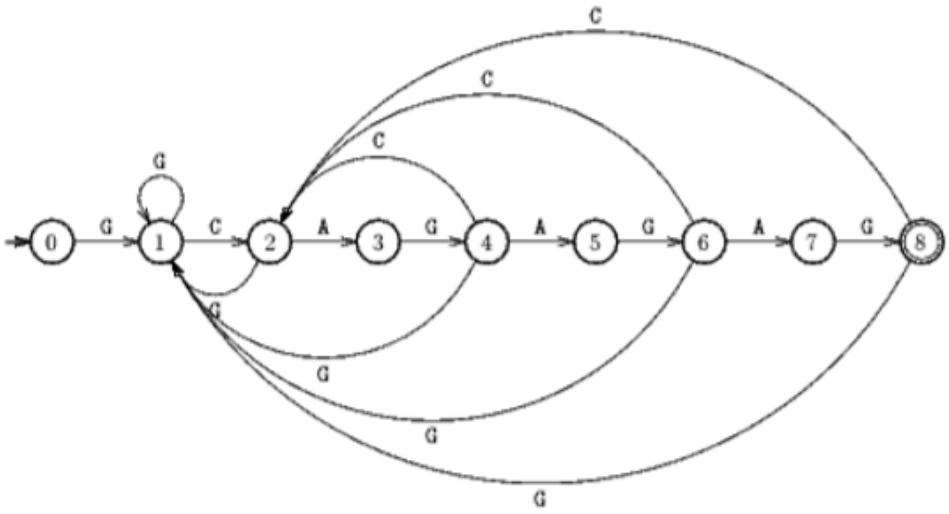
– **Kiểm nghiệm thuật toán**

Input:

X = “GCAGAGAG”

Y =”GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lý xây dựng DFA:



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

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

II. Tìm kiếm mẫu từ phải qua trái

1. Thuật toán Boyer-Moore

– Đặc điểm

+ Thực hiện so sánh từ phải sang trái

+ Có 1 bước tiền xử lý preBM để xác định khoảng cách từ 1 kí tự trong xâu mẫu đến kí tự cuối cùng
+ Độ phức tạp thuật toán: O(m)

– **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.

– **Code**

```
#include <iostream>
#include <string>
using namespace std;

const int ASize = 26;

void PreBM(string x, int m, int preBM[]) {
   for (int i = 0; i < ASize; i++) {
      preBM[i] = m;
   }
   for (int i = 0; i < m - 1; i++) {
      preBM[x[i] - 'A'] = m - i - 1;
   }
}

void BMSearching(string x, int m, string y, int n, int preBM[]) {
   int j = m - 1;
   while (j < n) {
      bool check = false;
      for (int i = m - 1; i >= 0; i--) {
         if (x[i] != y[j - (m - i - 1)]) {
            check = true;
            break;
```

```
            }
        }
        if (!check) {
            cout << "Tim thay: tai vi tri " << j << endl;
        }
        j += preBM[y[j] - 'A'];
    }
}

int main() {
    int *preBM = new int[ASize];
    string x = "ABCDAB";
    PreBM(x, 6, preBM);
    cout << "Bang du lieu chuan BM: ";
    for (int i = 0; i < ASize; i++) {
        cout << preBM[i] << " ";
    }
    cout<<endl;
    string y = "AABCDABBDEFAABCDABCDAB";
    BMSearching(x, 6, y, 22, preBM);

    delete[] preBM; // Gi?i phóng b? nh? du?c c?p phát d?ng
    return 0;
}
```

– **Kiểm nghiệm thuật toán**

Input:

X=” ABCDAB”, m=6
Y=” AABCDABBDEFAABCDABCDAB”, n=22

Tiền xử lý

X[i]	A	B	C	D	*
preBM[i]	1	4	3	2	6

j=m-1=5:

J		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
5	**Y**	A	A	B	C	D	**A**	B	B	D	E	F	A	A	B	C	D	A	B	C	D	A	B
	X	A	B	C	D	A	**B**																
Shift by 4																							
1	**Y**	A	A	B	C	D	A	***B***	B	D	E	F	A	A	B	C	D	A	B	C	D	A	B
	X		A	B	C	D	A	***B***															
Shift by 4, OUTPUT 1																							
3	**Y**	A	A	B	C	D	A	B	B	D	E	F	A	A	B	C	**D**	A	B	C	D	A	B
	X											A	B	C	D	A	**B**						
Shift by 2																							
4	**Y**	A	A	B	C	D	A	B	B	D	E	F	A	A	B	C	D	A	***B***	C	D	A	B
	X													A	B	C	D	A	***B***				
Shift by 4 OUTPUT 12																							
6	**Y**	A	A	B	C	D	A	B	B	D	E	F	A	A	B	C	D	A	B	C	D	A	***B***
	X																	A	B	C	D	A	***B***
Shift by 4, OUTPUT 16, END																							

OUTPUT: 1,12,16

2. Thuật toán Turbo- Boyer- Moore

- **Đặc điểm**
  - + Đây là thuật toán đơn giản hóa từ thuật toán Boyer-moore.
  - + Dễ cài đặt
- **Trình bày thuật toán**
  - + Tuned Boyer-moore là cài đặt đơn giản của thuật toán Boyer-Moore. Chi phí cho thuật toán string-matching thường phần nhiều là việc kiểm tra
  - + Để tránh việc phải so sánh nhiều lần. Chúng ta có thể thực hiện nhiều bước dịch hơn trước khi thực sự so sánh xâu. Thuật toán này sẽ sử dụng hàm bad-character xác định bước dịch. Và tìm x[m-1] trong y cho tới khi nào tìm được. Yêu cầu lưu giá trị bmBc[x[m-1]] vào biến shift và đặt lại giá trị bmBc[x[m-1]] = 0 . Khi ta tìm được vị trí x[m-1] trong y, thì bước dịch tiếp theo sẽ là shift.
- **Code**

```
#include<bits/stdc++.h>
```

```
#define For(i,a,b) for(long i = a;i<=b;i++)
using namespace std;

char x[100001], y[100001];
int m, n, ASIZE = 256;

void nhap() {
   cout << "Nhap x: ";
   cin.getline(x, 100001);
   m = strlen(x);

   cout << "Nhap y: ";
   cin.getline(y, 100001);
   n = strlen(y);
}

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 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 - m) {
        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 - m)
            cout << "Vi tri la " << j << endl;
        j += shift;  // Di chuy?n
    }
}

int main() {
    nhap();
    TUNEDBM(x, m, y, n);
    return 0;
}
```

– **Kiểm nghiệm thuật toán**

Input:

X = “GCAGAGAG”
Y =”GCATCGCAGAGAGTATACAGTACG”

X[i]	A	C	G	T	*
preBmBc	1	6	0	8	8

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

	X						G	C	A	G	A	G	A	G											
	X						G	C	A	G	A	G	A	G											
SHIFT BY 2, OUT PPUT 5																									
7	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
	X								G	C	A	G	A	G	A	G									
	X									G	C	A	G	A	G	A	G								
	X																	G	C	A	G	A	G	A	G
	X																	G	C	A	G	A	G	A	G
END																									

3. Thuật toán Zhu-Kakaota

- **Đặc điểm**
  - + Là một biến thể của Boyer Moore.
  - + Sử dụng 2 kí tự liên tiếp nhau để tính toán bước dịch bad charater
  - + Pha cài đặt có độ phức tạp thuật toán và không gian nhớ là O(m+σ2)
  - + Pha thực thi có độ phức tạp là O(mxn)
- **Trình bày thuật toán**
  - + Zhu và Takaoka thiết kế thuật toán mà chúng thực thi dựa trên bad charater. Trong quá trình tìm kiếm, việc so sánh được thực hiện từ phải qua trái, và khi cửa sổ đang ở vị trí y[j…j+m-1] và xuất hiện sự khác nhau giữa x[m- k] và y[j+m-k] trong khi x[m-k+1 .. m-1]=y[j+m-k+1 .. j+m1] . bước dịch good suffix cũng được sử dụng để tính toán bước dịch.
  - + Pha tiền xử lí của thuật toán bao gồm việc tính toán mỗi cặp kí tự (a,b) với a,b là mút bên phải của đoạn x[0…m-2]

    Với a,b thuộc : ztBc[a, b]=k và k có các giá trị:

    K < m-2 và x[m-k .. m-k+1]=ab và ab không xuất hiện trong đoạn x[m-k+2 .. m-2] hoặc

    k=m-1 và x[0]=b và ab không xuất hiện trong đoạn x[0 .. m-2]

    hoặc

    k=m and x[0] b và ab không xuất hiện trong đoạn x[0 .. m-2]

- **Code**

```
#include<bits/stdc++.h>
```

```
#define For(i,a,b) for(long i = a;i<=b;i++)
using namespace std;

char x[100001], y[100001];
int m, n, ASIZE = 256, XSIZE;

void nhap(){
   cout << "Nhap x: ";
   cin.getline(x, 100001);
   m = strlen(x);
   XSIZE = m;

   cout << "Nhap y: ";
   cin.getline(y, 100001);
   n = strlen(y);
}

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 preZtBc(char *x, int m, int ztBc[256][256]) {
    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 - 1]][x[i]] = m - 1 - i;
}

void ZT(char *x, int m, char *y, int n) {
    int i, j, ztBc[256][256], 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) {
            cout << "Vi tri la " << j << endl;
            j += bmGs[0];
        } else
            j += max(bmGs[i], ztBc[y[j + m - 2]][y[j + m - 1]]);
    }
}

int main() {
    nhap();
    ZT(x, m, y, n);
    return 0;
}
```

- **Kiểm nghiệm thuật toán**

Input:

X = “GCAGAGAG”

Y =”GCATCGCAGAGAGTATACAGTACG”

Tiền xử lý:

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

Pha tìm kiếm

J		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	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
	X	G	C	A	G	A	G	A	**G**																
SHIFT BY ztBc[C][A] =5																									
5	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
	x						G	C	A	G	A	G	A	G											
SHIFT BY bmGs[0]=7, OUTPUT(5)																									
12	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
	X													G	C	A	G	A	**G**	A	G				
SHIFT BY 7 bmGs[6]=7 - >END																									

4. Thuật toán Berry- Ravindran

- **Đặc điểm**
  - + Đây là thuật toán kết hợp giữa thuật toán Quic search và Zhu and Takaoka.
  - + Pha chuẩn bị có độ phức tạp O(m +∂2)
  - + Pha tìm kiếm có độ phức tạp O(mn)
- **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.

    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]]

- **Code**

```
#include <iostream>
#include <cstring>

const int ASIZE = 256;
char x[100001], y[100001];
int m, n;

void nhap() {
   std::cout << "Nhap x: ";
   std::cin.getline(x, 100001);
   m = strlen(x);
   std::cout << "Nhap y: ";
   std::cin.getline(y, 100001);
   n = strlen(y);
}

void preBrBc(char *x, int m, int brBc[ASIZE][ASIZE]) {
   for (int a = 0; a < ASIZE; ++a)
      for (int b = 0; b < ASIZE; ++b)
         brBc[a][b] = m + 2;

   for (int a = 0; a < ASIZE; ++a)
      brBc[a][x[0]] = m + 1;

   for (int i = 0; i < m - 1; ++i)
      brBc[x[i]][x[i + 1]] = m - i;

   for (int 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];
```

```
    preBrBc(x, m, brBc);
    y[n + 1] = '\0';
    j = 0;

    while (j <= n - m) {
        if (memcmp(x, y + j, m) == 0)
            std::cout << "position is " << j << std::endl;

        j += brBc[y[j + m]][y[j + m + 1]];
    }
}

int main() {
    nhap();
    BR(x, m, y, n);
    return 0;
}
```

- **Kiểm nghiệm thuật toán**

Input:

X = “GCAGAGAG”

Y =”GCATCGCAGAGAGTATACAGTACG”

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

Pha tìm kiếm:

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

	X	G	C	A	**G**	A	G	A	G																
SHIFT BY brBC[G][A] =1																									
1	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
	x		**G**	C	A	**G**	A	G	A	G															
SHIFT BY brBC[A][G] =2																									
3	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
	X				**G**	C	A	G	A	G	A	G													
SHIFT BY brBC[A][G] =2																									
5	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
	X						G	C	A	G	A	G	A	G											
SHIFT BY brBC[T][A] =10, OUTPUT(5)																									
15	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
	X																**G**	C	A	G	A	G	A	G	
SHIFT BY brBC[G][0] =1																									
16	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
	X																	**G**	C	A	G	A	G	A	G
END																									

5. Thuật toán Apostollico- giancarlo

- **Đặc điểm**
  - + Đây là thuật toán cải tiến từ thuật toán Boyer – moore
  - + Pha xử lý có độ phực tạp O(m + ∂) không gian, và thời gian □
  - + Pha tìm kiếm có độ phức tạp O(n).
  - + Trường hợp xấu nhất thuật toán cần 3/2n lần phép so sánh
- **Trình bày thuật toán**
  - + Thuật toán Boyer – moore là thuật toán khó để phân tích bởi vì sau mỗi lần thử mẫu nó sẽ quên đi tất cả các kí tự đã đứng trước đó. Apostolico và Giancarlo đã thiết kế một thuật toán có thể nhớ được độ dài lớn nhất của hậu tố xâu mẫu x. Thông tin này sẽ được lưu vào một mảng skip.
  - + Trong mỗi lần thứ tại vị trí j, nếu thuật toán đã so sánh đúng các kí tự y[i+j+1 .. j+m-1] . Sẽ có trường hợp xảy ra:
    - Trường hợp 1: k > suff[i] && suff[i] = i+ 1 ; nghĩa là tồn tại x xuất hiện trong y tại vị trí j. bước dịch tiếp theo là per(x).
    - Trường hợp 2: k > suff[i] && suff[i] <= i ; nghĩa là xuất hiện vị trí sai giữa 2 kí tự x[i-suff[i]] and y[i+j-suff[i]] . bước dịch tiếp theo sẽ sử dụng bmBc[y[i+j-suff[i]]] and bmGs[i-suff[i]+1].

- Trường hợp 3: k < suff[i] .nghĩa là vị trí sai xuật hiện tại x[i-k] and y[i+j-k] . Bước dịch tiếp theo sẽ sử dụng bmBc[y[i+j-k] and bmGs[i-k+1].
- Trường hợp 4: k = suff[i]; nghĩa là ta đa có k phân tử khơp nhau. Ta sẽ nhẩy qua k phần tử này. Tiếp tục so sánh ki tự tại y[i+j-k] and x[i-k]

+ Ta nhận thấy trong tất cả các trường hợp thông tin cân thiết duy nhất là độ dài hậu tố x xuất hiện tại vị trí i.

  Aposltolico - Giancarlo sử dụng 2 cấu trúc dữ liệu:

  - 1 mảng skip có thể cập nhập sau mỗi lần thử mẫu tại vị trí j:skip[j+m-1]=max{ k: x[m-k .. m-1]=y[j+m-k .. j+m-1]}
  - một mảng suff thỏa mãn: for 1 i < msuff[i]=max{k: x[i-k+1 .. i]=x[m-k .. m-1]}

– **Code**

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <algorithm>

using namespace std;

const int MAX = 100001;

char x[MAX], y[MAX];
int m, n, XSIZE;

void nhap() {
   cout << "Nhap x: ";
   gets(x);
   m = strlen(x);
   XSIZE = m;
   cout << "Nhap y: ";
   gets(y);
   n = strlen(y);
```

```
}

void preKmp(char *x, int m, int kmpNext[]) {
   int i = 0, j = -1;
   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;
   int 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)
                printf("position is %d\n", 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];
        }
    }
}

int main() {
    nhap();
    AXAMAC(x, m, y, n);
    return 0;
}
```

– **Kiểm nghiệm thuật toán**

Input:

X = “GCAGAGAG”

Y =”GCATCGCAGAGAGTATACAGTACG”

Pha tiên xử lý xác định:

	A	C	G	T	*

bmBc	1	6	2	8	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

J		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	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
	X	G	C	A	G	A	G	A	**G**																
SHIFT BY max(bmGs[7],bmBc[A]-8+8) = 1																									
1	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
	x		G	C	A	G	A	**G**	A	G															
SHIFT BY max(bmGs[5],bmBc[C]-8+6) = 4																									
5	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
	X						G	C	A	G	A	G	A	G											
SHIFT BY bmGs[0]=7,OUTPUT(5)																									
12	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
	X													G	C	A	G	A	**G**	A	G				
SHIFT BY max(bmGs[5],bmBc[C]-8+6) = 4																									
16	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
	X																	G	C	A	G	A	G	**A**	G
END																									

III. Tìm kiếm mẫu từ vị trí cụ thể

1. Thuật toán Skip- Search

- **Đặc điểm**
  - + Sử dụng thùng chứa (bucket) các vị trí xuất hiện của kí tự trong xâu mẫu.
  - + Pha tìm kiếm có độ phức tạp thời gian O(mn).
- **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).

– **Code**

```
void SKIP(const string &x, const string &y) {
   list<int> z[ASIZE];

   // Preprocessing
   for (int i = x.length() - 1; i >= 0; --i) {
      z[x[i]].push_back(i);
   }

   // Searching
   for (int j = x.length() - 1; j < y.length(); j += x.length()) {
      for (list<int>::iterator it = z[y[j]].begin(); it != z[y[j]].end(); ++it) {
         if (memcmp(x.c_str(), y.c_str() + j - (*it), x.length()) == 0) {
            cout << "position is " << j - (*it) << endl;
         }
      }
   }
}
```

– **Kiểm nghiệm thuật toán**

Input:

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	

Pha tìm kiếm

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

2. Thuật toán Galil-Giancarlo

– Đặc điểm

+ Sàng lọc 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

– Trình bày thuật toán

+ Các thuật toán Galil-Giancarlo là một biến thể của thuật toán Colussi. 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 $0 \leq i \leq l$, 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 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.

– **Code**

```
#include <iostream>
#include <cstring>

using namespace std;

#define XSIZE 100001

void OUTPUT(int pos) {
   cout << "position is " << pos << endl;
}

int preColussi(const string &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 GG(const string &x, int m, const string &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 ki?m cho m?t ký t? l?p l?i
   for (j = ell = 0; j < n; ++j) {
      if (x[0] == y[j]) {
         ++ell;
         if (ell >= m)
            OUTPUT(j - m + 1);
      } else {
         ell = 0;
      }
   }
} else {
   // Ti?n x? lý
   nd = preColussi(x, m, h, next, shift);

   // Tìm ki?m
   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 - 1;
            }
            j += shift[i];
            i = next[i];
        }
        heavy = (j > last ? 0 : 1);
    }
  }
}

int main() {
  string x, y;
  x="a";
  y="abdaldaxnadajax";

  GG(x, x.length(), y, y.length());

  return 0;
}
```

– **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	

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

IV. Tìm kiếm mẫu từ vị trí bất kỳ

1. Thuật toán Quick Search

- **Đặc điểm**

+ Phiên bản đơn giản của Boyer-Moore
+ Chỉ sử dụng Bad-Character shift
+ Dễ thực thi
+ Độ phức tạp: O(n)
+ Có 1 bước tiền xử lý PreQS
+ Rất nhanh trong thực thi với mẫu ngắn và bảng chữ cái lớn

– **Trình bày thuật toán**

– **Code**

```
#include <iostream>
#include <string>

using namespace std;

int *PreQS(const string &x, int m) {
   int *preQS = new int[26];
   for(int i = 0; i < 26; i++) {
      preQS[i] = m - 1;
   }
   for(int i = 0; i < m; i++) {
      preQS[x[i] - 'A'] = m - i;
   }
   return preQS;
}

void QuickSearch(const string &x, int m, const string &y, int n, int
preQS[]) {
   int j = 0;
   while(j <= n - m) {
      if(x.compare(y.substr(j, m)) == 0) {
         cout << "FOUND AT " << j << endl;
      }
      j += preQS[y[j + m] - 'A'];
   }
```

```
}

int main() {
   string x = "AABA";
   string y = "AABABBAABACDCDAABAABAA";
   int *preQS = PreQS(x, 4);
   QuickSearch(x, 4, y, 22, preQS);
   delete[] preQS;
   return 0;
}
```

– **Kiểm nghiệm thuật toán**

Tiền xử lý:

X[i]	A	C	G	T	*
preBM[i]	2	7	1	9	9

j=m-1=5:

J		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	**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
	X	G	C	A	**G**	A	G	A	G																
Shift by preQS[G]=1																									
1	**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
	X		**G**	C	A	G	A	G	A	G															
Shift by preQS[A]=2																									
3	**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
	X				**G**	C	A	G	A	G	A	G													
Shift by 2																									
5	**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
	X						G	C	A	G	A	G	A	G											
Shift by 9 OUTPUT 5																									
14	**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
	X															**G**	C	A	G	A	G	A	G		
Shift by 7, END																									

2. Thuật toán Smith

– **Đặc điểm**

+ Độ phức tạp của pha tiền xử lý O(m+d)
+ Độ phức tạp pha tìm kiếm là O(m*n)

– **Trình bày thuật toán**

+ Smith nhận thấy rằng có thể tính toán được rằng số bước dịch với kí tự tiếp theo kí tự đầu mút bên phải của cửa sổ dịch chuyển đôi khi cho số bước dịch ngắn hơn việc sử dụng đúng kí tự đầu mút bên phải cửa sổ ấy. Ông ấy khuyên nên chọn max giữa 2 giá trị.
+ Pha tiền xử lí của thuật toán Smith bao gồm tính toán của hàm dịch chuyển các bad- character: preBmBc(X,m) , và hàm dịch chuyển các bad- character trong Quick Search: preQsBc(X,m)

– **Code**

```
#include <iostream>
#include <string>

using namespace std;

const int SIZE = 26;

int* preBmBc(const string &x, int m) {
   int *bmbc = new int[SIZE];
   for(int i = 0; i < SIZE; i++) {
      bmbc[i] = m;
   }
   for(int i = 0; i < m - 1; i++) {
      bmbc[x[i] - 'A'] = m - 1 - i;
   }
   return bmbc;
}

int* preQsBc(const string &x, int m) {
   int *bmbc = new int[SIZE];
   for(int i = 0; i < SIZE; i++) {
      bmbc[i] = m + 1;
   }
   for(int i = 0; i < m; i++) {
```

```
        bmbc[x[i] - 'A'] = m - i;
    }
    return bmbc;
}

int max(int x, int y) {
    return x > y ? x : y;
}

void search(const string &x, int m, const string &y, int n, int *preBmBc,
int *preQsBc) {
    int i = 0;
    while(i <= n - m) {
        if(x.compare(y.substr(i, m)) == 0) {
            cout << "FOUND AT: " << i << endl;
        }
        i += max(preBmBc[y[i + m - 1] - 'A'], preQsBc[y[i + m] - 'A']);
    }
}

int main() {
    string x = "GCAGAGAG";
    string y =
"GCATCGCAGAGAGTATACAGTACGGCAGAGAGGCAGAGAGG
CAGAGA";
    int *bmbc = preBmBc(x, x.length());
    int *qsbc = preQsBc(x, x.length());
    search(x, x.length(), y, y.length(), bmbc, qsbc);
    delete[] bmbc;
    delete[] qsbc;
    return 0;
}
```

– **Kiểm nghiệm thuật toán**

Input:

x=”GCAGAGAG” m=8;
y=”GCAGAGAGGCGCAGAGAGGGGG” n=22;

Tiền xử lý

X	A	C	G	*
preBmBc[i]	1	6	2	8
preQsBc[i]	2	7	1	9

J		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	Y	**G**	**C**	**A**	**G**	**A**	**G**	**A**	**G**	G	C	G	C	A	G	A	G	A	G	G	G	G	G
	X	**G**	**C**	**A**	**G**	**A**	**G**	**A**	**G**														
Shift by 2 (preBmBc[G]=2,preQsBc[G]=2), OUTPUT(0)																							
2	Y	G	C	**A**	G	A	G	A	G	G	C	G	C	A	G	A	G	A	G	G	G	G	G
	X			**G**	C	A	G	A	G	A	G												
Shift by 6 (preBmBc[C]=6,preQsBc[G]=2)																							
8	Y	G	C	A	G	A	G	A	G	**G**	**C**	G	C	A	G	A	G	A	G	G	G	G	G
	X									**G**	**C**	A	G	A	G	A	G						
Shift by 2 (preBmBc[G]=2,preQsBc[A]=1)																							
10	Y	G	C	A	G	A	G	A	G	G	C	**G**	**C**	**A**	**G**	**A**	**G**	**A**	**G**	G	G	G	G
	X											**G**	**C**	**A**	**G**	**A**	**G**	**A**	**G**				
Shift by 2 (preBmBc[G]=2,preQsBc[G]=2) , OUTPUT(10)																							
12	Y	G	C	A	G	A	G	A	G	G	C	G	C	**A**	G	A	G	A	G	G	G	G	G
	X													**G**	C	A	G	A	G	A	G		
Shift by 2 (preBmBc[G]=2,preQsBc[G]=2)																							
14	Y	G	C	A	G	A	G	A	G	G	C	G	C	A	G	**A**	G	A	G	G	G	G	G
	X															**G**	C	A	G	A	G	A	G
END																							

3. Thuật toán Raita

- **Đặc điểm**
  + Đầu tiên so sánh kí tự cuối cùng của xâu mẫu, sau đó là kí tự đầu tiên và kí tự giữa trước khi thực thi lệnh so sánh khác.
  + Pha cài đặt có độ phức tạp thuật toán là O(m+σ) và độ phức tạp bộ nhớ là O(σ)
  + Pha thực thi có độ phức tạp là O(mxn)

- **Trình bày thuật toán**
  + Raita thiết kế một thuật toán mà với mỗi lần duyệt, đầu tiên đem so sánh kí tự cuối cùng của mẫu với kí tự đầu mút bên phải của cửa sổ, sau đó nếu chúng trùng nhau, so sánh kí tự đầu tiên của mẫu với kí tự đầu mút bên trái của cửa sổ. Nếu chúng trùng nhau, tiếp tục so sánh kí tự giữa của mẫu với kí tự giữa của xâu văn bản đang trong cửa sổ duyệt. Nếu chúng vẫn trùng nhau, sẽ đem so sánh các kí tự khác trong xâu mẫu bắt đầu vị trí thứ 2 tới vị trí cuối cùng(so sánh lại kí tự ở giữa một lần nữa).
- **Code**

```
const int SIZE = 26;


int *preBmBc(const string &x, int m) {
   int *bmbc = new int[SIZE];
   for(int i = 0; i < SIZE; i++) {
      bmbc[i] = m;
   }
   for(int i = 0; i < m - 1; i++) {
      bmbc[x[i] - 'A'] = m - 1 - i;
   }
   return bmbc;
}


void raitaSearch(const string &x, int m, const string &y, int n, int
*bmbc) {
   char first = x[0], last = x[m - 1], middle = x[m / 2];
```

```
    int i = 0;
    while(i <= n - m) {
        if(last == y[i + m - 1] && first == y[i] && middle == y[i + m / 2])
{
            if(x.substr(1, m - 2) == y.substr(i + 1, m - 2)) {
                cout << "FOUND AT: " << i << endl;
            }
        }
        i += bmbc[y[i + m - 1] - 'A'];
    }
}

int main() {
    string x = "GCAGAGAG";
    string y = "GCAGAGAGGCGCAGAGAGGGGG";
    int *bmbc = preBmBc(x, x.length());
    raitaSearch(x, x.length(), y, y.length(), bmbc);
    delete[] bmbc;
    return 0;
}
```

– **Kiểm nghiệm thuật toán**

Input:

x=”GCAGAGAG” m=8;
y=”GCAGAGAGGCGCAGAGAGGGGG” n=22;

Tiền xử lý

X	A	C	G	*
preBmBc[i]	1	6	2	8

J		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	Y	**G**	**C**	**A**	**G**	**A**	**G**	**A**	**G**	G	C	G	C	A	G	A	G	A	G	G	G	G	G
	X	**G**	**C**	**A**	**G**	**A**	**G**	**A**	**G**														
Shift by 2 (preBmBc[G]=2) OUTPUT(0)																							
2	Y	G	C	A	G	A	G	A	G	G	**C**	G	C	A	G	A	G	A	G	G	G	G	G
	X			G	C	A	G	A	G	A	**G**												
Shift by 6 (preBmBc[C]=6)																							
8	Y	G	C	A	G	A	G	A	G	**G**	C	G	C	A	G	A	**G**	A	G	G	G	G	G
	X									**G**	C	A	G	A	G	A	**G**						
Shift by 2(preBmBc[G]=2)																							
10	Y	G	C	A	G	A	G	A	G	G	C	**G**	***C***	***A***	***G***	***A***	***G***	***A***	**G**	G	G	G	G
	X											**G**	***C***	***A***	***G***	***A***	***G***	***A***	**G**				
Shift by 2 (preBmBc[G]=2) , OUTPUT(10)																							
12	Y	G	C	A	G	A	G	A	G	G	C	G	C	A	G	A	G	A	G	G	**G**	G	G
	X													G	C	A	G	A	G	A	**G**		
Shift by 2 (preBmBc[G]=2)																							
14	Y	G	C	A	G	A	G	A	G	G	C	G	C	A	G	A	G	A	G	G	G	G	**G**
	X															G	C	A	G	A	G	A	**G**
END																							

4. Thuật toán HorsePool

– **Đặc điểm**

+ Là thuật toán đơn giản hơn của Boyer Moore
+ Sử dụng dụng bad- character Dễ để cài đặt
+ Pha cài đặt có độ phức tạp thuật toán là O(m+σ) và độ phức tạp bộ nhớ là O(σ)

+ Pha thực thi có độ phức tạp là O(mxn)
+ Số lượng so sánh trung bình cho một văn bản là trong khoảng 1/ và 2/(+1)

– **Trình bày thuật toán**

+ Quy tắc dịch bad- character được sử dụng trong Boyer Moore không có hiệu quả trong đoạn văn bản kí tự nhỏ, nhưng khi đoạn văn bản là lớn và phải so sánh với mẫu dài, nó thường trong trường hợp bảng bã ASCII và tìm kiếm thông thường trong soạn thảo văn bản, chúng rất hữu dụng. Sử dụng nó như những kết quả riêng biệt lại rất hiệu quả.

– **Code**

```
const int SIZE = 26;

int *preBmBc(const char *x, int m) {
   int *bmbc = new int[SIZE];
   for (int i = 0; i < SIZE; i++) {
      bmbc[i] = m;
   }
   for (int i = 0; i < m - 1; i++) {
      bmbc[x[i] - 'A'] = m - 1 - i;
   }
   return bmbc;
}

void horsePoolSearch(const char *x, int m, const char *y, int n, int
*bmbc) {
   int i = 0;
   while (i <= n - m) {
      char c = y[i + m - 1];
      if (c == x[m - 1] && memcmp(x, y + i, m - 1) == 0) {
         cout << "FOUND AT: " << i << endl;
      }
      i += bmbc[y[i + m - 1] - 'A'];
   }
}
```

```
int main() {
    const char *x = "GCAGAGAG";
    const char *y = "GCAGAGAGGCGCAGAGAGGGGG";
    int *bmbc = preBmBc(x, strlen(x));
    horsePoolSearch(x, strlen(x), y, strlen(y), bmbc);
    delete[] bmbc; // Don't forget to free memory allocated with new
    return 0;
}
```

– **Kiểm nghiệm thuật toán**

Input:

x=”GCAGAGAG” m=8;
y=”GCAGAGAGGCGCAGAGAGGGGG” n=22;

Tiền xử lý

X	A	C	G	*
preBmBc[i]	1	6	2	8

J		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	Y	***G***	***C***	***A***	***G***	***A***	***G***	***A***	**G**	G	C	G	C	A	G	A	G	A	G	G	G	G	G
	X	***G***	***C***	***A***	***G***	***A***	***G***	***A***	**G**														
Shift by 2 (preBmBc[G]=2) OUTPUT(0)																							
2	Y	G	C	A	G	A	G	A	G	G	**C**	G	C	A	G	A	G	A	G	G	G	G	G
	X			G	C	A	G	A	G	A	**G**												
Shift by 6 (preBmBc[C]=6)																							
8	Y	G	C	A	G	A	G	A	G	***G***	***C***	G	C	A	G	A	**G**	A	G	G	G	G	G
	X									***G***	***C***	A	G	A	G	A	**G**						
Shift by 2(preBmBc[G]=2)																							
10	Y	G	C	A	G	A	G	A	G	G	C	***G***	***C***	***A***	***G***	***A***	***G***	***A***	**G**	G	G	G	G
	X											***G***	***C***	***A***	***G***	***A***	***G***	***A***	**G**				
Shift by 2 (preBmBc[G]=2) , OUTPUT(10)																							
12	Y	G	C	A	G	A	G	A	G	G	C	G	C	A	G	A	G	A	G	G	**G**	G	G
	X													G	C	A	G	A	G	A	**G**		
Shift by 2 (preBmBc[G]=2)																							
14	Y	G	C	A	G	A	G	A	G	G	C	G	C	A	G	A	G	A	G	G	G	G	**G**
	X															G	C	A	G	A	G	A	**G**

END		