Mục lục

[I. Các thuật toán tìm kiếm mẫu từ trái qua phải 2](#_Toc45386938)

[1. Thuật toán Brute Force 2](#_Toc45386939)

[1.1 Đặc điểm 2](#_Toc45386940)

[1.2 Trình bày thuật toán 2](#_Toc45386941)

[1.3 Kiểm thử thuật toán 2](#_Toc45386942)

[1.4 Lập trình theo thuật toán 5](#_Toc45386943)

[2. Thuật toán Shift Or 6](#_Toc45386944)

[2.1 Đặc điểm 6](#_Toc45386945)

[2.2 Trình bày thuật toán 6](#_Toc45386946)

[2.3 Kiểm nghiệm thuật toán 7](#_Toc45386947)

[2.4 Lập trình theo thuật toán 8](#_Toc45386948)

[3. Thuật toán Morris-Pratt 10](#_Toc45386949)

[3.1 Đặc điểm 10](#_Toc45386950)

[3.2 Trình bày thuật toán 10](#_Toc45386951)

[3.3 Kiểm thử thuật toán 11](#_Toc45386952)

[3.4 Lập trình theo thuật toán 13](#_Toc45386953)

[4. Thuật toán Knuth-Morris-Pratt 15](#_Toc45386954)

[4.1 Đặc điểm 15](#_Toc45386955)

[4.2 Trình bày thuật toán 15](#_Toc45386956)

[4.3 Kiểm thử thuật toán 16](#_Toc45386957)

[4.4 Lập trình theo thuật toán 18](#_Toc45386958)

[II. Các thuật toán tìm kiếm mẫu từ phải qua trái 20](#_Toc45386959)

[1. Thuật toán Boyer-Moore 20](#_Toc45386960)

[1.1 Đặc điểm 20](#_Toc45386961)

[1.2 Trình bày thuật toán 20](#_Toc45386962)

[1.3 Kiểm thử thuật toán 23](#_Toc45386963)

[1.4 Lập trình theo thuận toán 24](#_Toc45386964)

[2. Thuật toán Zhu Kataoka 26](#_Toc45386965)

[2.1 Đặc điểm 26](#_Toc45386966)

[2.2 Trình bày thuật toán 27](#_Toc45386967)

[2.3 Kiểm thử thuật toán 27](#_Toc45386968)

[2.4 Lập trình theo thuật toán 28](#_Toc45386969)

[3. Thuật toán Berry-Ravindran 31](#_Toc45386970)

[3.1 Đặc điểm 31](#_Toc45386971)

[3.2 Trình bày thuật toán 31](#_Toc45386972)

[3.3 Kiểm nghiệm thuật toán 32](#_Toc45386973)

[3.4 Lập trình theo thuật toán 33](#_Toc45386974)

[4. Thuật toán Turbo-BM 35](#_Toc45386975)

[4.1 Đặc điểm 35](#_Toc45386976)

[4.2 Trình bày thuật toán 35](#_Toc45386977)

[4.3 Kiểm nghiệm thuật toán 36](#_Toc45386978)

[4.4 Lập trình theo thuật toán 37](#_Toc45386979)

[III. Các thuật toán tìm kiếm mẫu từ vị trí cụ thể 40](#_Toc45386980)

[1. Thuật toán Colussi 40](#_Toc45386981)

[1.1 Đặc điểm 40](#_Toc45386982)

[1.2 Trình bày thuật toán 40](#_Toc45386983)

[1.3 Kiểm nghiệm thuật toán 41](#_Toc45386984)

[1.4 Lập trình theo thuật toán 43](#_Toc45386985)

[2. Thuật toán Skip Search 46](#_Toc45386986)

[2.1 Đặc điểm 46](#_Toc45386987)

[2.2 Trình bày thuật toán 46](#_Toc45386988)

[2.3 Kiểm thử thuật toán 47](#_Toc45386989)

[2.4 Lập trình theo thuật toán 48](#_Toc45386990)

[IV. Các thuật toán tìm kiếm mẫu từ vị trí bất kỳ 49](#_Toc45386991)

[1. Thuật toán Horspool 49](#_Toc45386992)

[1.1 Đặc điểm 49](#_Toc45386993)

[1.2 Trình bày thuật toán 50](#_Toc45386994)

[1.3 Kiểm thử thuật toán 50](#_Toc45386995)

[1.4 Lập trình theo thuật toán 52](#_Toc45386996)

[2. Thuật toán Quick Search 53](#_Toc45386997)

[2.1 Đặc điểm 53](#_Toc45386998)

[2.2 Trình bày thuật toán 54](#_Toc45386999)

[2.3 Kiểm thử thuật toán 54](#_Toc45387000)

[2.4 Lập trình theo thuật toán 55](#_Toc45387001)

[3. Smith algorithm 57](#_Toc45387002)

[3.1 Đặc điểm 57](#_Toc45387003)

[3.2 Trình bày thuật toán 57](#_Toc45387004)

[3.3 Kiểm thử thuật toán 57](#_Toc45387005)

[3.4 Lập trình theo thuật toán 59](#_Toc45387006)

# Các thuật toán tìm kiếm mẫu từ trái qua phải

## Thuật toán Brute Force

## 1.1 Đặc điểm

* Không có pha chuẩn bị
* 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)

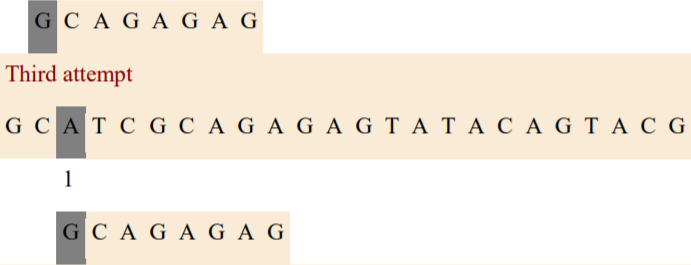
## 1.2 Trình bày thuật toán

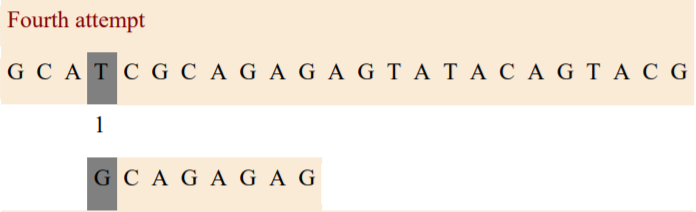
* + - Thuật toán Brute Force bao gồm kiểm tra, tất cả các vị trí trong đoạn văn bản giữa 0 và n-m, không cần quan tâm liệu mẫu này có tồn tại ở vị trí đó hay không. Sau đó, sau mỗi lần kiểm tra mẫu sẽ dịch sang phải một vị trí.
    - Thuật toán Brute Force không cần đến pha chuẩn bị.

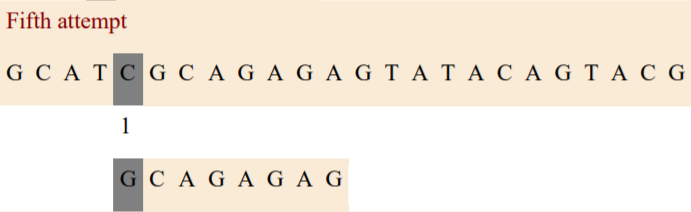
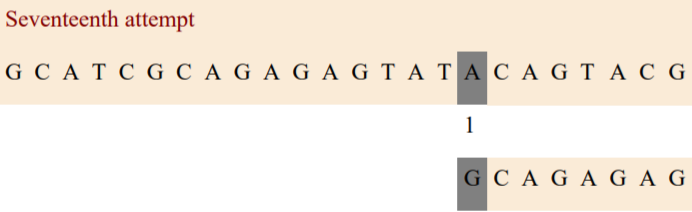
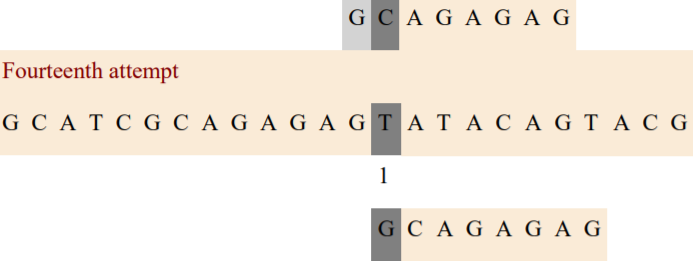
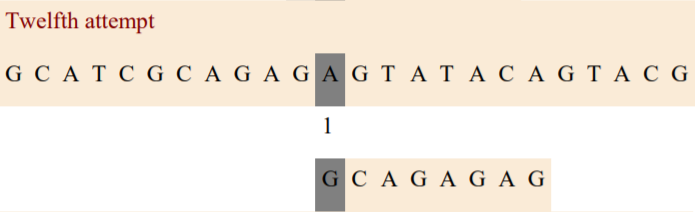
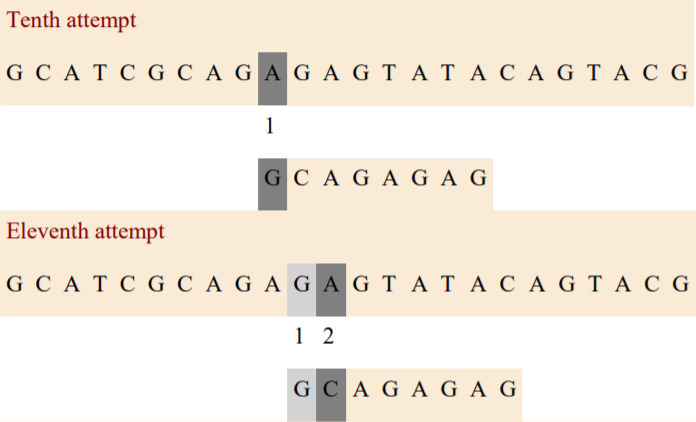
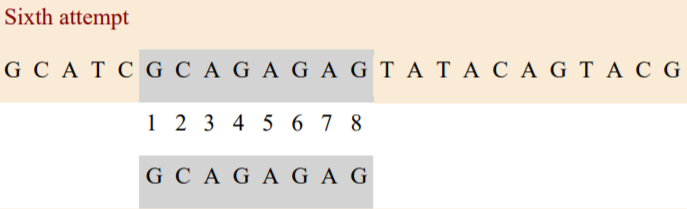
## 1.3 Kiểm thử thuật toán

X=“GCAGAGA”

Y =”GCATCGCAGAGAGTATACAGTACG”

Qúa trình tìm kiếm



Brute Force cần 30 lần so sánh kí tự trong trường hợp này

## 1.4 Lập trình theo thuật toán

|  |
| --- |
| #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;  void nhap(){  printf("Nhap x: "); gets(x); m = strlen(x);  printf("Nhap y: "); gets(y); n = strlen(y);  }  void BF(char \*x, int m, char \*y, int n) {  int i, j;  for (j = 0; j <= n - m; ++j) {  for (i = 0; i < m && x[i] == y[i + j]; ++i);  if (i >= m)  printf("position is %d\n",j);  }  }  main(){  nhap();  BF(x,m,y,n);  } |

Thử nghiệm:

Input:

* Nhập x: GCAGAGAG
* Nhập y: GCATCGCAGAGAGTATACAGTACG

Output:

* Vị trí là 5

## Thuật toán Shift Or

## 2.1 Đặc điểm

* Thuật toán sử dụng công nghệ bitwise
* Thuật toán đạt hiệu quả nếu độ dài mẫu không vượt quá kích thước bộ nhớ máy.
* Pha xử lý có độ phức tạp O(m +∂)
* Pha tìm kiếm có độ phức tạp O(n) . nó còn phụ thuộc vào kích thước bảng chữ cái và đồ dài mẫu.

## 2.2 Trình bày thuật toán

* Thuật toán Shift Or sử dụng công nghệ bitwise. Ta có R là mảng bit độ dài là m. vector Rj là giá trị của mảng R sau vị trí y[j] đã xử lý.
* Nó chứa các thông tin về tất cả các tiền tố x đã phù hợp tại vị trí j hay chưa

for 0 < i <= m-1

* Vector Rj+1 có thể xác định sau khi có Rj



* Nếu Rj+1[m-1] = 0 . có nghĩa là ta đã hoàn thành so sánh và tìm được vị trí x xuất hiện trong y. Với mỗi kí tự c ta có Sc là mảng bít độ dài m:

for 0 <= i <m-1 , Sc [i] =0 if x[i] = c;

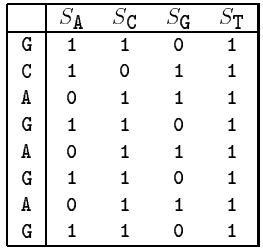
* Mảng Sc xác định vị trí của kí tự c trong mẫu x. mỗi Sc sẽ được chuẩn bị trước. Ta có thể xác ịnh nhanh Rj+1 : *Rj*+1=*SHIFT*(*Rj*) *OR Sy*[*j*+1]

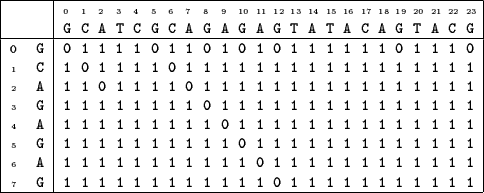
## 2.3 Kiểm nghiệm thuật toán

X = “GCAGAGAG”

Y =”GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lý xác định:



 Pha tìm kiếm thực hiện qua bảng sau:

Tại vị trí 12 ta có R12 [7] = 0. Nghĩa là có x xuất hiện trong y tại vị trí 12-8+1=5

## 2.4 Lập trình theo thuật toán

|  |
| --- |
| #include<bits/stdc++.h>  using namespace std;  char x[101], y[1000001];  int m, n, ASIZE = 256;  void nhap(){      printf("Nhap x: ");      gets(x);      m = strlen(x);      printf("Nhap y: ");      gets(y);      n = strlen(y);  }  int preSo(char \*x, int m, unsigned int S[])  {      unsigned int j, lim;      int i;      for (i = 0; i < ASIZE; ++i)          S[i] = ~0;      for (lim = i = 0, j = 1; i < m; ++i, j <<= 1)      {          S[x[i]] &= ~j;          lim |= j;      }      lim = ~(lim >> 1);      return (lim);  }  void SO(char \*x, int m, char \*y, int n)  {      unsigned int lim, state;      unsigned int S[ASIZE];      int j;      // if (m > WORD)      // error("SO: Use pattern size <= word size");      /\* Preprocessing \*/      lim = preSo(x, m, S);      /\* Searching \*/      for (state = ~0, j = 0; j < n; ++j)      {          state = (state << 1) | S[y[j]];          if (state < lim)              printf("position is %d\n", j - m + 1);      }  }  main()  {      nhap();      SO(x, m, y, n);  } |

**Thử nghiệm:**

Input:

* Nhập x: GCAGAGAG
* Nhập y: GCATCGCAGAGAGTATACAGTACG

Output:

* Position is 5

## Thuật toán Morris-Pratt

## 3.1 Đặc điểm

* Thực hiện so sánh từ trái qua phải
* Pha tiền xử lí có độ phức tạp O(m)
* Pha thực thi có độ phức tạp O(m+n) phụ thuộc vào kích thước của mảng chữ cái
* Thực hiện nhiều nhất 2n-1 các so sánh kí tự trong pha thực thi

## 3.2 Trình bày thuật toán

* Thiết kế của thuật toán Morris Partt là thuật toán Brute Force với các ràng buộc chặt chẽ hơn, sử dụng những thông tin thu thập được trong quá trình quét văn bản.
* Cùng nhìn lại thuật toán Brute Force, chúng có thể tăng độ dài bước dịch và đồng thời nhớ một vài phần của văn bản, phần mà nó chiếu với mẫu. nó sẽ hạn chế số lần so sánh giữa các kí tự trong xâu mẫu và xâu văn bản và tăng tốc độ thực hiện.
* neqQuan 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* . 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. độ dài lớn nhất của v được gọi là biên của u. mảng mpNext[i] là độ dài lớn nhất của biên của x[0…i-1] 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 mpNext[0] được thiết lập là -1.

## 3.3 Kiểm thử thuật toán

X = “GCAGAGAG”

Y =”GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lí chuẩn bị bảng mpNext:

Morris-Pratt mpNext table

Bảng *mpNext*

Pha tìm kiếm thực hiện:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Nineth 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)

## 3.4 **Lập trình theo thuật toán**

|  |
| --- |
| #include<bits/stdc++.h>  using namespace std;  void preMp(char \*x, int m, int mpNext[])  {      int i, j;      i = 0;      j = mpNext[0] = -1;      while (i < m)      {          while (j > -1 && x[i] != x[j])              j = mpNext[j];          mpNext[++i] = ++j;      }  }  void MP(char \*x, char \*y)  {      int m = strlen(x);      int n = strlen(y);      int i, j, mpNext[m + 1];      preMp(x, m, mpNext);      i = j = 0;      int d = 0;      while (j < n)      {          //  printf("count: %d\n",++d); int dd = 0;          while (i > -1 && x[i] != y[j])          {              i = mpNext[i];              //  printf("count name: %d\n",++dd);          }          i++;          j++;          if (i >= m)          {              printf("position is %d \n", j - i);              i = mpNext[i];          }      }  }  main()  {      char x[1000010], y[1000010];      printf("nhap x: ");      gets(x);      printf("nhap y: ");      gets(y);      MP(x, y);  } |

**Thử nghiệm:**

Input:

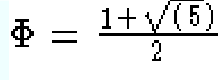
* Nhập x: GCAGAGAG
* Nhập y: GCATCGCAGAGAGTATACAGTACG

Output:

* Vị trí là 5

## Thuật toán Knuth-Morris-Pratt

## 4.1 Đặc điểm

* Thực hiện so sánh từ trái qua phải
* Pha tiền xử lí có độ phức tạp thuật toán là O(m)
* Pha thực thi có độ phức tạp O(m+n) phụ thuộc vào kích thước của mảng chữ cái
* Thực hiện nhiều nhất 2n-1 các so sánh kí tự trong pha thực thi
* Độ trễ biên được tính bằng logɸ(m) với ɸ được tính là 

## 4.2 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 .
* 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 .

## 4.3 Kiểm thử thuật toán

X = “GCAGAGAG”

Y =”GCATCGCAGAGAGTATACAGTACG”

Knuth-Morris-Pratt kmpNext tablePha tiền xử lý xác định:

bảng kmpNext

Pha tìm kiếm thực hiện như sau:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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)

Knutt Morris Partt sử dụng 18 lần so sánh kí tự trong ví dụ này.

## 4.4 **Lập trình theo thuật toán**

|  |
| --- |
| #include<bits/stdc++.h>  using namespace std;  const int Max = 1000010;  int m, n, kmp[Max];  char a[Max], b[Max];  void preKMP()  {  int i = 0, j = kmp[0] = -1;  while (i < m)  {  while (j > -1 && b[i] != b[j])  j = kmp[j];  i++;  j++;  if (b[i] == b[j])  kmp[i] = kmp[j];  else  kmp[i] = j;  }  }  main()  {  printf("Nhap x: ");  gets(b);  printf("Nhap y: ");  gets(a);  n = strlen(a);  m = strlen(b);  preKMP();  int i = 0, j = 0, found = 0;  while (j < n)  {  while (i > -1 && b[i] != a[j])  i = kmp[i];  i++;  j++;  if (i >= m)  {  if (found)  putchar(' ');  printf("position is %d\n", j - i);  i = kmp[i];  found = 1;  }  }  putchar('\n');  } |

# Các thuật toán tìm kiếm mẫu từ phải qua trái

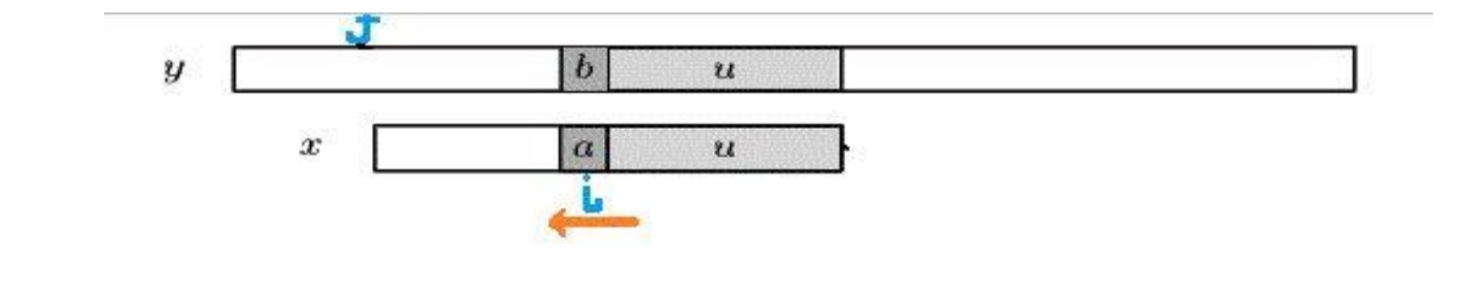
## Thuật toán Boyer-Moore

## 1.1 Đặc điểm

* Thuật toán thực thi việc so sánh từ phải sang trái
* Pha cài đặt có độ phức tạp thuật toán và không gian nhớ là O(m+σ)
* Pha thực thi có độ phức tạp là O(mxn)
* 3n kí tự xâu văn bản được so sánh trong trường hợp xấu nhất khi thực thi với mẫu không tuần hoàn
* Độ phức tạp O(n/m) khi thực thi tốt nhất.

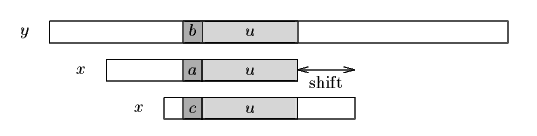
## 1.2 Trình bày thuật toán

* Boyer-Moore quan tâm tới thuật toán đối sánh xâu hiệu quả nhất thường thấy. Các biến thể của nó thường được dùng trong các bộ soạn thảo cho các lệnh như <<search>> và <<subtitute>>.
* Thuật toán sẽ quét các ký tự của mẫu (pattern) từ phải sang trái bắt đầu ở phần tử cuối cùng.
* Trong trường hợp mis-match (hoặc là trường hợp đã tìm được 1 đoạn khớp với mẫu), nó sẽ dùng 2 hàm được tính toán trước để dịch cửa sổ sang bên phải. Hai hàm dịch chuyển này được gọi là **good-suffix shift** (còn được biết với cái tên phép dịch chuyển khớp) và **bad-character shift** (còn được biết với cái tên phép dịch chuyển xuất hiện).
* Đối với mẫu x[0..m-1] ta dùng 1 biến chỉ số i chạy từ cuối về đầu, đối với chuỗi y[0..n-1] ta dùng 1 biến j để chốt ở phía đầu. G/s rằng trong quá trình so sánh ta gặp 1 mis-match tai vị trí x[i]=a của mẫu và y[i+j]=b trong khi đang thử khớp tại vị trí j.



* Khi đó, x[i+1..m-1] = y[j+i+1..j+m-1] = u và x[i] ≠ y[i+j] . Bây giờ ta đi xét xem đối với từng trường hợp, 2 hàm trên sẽ thực hiện việc dịch chuyển như thế nào:

Phép dịch chuyển good-suffix shift sẽ dịch cửa sổ sang bên phải cho đến khi gặp 1 ký tự khác với x[i] trong trường hợp đoạn u lại xuất hiện trong x.



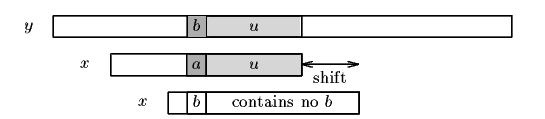
**Fig2**: good-suffix shift, trường hợp u lại xuất hiện trong x

* Nếu đoạn u không xuất hiện lại trong x, mà chỉ có 1 phần cuối (suffix) của u khớp với phần đầu (prefix) của x, thì ta sẽ dịch 1 đoạn sao cho phần suffix dài nhất v của y[j+i+1..j+m-1] khớp với prefix của x.



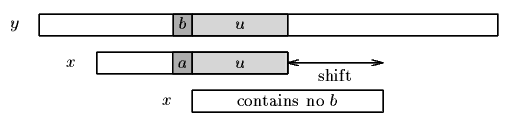
**Fig3:** good-suffix shift, trường hợp chỉ suffix của u xuất hiện trong x

Phép dịch chuyển bad-character shift sẽ khớp kí tự y[i+j] với 1 ký tự (bên phải nhất) trong đoạn x[0..m-2]



**Fig4**: bad-character shift

Nếu y[i+j] không xuất hiện trong x, ta thấy ngay rằng không có xuất hiện nào của x trong y mà lại chứa chấp y[i+j], do đó ta có thể đặt cửa sổ ngay sau y[i+j], tức là y[j+i+1] .



Thuật toán Boyer-Moore sẽ chọn đoạn dịch chuyển dài nhất trong 2 hàm dịch chuyển good-suffix shift và bad-character shift. Hai hàm này được định nghĩa như sau:

- Hàm good-suffix shift được lưu trong bảng bmGs có kích thước m+1.

Ta định nghĩa 2 điều kiện sau:

1. *Cs(i, s): với mỗi k mà i < k < m, s ≥ k hoặc x[k-s]=x[k] và*

2. *Co(i, s): nếu s <i thì x[i-s] ≠ x[i]*

Khi đó, với 0≤ i <m: *bmGs*[i+1]=min{s>0: *Cs*(i, s) and *Co*(i, s) hold} và chúng ta định nghĩa *bmGs*[0] là độ dài chu kỳ của x. Việc tính toán bảng bmGs sử dụng 1 bảng suff định nghĩa như sau*: với 1 ≤ i < m, suff[i]=max{k: x[i-k+1 ..i]=x[m-k .. m-1]}*

- Hàm bad-character shift được lưu trong bảng *bmBc* có kích thước σ. Cho c trong Σ: *bmBc[c] = min{i: 1≤ i <m-1 và x[m-1-i]=c}* nếu c xuất hiện trong x, m ngược lại.

Bảng bmGs và bmBc được tính toán trong thời gian O(m+σ) trước khi thực hiện tìm kiếm và cần 1 không gian phụ là O(m+σ). 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 am-1 b trong bn

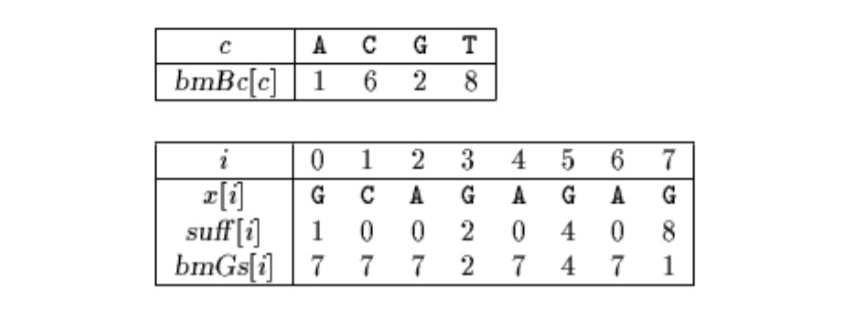
chuỗi thuật toán chỉ sử dụng O(n/m) phép so sánh, đây được coi là “cảnh giới” cho bất cứ một thuật toán tìm kiếm chuỗi nào mà mẫu đã được xử lý trước.

## 1.3 Kiểm thử thuật toán

X = “GCAGAGAG”

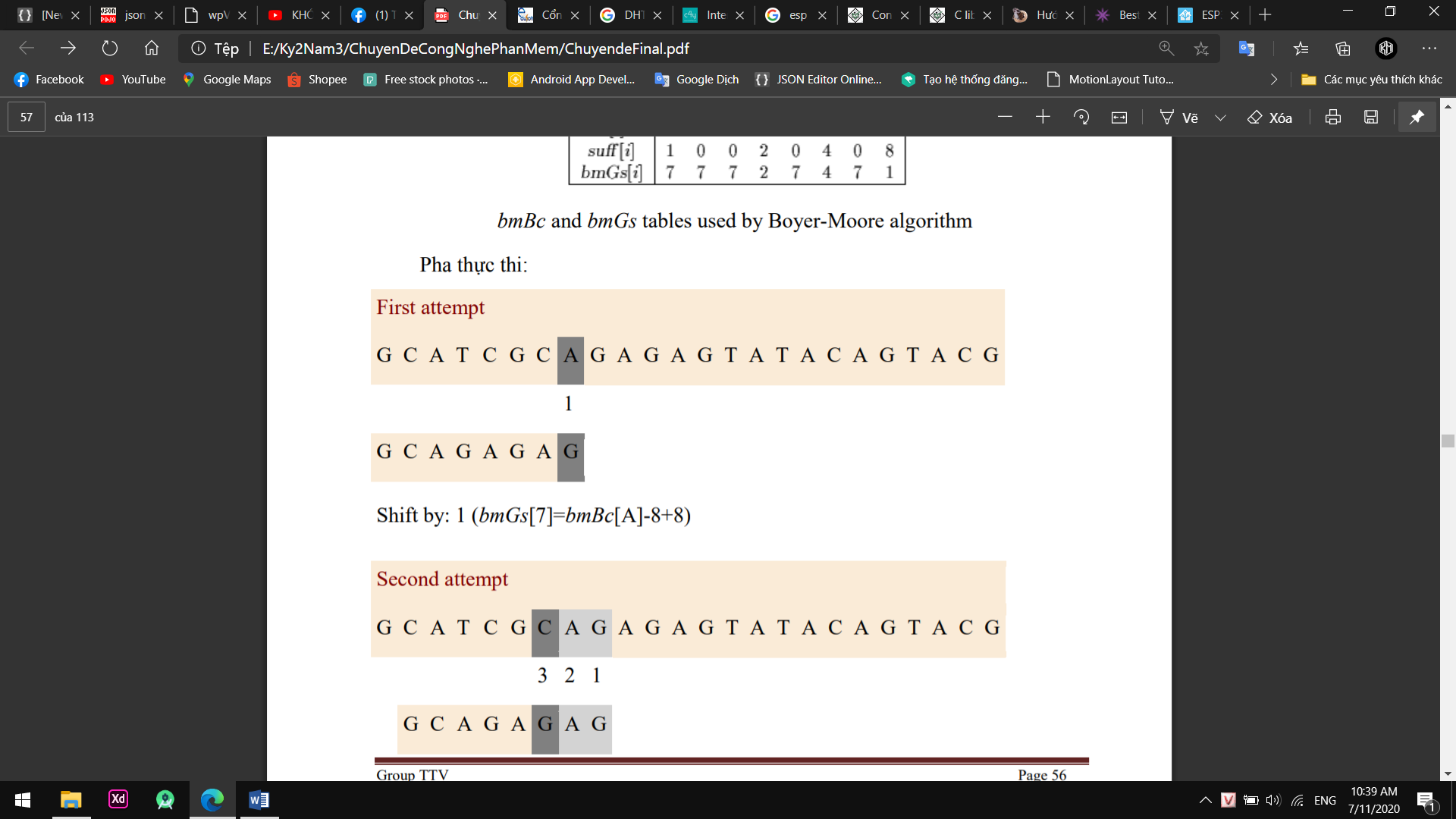
Y =”GCATCGCAGAGAGTATACAGTACG”

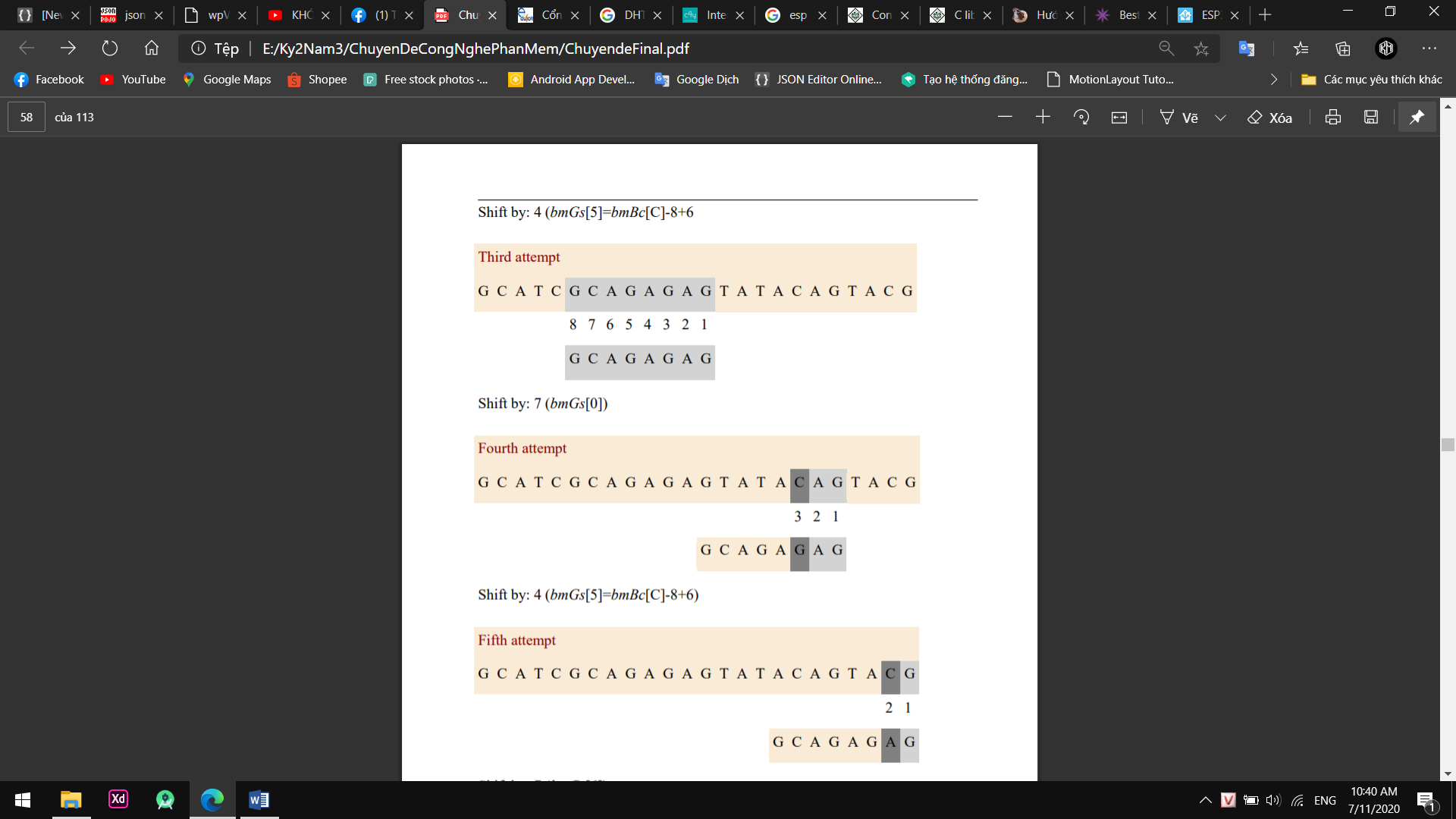
* Pha tiền xử lí:



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

* Pha thực thi





Shift by: 7 (bmGs[6])

Boyer Moore thực thi 17 lần so sánh kí tự trong ví dụ này.

## 1.4 Lập trình theo thuận toán

#include<iostream>

#include<algorithm>

#include<iomanip>

#include<cstdio>

#include<cstring>

#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(){

printf("Nhap x: "); gets(x); m = strlen(x); XSIZE = m;

printf("Nhap y: "); gets(y); 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 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[]) {

nt i, j, suff[XSIZE];

suffixes(x, m, suff);

for (i = 0; i < m; ++i)

bmGs[i] = m;

j = 0;

for (i = m - 1; i >= 0; --i) if (suff[i] == i + 1)

for (; j < m - 1 - i; ++j)

if (bmGs[j] == m)

bmGs[j] = m - 1 - i;

for (i = 0; i <= m - 2; ++i)

bmGs[m - 1 - suff[i]] = m - 1 - i;

}

void BM(char \*x, int m, char \*y, int n) {

int i, j, bmGs[XSIZE], bmBc[ASIZE];

/\* Preprocessing \*/

preBmGs(x, m, bmGs);

preBmBc(x, m, bmBc);

/\* Searching \*/

j = 0;

while (j <= n - m) {

for (i = m - 1; i >= 0 && x[i] == y[i + j]; --i);

if (i < 0) {

printf("position is %d\n",j);

j += bmGs[0];

}

else

j += max(bmGs[i], bmBc[y[i + j]] - m + 1 + i);

}

}

main(){

nhap();

BM(x,m,y,n);

}

**Thử nghiệm:**

Input:

* Nhập x: GCAGAGAG
* Nhập y: GCATCGCAGAGAGTATACAGTACG

Output:

* Vị trí là 5

## Thuật toán Zhu Kataoka

## 2.1 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+σ)
* Pha thực thi có độ phức tạp là O(mxn)

## 2.2 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]

or

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

or

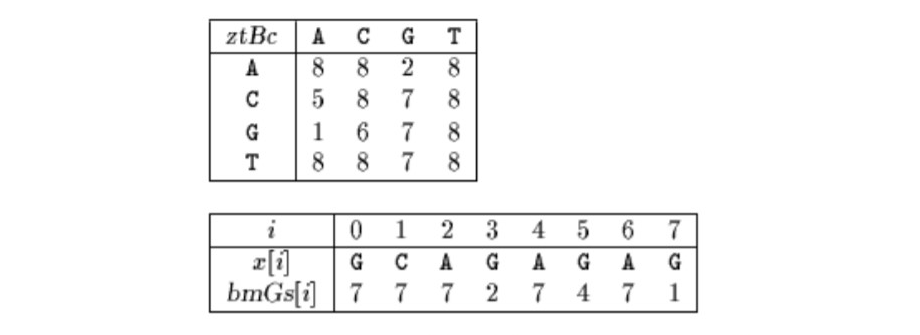
k=m and x[0] b và ab không xuất hiện trong đoạn x[0 .. m-2]

## 2.3 Kiểm thử thuật toán

X = “GCAGAGAG”

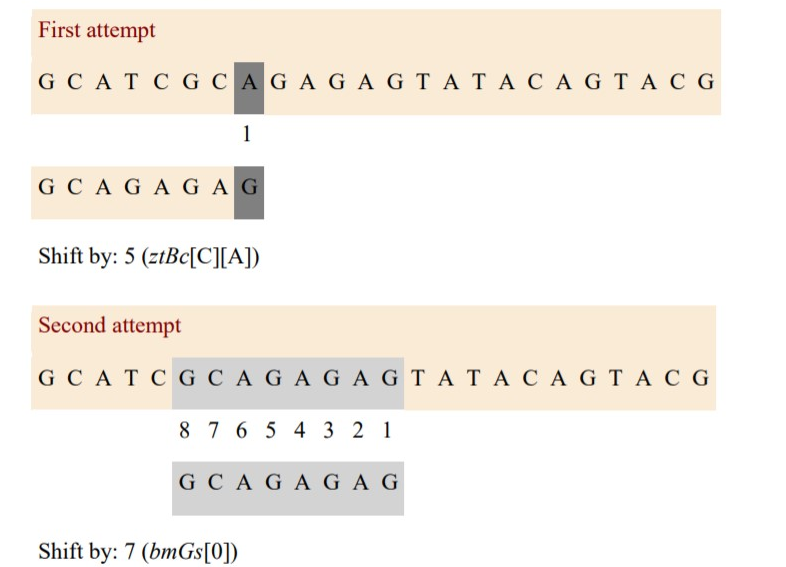
Y =”GCATCGCAGAGAGTATACAGTACG”

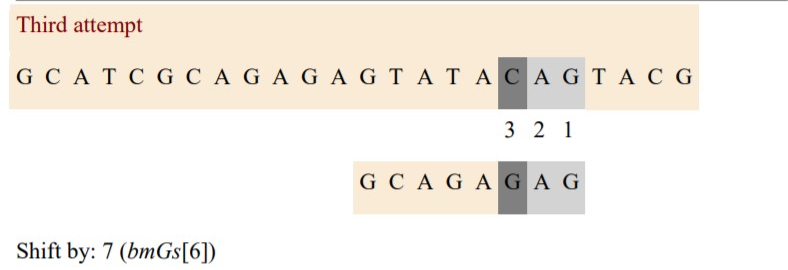
* Pha tiền xử lí:



Bảng ztBc và bmGs được sử dụng bởi thuận toán Zhu-Takaoka

Pha tìm kiếm thực hiện như sau:





Thuật toán Zhu Taraoka thực hiện 12 lần so sánh xâu kí tự trong ví dụ này.

## 2.4 Lập trình theo thuật toán

#include <iostream>

#include <algorithm>

#include <iomanip>

#include <cstdio>

#include <cstring>

#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()

{

printf("Nhap x: ");

gets(x);

m = strlen(x);

XSIZE = m;

printf("Nhap y: ");

gets(y);

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)

Group TTV Page 63 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];

Group TTV Page 64

/\* 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) {

printf("position is %d\n", j);

j += bmGs[0];

}

else

j += max(bmGs[i],

ztBc[y[j + m - 2]][y[j + m - 1]]);

}

}

main()

{

nhap();

ZT(x, m, y, n);

}

Thử nghiệm:

Input:

* Nhập x: GCAGAGAG
* Nhập y: GCATCGCAGAGAGTATACAGTACG

Output:

* Vị trí là 5

## Thuật toán Berry-Ravindran

## 3.1 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)

## 3.2 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 trí ngoài cũng bên phải gần nhất bắt đầu xuất hiện ab trong x.

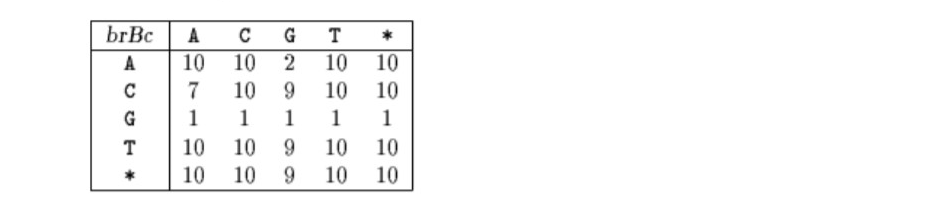
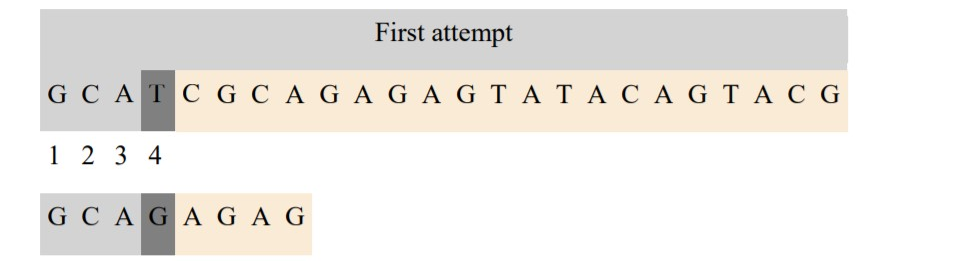


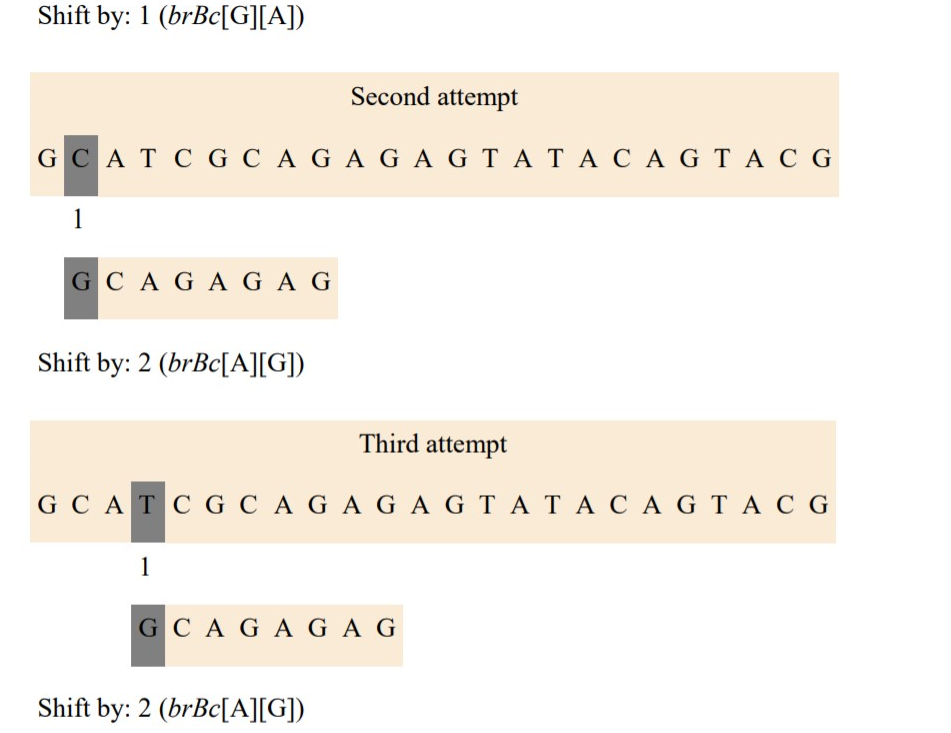
Sau mỗi lần thử mẫu khi cửa sổ đang ở vị trí tương ứng y[j .. j+m-1] bước dịch tiếp theo sẽ là brBc[y[j+m],y[j+m+1]]

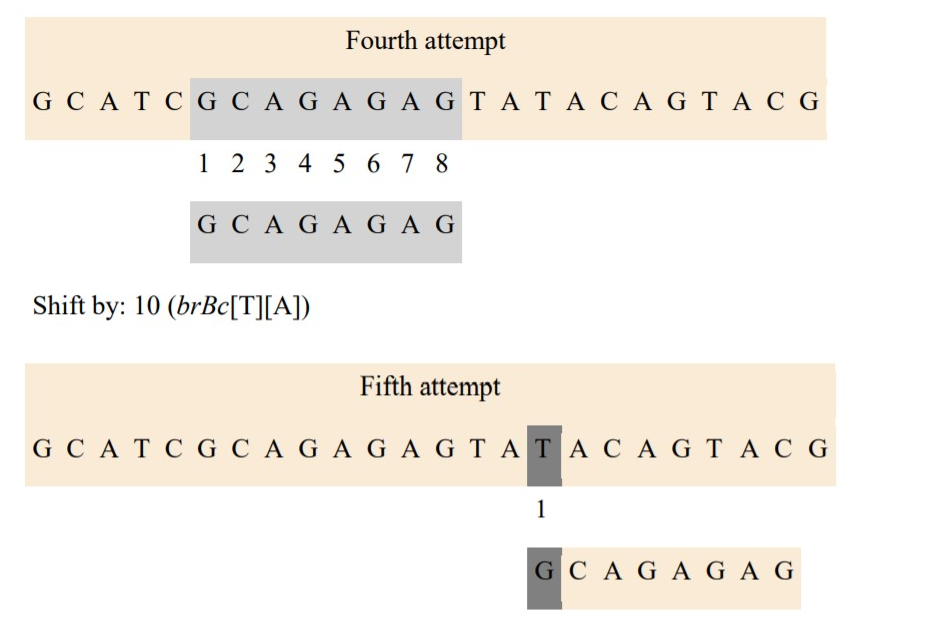
## 3.3 Kiểm nghiệm thuật toán

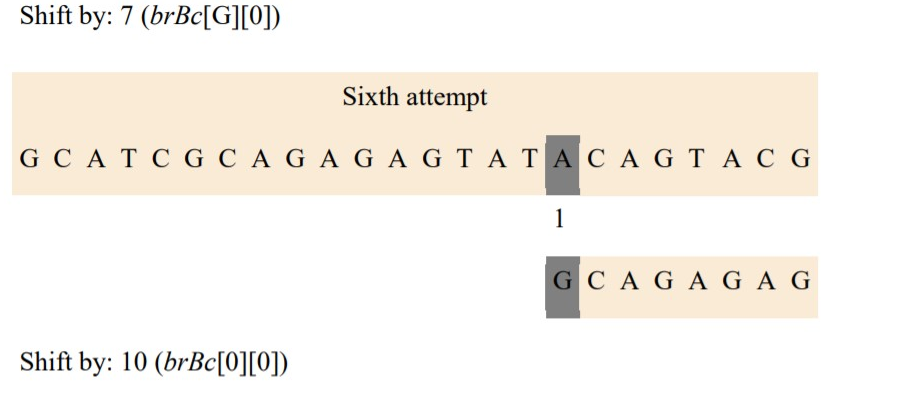
X = “GCAGAGAG”

Y =”GCATCGCAGAGAGTATACAGTACG”

* Pha tiền xử lý xác định:
* Pha tìm kiếm thực hiện như sau:







Trong ví dụ Berry-Ravindran algorithm thực hiện 16 lần so sánh kí tự.

## 3.4 Lập trình theo thuật toán

#include <iostream>

#include <algorithm>

#include <iomanip>

#include <cstdio>

#include <cstring>

#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()

{

printf("Nhap x: ");

gets(x);

m = strlen(x);

printf("Nhap y: ");

gets(y);

n = strlen(y);

}

void preBrBc(char\* x, int m, int brBc[256][256])

{

int a, b, i;

for (a = 0; a < ASIZE; ++a)

for (b = 0; b < ASIZE; ++b)

brBc[a][b] = m + 2;

Group TTV Page 68 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[256][256];

/\* Preprocessing \*/

preBrBc(x, m, brBc);

/\* Searching \*/

y[n + 1] = '\0';

j = 0;

while (j <= n - m) {

if (memcmp(x, y + j, m) == 0)

printf("position is %d\n", j);

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

}

}

main()

{

nhap();

BR(x, m, y, n);

}

**Thử nghiệm:**

Input:

* Nhập x: GCAGAGAG
* Nhập y: GCATCGCAGAGAGTATACAGTACG

Output:

* Vị trí là 5

## Thuật toán Turbo-BM

## 4.1 Đặc điểm

* Đây là thuật toán phát triến từ thuật toán boyer – moore
* Pha chuẩn bị có độ phức tạp O(m + ∂ )
* Pha tìm kiếm có độ phức tạp O(n)
* Trong trường hợp xấu nhất thuật toán cần so sánh 2n kí tự.

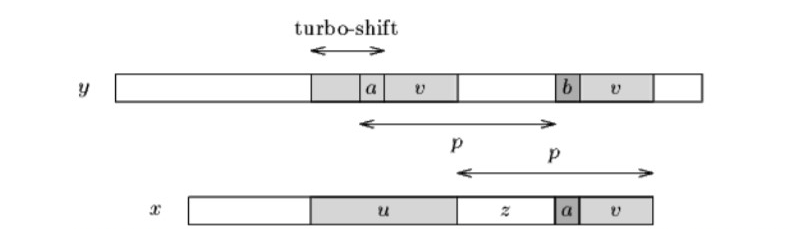
## 4.2 Trình bày thuật toán

Turbo-BM là thuật toán cải tiến từ thuật toán Boyer-moore. Nó yêu cầu mở rộng thêm quá trính tiền xử lý so với thuật toán boyer-moore cơ bản. Bao gồm việc nhớ một đoạn hậu tố của mẫu trùng với đoạn con của xâu y. Trong lần thử mẫu trước (chỉ trường hợp good-suffix ta mới xác định được đoạn u này. Trường hợp khác u = 0)

Kĩ thuật này có hai lợi thế sau:

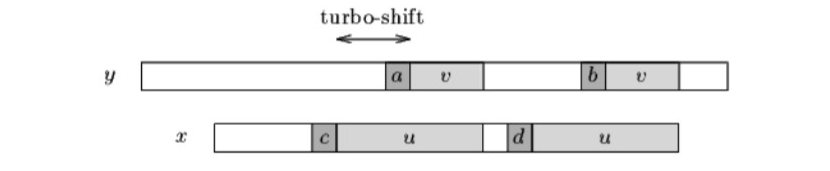
* Nó có thể nhảy qua đoạn xâu con đã biết trước.
* Nó có thể có một bước dịch khác gọi là turbo-shift.

Bước dịch turbo-shift tồn tại trong trường hợp tại vị trí thử hiện tại, hậu tố xâu mẫu đã matches nhỏ hơn so với đoạn nhớ trước đó. Chúng ta có thể gọi u là đoạn nhớ trước đó và v là đoạn hậu tố của x đã match. Như vậy turbo-shift = |u| - |v|



Trường hợp |v| < |u| , độ dài bad-character shift lớn hơn độ dài good- suffix shift và turbo-shift thì bước dịch thực sự sẽ lớn hơn |u|+1.

Như hình vẽ dưới.



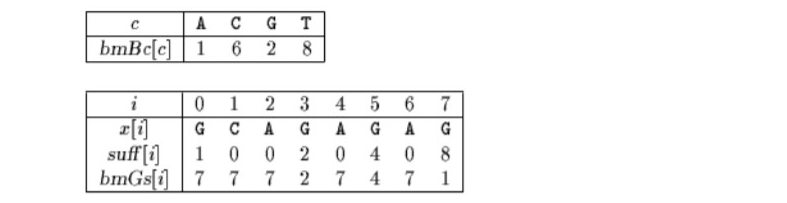
Lý do là bởi 2 kí từ c và d là khác nhau. Với bước dịch < |u| + 1. Nếu c khớp với kí tự trong v. Thì d sẽ không khớp. Ta luôn thấy có 1 tập kí tự khác nhau trong mẫu x và xâu y

## 4.3 Kiểm nghiệm thuật toán

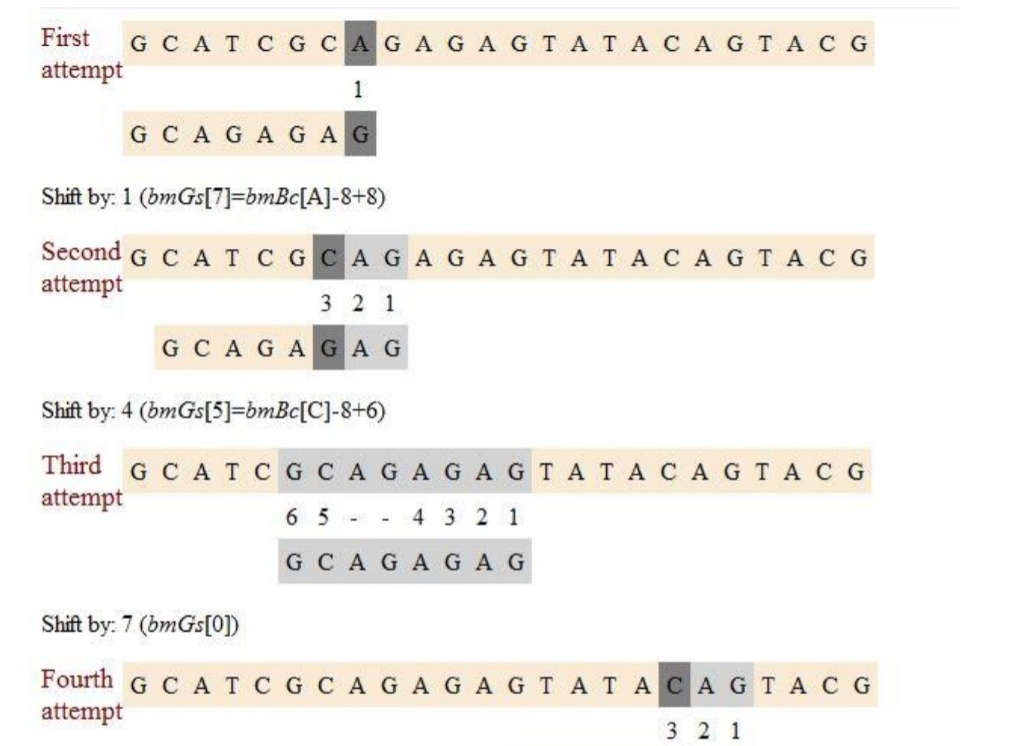
X = “GCAGAGAG”

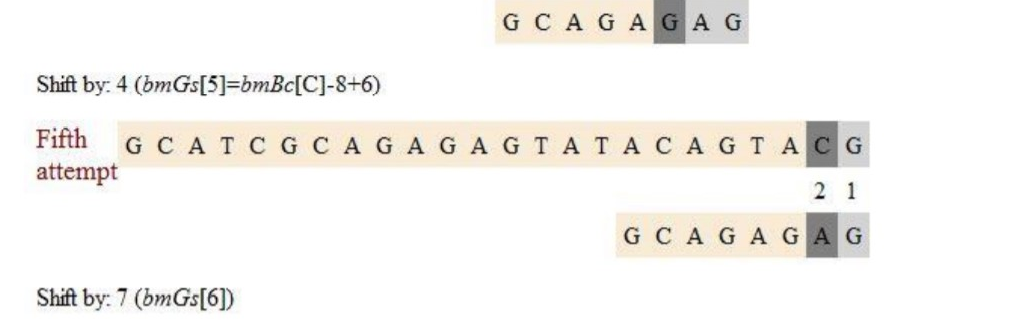
Y =”GCATCGCAGAGAGTATACAGTACG”

* Pha chuẩn bị xác định:



* Pha tìm kiếm thực hiện như sau:





Thuật toán Turbo Boyer-Moore thực hiện 15 lần so sánh kí tự.

## 4.4 Lập trình theo thuật toán

#include <iostream>

#include <algorithm>

#include <iomanip>

#include <cstdio>

Group TTV Page 72

#include <cstring>

#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()

{

printf("Nhap x: ");

gets(x);

m = strlen(x);

XSIZE = m;

printf("Nhap y: ");

gets(y);

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 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 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) {

printf("position is %d\n", 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;

}

}

main()

{

nhap();

TBM(x, m, y, n);

}

# Các thuật toán tìm kiếm mẫu từ vị trí cụ thể

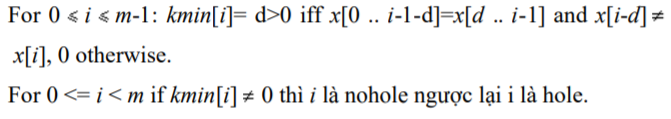
## Thuật toán Colussi

## 1.1 Đặc điểm

* Đây là thuật toán cái tiến từ thuật toán Knuth, Morris and Pratt.
* Mẫu tìm kiếm sẽ được chia thành 2 tập con. Tập thứ nhất sẽ được duyệt từ trái sang phải khi không gặp mismatch ta duyệt tập thứ hai từ trái qua phải.
* Pha xử lý có độ phức tạp là O(m)
* Pha tìm kiếm có độ phức tạp là O(n)
* Trong trường hợp xấu nhất quá trình thực thi sẽ mất 3/2 n lần so sánh kí tự.

## 1.2 Trình bày thuật toán

* Thuật toán Colussi được thiết kế dựa theo những phân tích từ thuật toán knuth, Morris, và Pratt. các vị trí kí tự trong mẫu được chia thành 2 tập riêng biệt. mỗi phép thử sẽ bao gồm 2 pha:
* Trong pha thứ nhất thực hiên so sánh từ trái sang phải lần lượt các kí tự tại những vị trí kmpNext > -1. Những vị trí này gọi là noholes.
* Pha thứ hai so sánh kí tự từ phải sang trái (các kí tự còn lại ) nhưng kí tự này có kmpNext = -1 và gọi là holes.
* Chiến lược này có 2 lợi thế:
* Khi gặp mismath trong pha thứ nhất, sau khi thực hiện shift bước dịch ta không cần phải so sánh lại các kí tự tại những vị trí noholes phía trước.
* Khi gặp mismath trong pha thứ hai có nghĩa là đoạn hậu tố đã matches tất cả kí tự định nghĩa biến:

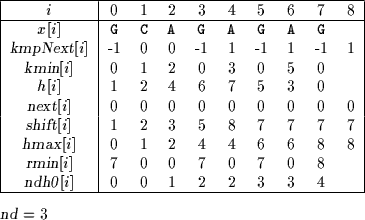


## 1.3 Kiểm nghiệm thuật toán

X = “GCAGAGAG”

Y =”GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lí xác định



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: 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 | | | | | | | | | | | | | | | | | | | | | | | | |
| 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])

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 (*shift*[0])

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 (*shift*[0])

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 | 2 |  | 3 |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | G | C | A | G | A | G | A | G |

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

Thuật toán Colussi thực hiện 20 lần so sánh kí tự trong ví dụ.

## 1.4 Lập trình theo thuật toán

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  char x[100001], y[100001];  int XSIZE, m, n;  void nhap()  {  printf("Nhap x: ");  gets(x);  m = strlen(x);  printf("Nhap y: ");  gets(y);  n = strlen(y);  XSIZE = m;  }  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])  {  printf("position is %d\n", j);  i = m;  }  if (i > nd)  last = j + m - 1;  j += shift[i];  i = next[i];  }  }  main()  {  nhap();  COLUSSI(x, m, y, n);  } |

## **Thuật toán Skip Search**

## 2.1 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)

## 2.2 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)

## 2.3 Kiểm thử thuật toán

X = “GCAGAGAG”

Y = ”GCATCGCAGAGAGTATACAGTACG”

Pha tiền xử lý xác định:

Pha tìm kiếm:



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 |

Thuật toán Skip Search thực hiện 14 lần so sánh kí tự trong ví dụ.

## 2.4 Lập trình theo thuật toán

|  |
| --- |
| #include <bits/stdc++.h>  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);  } |

**Thử nghiệm**

Input:

* Nhập x: GCAGAGAG
* Nhập y: GCATCGCAGAGAGTATACAGTACG

Output :

* Vị trí là 5

# Các thuật toán tìm kiếm mẫu từ vị trí bất kỳ

## Thuật toán Horspool

## 1.1 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)

## 1.2 Trình bày thuật toán

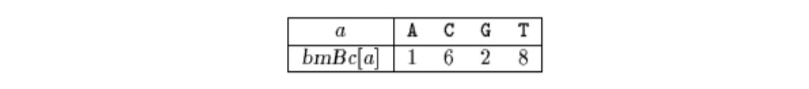
Quy tắc dịch bad- character được sử dụng trong Boyer Moore không có hiệu quả trong đoạn văn bản kí tự nhỏ, nhưng khi đoạn văn bản là lớn và phải so sánh với mẫu dài, nó thường trong trường hợp bảng bã ASCII và tìm kiếm thông thường trong soạn thảo văn bản, chúng rất hữu dụng. Sử dụng nó như những kết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

## 1.3 Kiểm thử thuật toán

X = “GCAGAGAG”

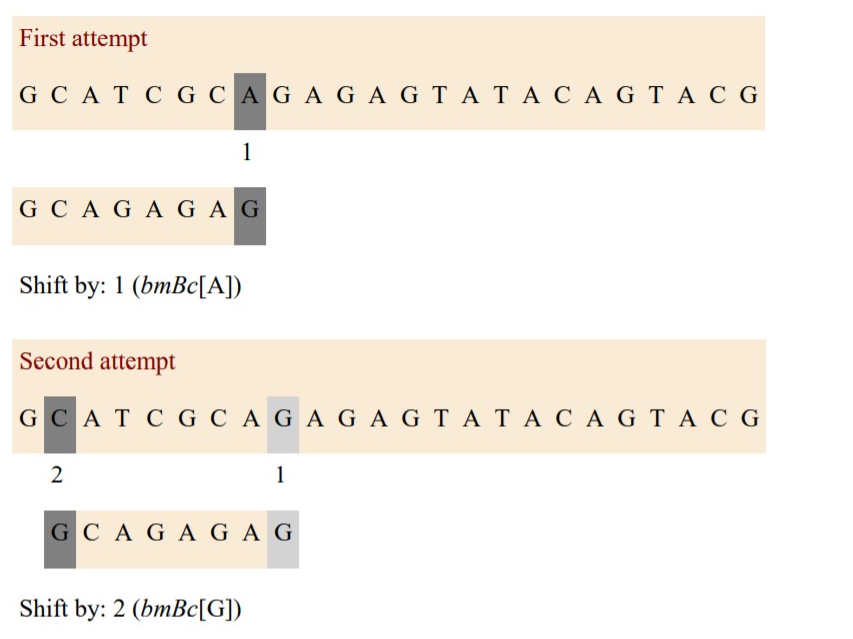
Y =”GCATCGCAGAGAGTATACAGTACG”

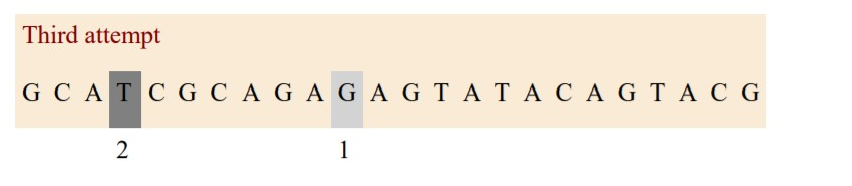
* Pha tiền xử lí:

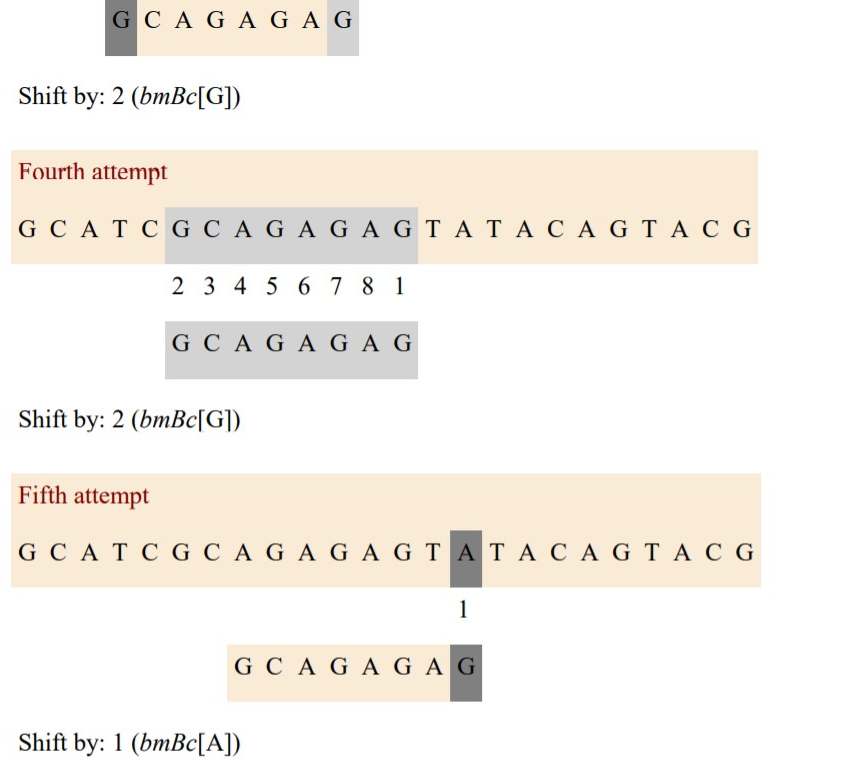


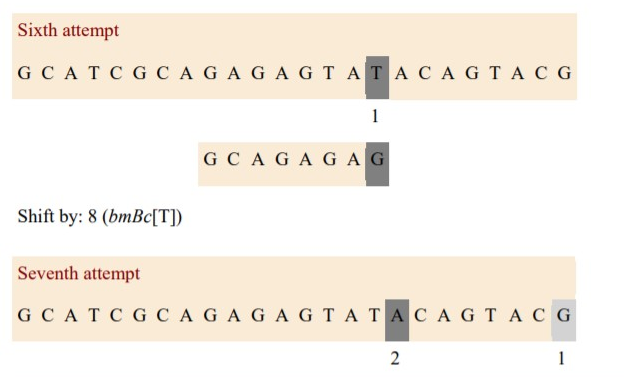
bảng bmBc của thuật toán Horspool

* Pha tìm kiếm thực hiện:











Cần 17 lượt so sánh trong ví dụ này.

## 1.4 Lập trình theo thuật toán

#include <iostream>

#include <algorithm>

#include <iomanip>

#include <cstdio>

#include <cstring>

#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()

{

printf("Nhap x: ");

gets(x);

m = strlen(x);

printf("Nhap y: ");

gets(y);

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 HORSPOOL(char\* x, int m, char\* y, int n)

{

int j, bmBc[ASIZE];

char c;

/\* Preprocessing \*/

preBmBc(x, m, bmBc);

/\* Searching \*/

j = 0;

while (j <= n - m) {

c = y[j + m - 1];

if (x[m - 1] == c && memcmp(x, y + j, m - 1) == 0)

printf("position is %d\n", j);

j += bmBc[c];

}

}

main()

{

nhap();

HORSPOOL(x, m, y, n);

}

**Thử nghiệm:**

Input:

* Nhập x: GCAGAGAG
* Nhập y: GCATCGCAGAGAGTATACAGTACG

Output:

* Vị trì là 5

## Thuật toán Quick Search

## 2.1 Đặc điểm

* Là thuật toán đơn giản của Boyer Moore
* Chỉ 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)
* Rất nhanh trong thực tế với những mẫu ngắn và anphabets lớn.

## 2.2 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 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

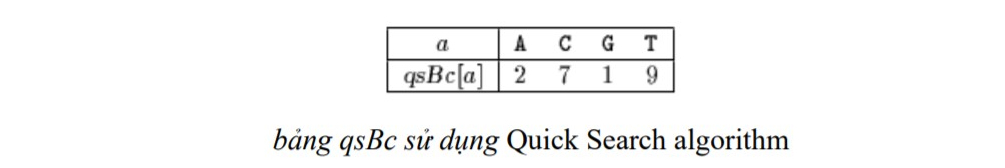
với trường hợp còn lại.

## 2.3 Kiểm thử thuật toán

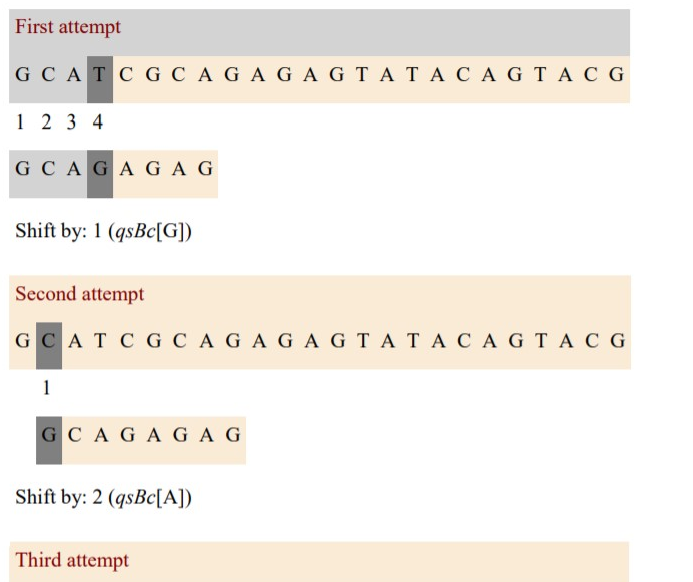
X = “GCAGAGAG”

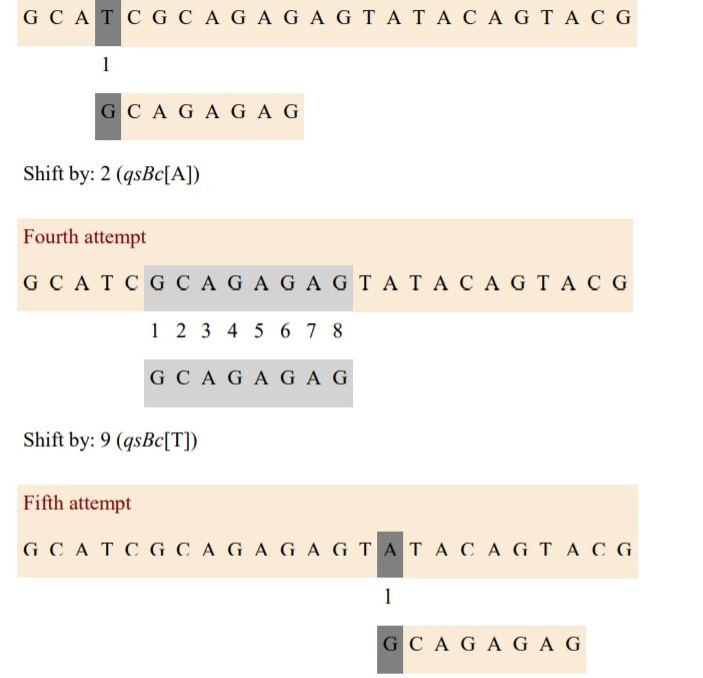
Y =”GCATCGCAGAGAGTATACAGTACG”

* Pha tiền xử lí:



* Pha tìm kiếm thực hiện:





Shift by: 7 (qsBc[C])

Quick Search algorithm thực thi 15 lần so sánh kí tự trong ví dụ này.

## 2.4 Lập trình theo thuật toán

#include <iostream>

#include <algorithm>

#include <iomanip>

#include <cstdio>

#include <cstring>

#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()

{

printf("Nhap x: ");

gets(x);

m = strlen(x);

printf("Nhap y: ");

gets(y);

n = strlen(y);

}

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)

printf("position is %d\n", j);

j += qsBc[y[j + m]]; /\* shift \*/

}

}

main()

{

nhap();

QS(x, m, y, n);

}

**Thử nghiệm:**

Input:

* Nhập x: GCAGAGAG
* Nhập y: GCATCGCAGAGAGTATACAGTACG

Output:

* Vị trí là 5

## Smith algorithm

## 3.1 Đặc điểm

* Thuật toán Smith tính max của hàm dịch bad charater Horspool và hàm dịch bad charater Quick Search.
* 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)

## 3.2 Trình bày thuật toán

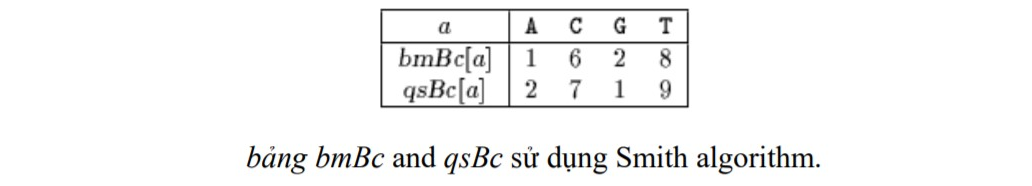
* Smith nhận thấy rằng có thể tính toán được rằng số bước dịch với kí tự tiếp theo kí tự đầu mút bên phải của cửa sổ dịch chuyển đôi khi cho số bước dịch ngắn hơn việc sử dụng đúng kí tự đầu mút bên phải cửa sổ ấy. Ông ấy khuyên nên chọn max giữa 2 giá trị.
* Pha tiền xử lí của thuật toán Smith bao gồm tính toán của hàm dịch chuyển các bad- character: preBmBc(X,m) , và hàm dịch chuyển các badcharacter trong Quick Search: preQsBc(X,m)
* Pha tìm kiếm có độ phức tạp về thời gian trong trường hợp xấu nhất là một hàm bậc 2.

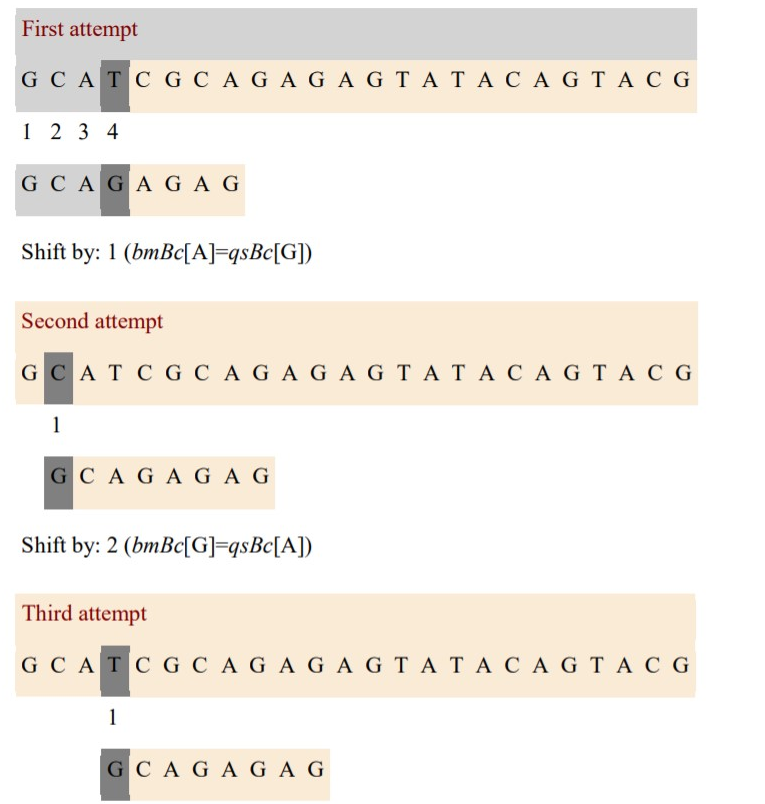
## 3.3 Kiểm thử thuật toán

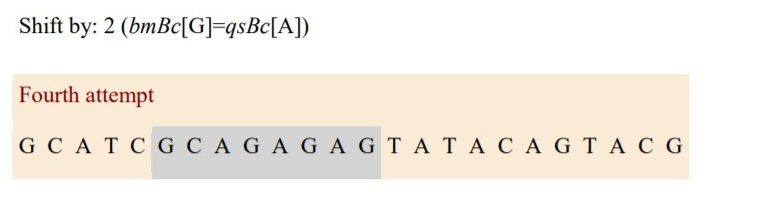
X = “GCAGAGAG”

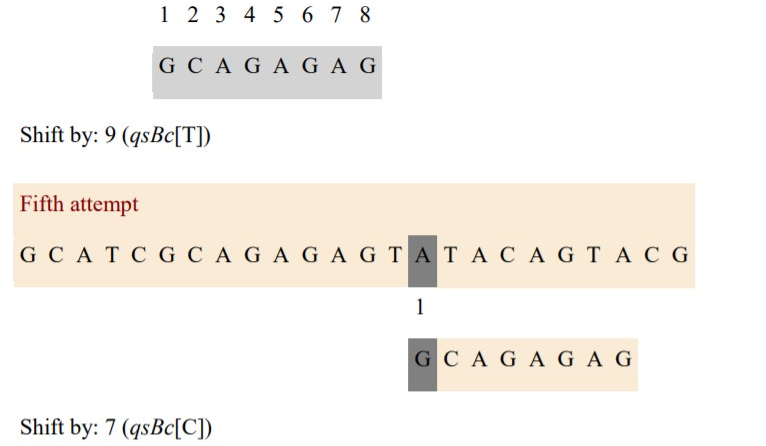
Y =”GCATCGCAGAGAGTATACAGTACG”

* Pha tiền xử lí tính toán được:









## 3.4 Lập trình theo thuật toán

#include <iostream>

#include <algorithm>

#include <iomanip>

#include <cstdio>

#include <cstring>

#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()

{

printf("Nhap x: ");

gets(x);

m = strlen(x);

printf("Nhap y: ");

gets(y);

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 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 SMITH(char\* x, int m, char\* y, int n)

{

int j, bmBc[ASIZE], qsBc[ASIZE];

/\* Preprocessing \*/

preBmBc(x, m, bmBc);

preQsBc(x, m, qsBc);

j = 0;

while (j <= n - m) {

if (memcmp(x, y + j, m) == 0)

printf("position is %d\n", j);

j += max(bmBc[y[j + m - 1]], qsBc[y[j + m]]);

}

}

main()

{

nhap();

SMITH(x, m, y, n);

}

**Thử nghiệm:**

Input:

* Nhập x: GCAGAGAG
* Nhập y: GCATCGCAGAGAGTATACAGTACG

Output:

* Vị trí là 5