# Mục lục

[**I. Tổng quan 1**](#_goz5k01wj1f8)

[**II. Tìm kiếm mẫu từ trái sang phải 1**](#_e06telu4fear)

[1. Thuật toán Karp-Rabin 1](#_nr6yad7g0v9c)

[2. Thuật toán Morris-Pratt 4](#_ip4ccygfmhmp)

[3. Thuật toán Knuth-Morris-Pratt 7](#_r4m5t0y1qdp8)

[4. Thuật toán Not so Naive 10](#_iu1tj33fsbad)

[5. Thuật toán Apostolico-Crochemore 13](#_qddqtx17feg8)

[**III. Tìm kiếm mẫu từ phải sang trái 17**](#_pnkjpge3aku1)

[1. Thuật toán Boyer-Moore 17](#_b2dq2iicf6ox)

[2. Thuật toán Turbo-Boyer-Moore 24](#_4ypzxl925dgs)

[3. Thuật toán Zhu-Takaoka 30](#_4nz9kar1x89d)

[4. Thuật toán Berry-Ravindran 35](#_z0unm93b9zt8)

[**IV. Tìm kiếm mẫu từ vị trí xác định 39**](#_rt9vps2bztl1)

[1. Thuật toán Skip-Search 39](#_hnbovo5xbpah)

[**V. Tìm kiếm mẫu từ vị trí ngẫu nhiên 43**](#_jj03xzb7dof8)

[1. Thuật toán Horspool 43](#_e1bw1q5ner4q)

[2. Thuật toán Quick search 45](#_zc8zbdud0hhd)

[3. Thuật toán Tuned Boyer-Moore 47](#_5yv4x78nqul)

[4. Thuật toán Raita 51](#_dvcoogls5rdj)

[5. Thuật toán Smith 54](#_74llk7psve7n)

[6. Thuật toán Brute Force 56](#_fmckgl745c6)

# 

# Tổng quan

So khớp chuỗi là một chủ đề rất quan trọng trong phạm vi rộng hơn của

xử lý văn bản. Các thuật toán so khớp chuỗi là các thành phần cơ bản được sử dụng trong việc triển khai các phần mềm thực tế hiện có trong hầu hết các hệ điều hành. Hơn nữa, họ nhấn mạnh các phương pháp lập trình đóng vai trò là mô hình trong các lĩnh vực khoa học máy tính khác (thiết kế hệ thống hoặc phần mềm). Cuối cùng, chúng cũng đóng một vai trò quan trọng trong khoa học máy tính lý thuyết bằng cách đưa ra những vấn đề đầy thách thức.

Mặc dù dữ liệu được ghi nhớ theo nhiều cách khác nhau nhưng văn bản vẫn là hình thức trao đổi thông tin chính. Điều này đặc biệt rõ ràng trong văn học hoặc ngôn ngữ học, nơi dữ liệu bao gồm các kho văn bản và từ điển khổng lồ. Điều này cũng áp dụng cho khoa học máy tính nơi một lượng lớn dữ liệu được lưu trữ trong các tệp tuyến tính. Và đây cũng là trường hợp, ví dụ, trong sinh học phân tử vì các phân tử sinh học thường có thể được coi gần đúng là các chuỗi nucleotide hoặc axit amin. Hơn nữa, số lượng dữ liệu có sẵn trong các lĩnh vực này có xu hướng tăng gấp đôi sau mỗi 18 tháng. Đây là lý do tại sao các thuật toán phải hiệu quả ngay cả khi tốc độ và dung lượng lưu trữ của máy tính tăng lên thường xuyên.

So khớp chuỗi bao gồm việc tìm một hoặc tổng quát hơn là tất cả các lần xuất hiện của một chuỗi (thường được gọi là mẫu) trong văn bản. Tất cả các thuật toán trong cuốn sách này đều xuất ra tất cả các lần xuất hiện của mẫu trong văn bản. Mẫu được ký hiệu là x = x[0… m - 1]; chiều dài của nó bằng m. Văn bản được ký hiệu là y = y[0… n - 1]; chiều dài của nó bằng n. Cả hai chuỗi đều được xây dựng trên một tập hợp ký tự hữu hạn được gọi là bảng chữ cái được biểu thị bằng kí tự Σ kích thước bằng σ .

# Tìm kiếm mẫu từ trái sang phải

## Thuật toán Karp-Rabin

**Đặc điểm**

* Dùng một hàm băm
* Pha tiền xử lý có độ phức tạp thời gian O(m) và độ phức tạp không gian là hằng số
* Độ phức tạp thời gian pha tìm kiếm là O(m x n)
* Độ phức tạp thời gian mong đợi khi chạy là O(m + n)

**Mô tả**

Băm cung cấp một phương pháp đơn giản để tránh số lượng so sánh ký tự bậc hai trong hầu hết các tình huống thực tế. Thay vì kiểm tra ở mỗi vị trí của văn bản xem mẫu có xuất hiện không, dường như việc chỉ kiểm tra xem nội dung của cửa sổ có "giống" mẫu không là hiệu quả hơn.

Để kiểm tra sự giống nhau giữa hai từ này, một hàm băm được sử dụng. Để hữu ích cho vấn đề khớp chuỗi, một hàm băm (hash) nên có các thuộc tính sau:

* Dễ tính toán
* Có khả năng phân biệt cao cho chuỗi
* hash(y[j + 1 … j + m]) có thể tính được dễ dàng từ hash(y[j … j + m - 1]) và y[j + m]
* hash(y[j + 1 … j + m]) = rehash(y[j], y[j + m], hash(y[j … j + m - 1]))

Hash(w) của từ một từ w có độ dài m được định nghĩa như sau:



(q là một số lớn), từ đó ta có:



Các bước tiền xử lý của thuật toán Karp-Rabin bao gồm việc tính toán giá trị băm của x. Điều này có thể được thực hiện trong không gian hằng số và thời gian O(m).

Trong pha tìm kiếm, so sánh giá trị băm của x với giá trị băm của y[j … j + m - 1] với 0 <= j <= n – m là đủ. Nếu tìm thấy sự bằng nhau, ta vẫn nên kiểm tra sự bằng nhau x = y[j ... j + m - 1] theo từng ký tự

Độ phức tạp thời gian của giai đoạn tìm kiếm của thuật toán Karp-Rabin là O(m x n). Số lần so sánh ký tự trong văn bản được kỳ vọng là O(m + n).

**Kiểm nghiệm**

Cho 2 xâu:

* X = GCAGAGAG, m = 8
* Y = GCATCGCAGAGAGTATACAGTACG, n = 24

Tìm kiếm:

|  |  |  |
| --- | --- | --- |
| **Vị trí cửa sổ** | **y** | **x** |
| 1 | Hash(y[0 .. 7]) = 17819 | Hash(x) = 17597 |
| 2 | Hash(y[1 .. 8]) = 17533 | Hash(x) = 17597 |
| 3 | Hash(y[2 .. 9]) = 17979 | Hash(x) = 17597 |
| 4 | Hash(y[3 .. 10]) = 19389 | Hash(x) = 17597 |
| 5 | Hash(y[4 .. 11]) = 17339 | Hash(x) = 17597 |
| 6 | Hash(y[5 .. 12]) = 17597 | Hash(x) = 17597 |
| 7 | Hash(y[6 .. 13]) = 17102 | Hash(x) = 17597 |
| 8 | Hash(y[7 .. 14]) = 17117 | Hash(x) = 17597 |
| 9 | Hash(y[8 .. 15]) = 17678 | Hash(x) = 17597 |
| 10 | Hash(y[9 .. 16]) = 17245 | Hash(x) = 17597 |
| 11 | Hash(y[10 .. 17]) = 17917 | Hash(x) = 17597 |
| 12 | Hash(y[11 .. 18]) = 17723 | Hash(x) = 17597 |
| 13 | Hash(y[12 .. 19]) = 18877 | Hash(x) = 17597 |
| 14 | Hash(y[13 .. 20]) = 19662 | Hash(x) = 17597 |
| 15 | Hash(y[14 .. 21]) = 17885 | Hash(x) = 17597 |
| 16 | Hash(y[15 .. 22]) = 19197 | Hash(x) = 17597 |
| 17 | Hash(y[16 .. 23]) = 16961 | Hash(x) = 17597 |

**Cài đặt**

import sys

q = sys.maxsize

def rehash(*a*,*b*,*h*):

return (((*h* - *a*\*d) << 1) + *b*) % q

def KR(*x*,*m*,*y*,*n*):

hx, hy, j = 0, 0, 0

for i in range(*m*):

hx = ((hx << 1) + ord(*x*[i])) % q

hy = ((hy << 1) + ord(*y*[i])) % q

while j <= *n* - *m*:

# print(f"Lần thử {j + 1}")

# print(hx)

# print(hy)

if hx == hy and *x* == *y*[j:*m*+j]:

print(f"Vị trí {j} xuất hiện vị trí của xâu x trong y")

if j < *n* - *m*:

hy = rehash(ord(*y*[j]), ord(*y*[j+*m*]),hy)

j += 1

x = 'GCAGAGAG'

y = 'GCATCGCAGAGAGTATACAGTACG'

m = len(x)

n = len(y)

d = 1

for i in range(1,m):

d = d << 1

KR(x,m,y,n)

## Thuật toán Morris-Pratt

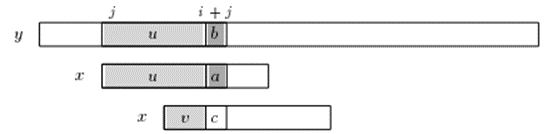
**Đặc điểm**

* Thực hiện các so sánh từ trái sang phải
* Giai đoạn tiền xử lý có độ phức tạp không gian O(m) và độ phức tạp thời gian là O(m)
* Giai đoạn tìm kiếm có độ phức tạp thời gian là O(m+n) (độc lập với kích thước của bảng chữ cái)
* Thực hiện tối đa 2n - 1 lần so sánh ký tự trong văn bản trong giai đoạn tìm kiếm
* Độ trễ được giới hạn bởi m (số lần so sánh tối đa cho một ký tự trong văn bản)

**Mô tả**

Thiết kế của thuật toán Morris-Pratt tuân theo một phân tích chặt chẽ của thuật toán Brute Force, và đặc biệt là cách mà thuật toán Brute Force lãng phí thông tin đã thu thập trong quá trình quét văn bản.

Xem xét ở một vị trí j trên y, tức là khi cửa sổ được đặt trên phần văn bản y[j … j+m-1]. Giả sử sự không khớp đầu tiên xảy ra giữa x[i] và y[i+j] với 0 < i < m. Khi đó, x[0 … i-1] = y[j … i+j-1] = u và a = x[i] ≠ y[i+j]=b.



Khi dịch chuyển, hợp lý khi mong đợi rằng một tiền tố v của mẫu khớp với một hậu tố của phần u của văn bản. Tiền tố dài nhất như vậy được gọi là biên của u (nó xảy ra ở cả hai đầu của u). Điều này giới thiệu ký hiệu: hãy cho mpNext[i] là độ dài của biên dài nhất của x[0 ... i-1] cho 0 < i ≤ m. Sau đó, sau mỗi dịch chuyển, các so sánh có thể tiếp tục giữa các ký tự c=x[mpNext[i]] và y[i+j]=b mà không bỏ lỡ bất kỳ sự xuất hiện nào của x trong y, và tránh việc quay lại trên văn bản. Giá trị của mpNext[0] được thiết lập là -1.

Bảng mpNext có thể được tính toán với độ phức tạp không gian và thời gian O(m) nếu như áp dụng cùng thuật toán tìm kiếm cho mẫu chính nó, như nếu x=y.

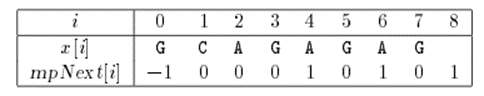
Sau đó, giai đoạn tìm kiếm có thể được thực hiện trong thời gian O(m+n).

**Kiểm nghiệm**

Cho 2 xâu:

* X = GCAGAGAG, m = 8
* Y = GCATCGCAGAGAGTATACAGTACG, n = 24

Bảng mpNext:



Tìm kiếm:

|  |  |  |
| --- | --- | --- |
| **Index bắt đầu** | **So sánh** | **Dịch (i – mpNext[i])** |
| 0 | y = **GCA**TCGCA  x = **GCA**GAGAG | 3 |
| 3 | y = TCGCAGAG  x = GCAGAGAG | 1 |
| 4 | y = TCGCAGAG  x = GCAGAGAG | 1 |
| 5 | y = **GCAGAGAG**  x = **GCAGAGAG** | 7 |
| 12 | y = **G**TATACAG  x = **G**CAGAGAG | 1 |
| 13 | y = TATACAGT  x = GCAGAGAG | 1 |
| 14 | y = ATACAGTA  x = GCAGAGAG | 1 |
| 15 | y = TACAGTAC  x = GCAGAGAG | 1 |
| 16 | y = ACAGTACG  x = GCAGAGAG | 1 |

**Cài đặt**

def preMP(*x*,*m*,*mpNext*):

i = 0

j = *mpNext*[0]

while i < *m*:

while j > -1 and *x*[i] != *x*[j]:

j = *mpNext*[j]

i += 1

j += 1

*mpNext*[i] = j

def MP(*x*,*m*,*y*,*n*):

mpNext = [-1 for i in range(*m* + 1)]

preMP(*x*,*m*,mpNext)

i = j = 0

while j < *n*:

if i > -1 and *x*[i] != *y*[j] and j >= *n* - *m*:

# print(f"Dịch 1 khoảng = {i - mpNext[i]} ({i} - {mpNext[i]})")

break

while i > -1 and *x*[i] != *y*[j]:

# print(f"Dịch 1 khoảng = {i - mpNext[i]} ({i} - {mpNext[i]})")

i = mpNext[i]

i += 1

j += 1

if i >= *m*:

print(f"Vị trí {j - i} xuất hiện vị trí của xâu x trong y")

# print(f"Dịch 1 khoảng = {i - mpNext[i]} ({i} - {mpNext[i]})")

i = mpNext[i]

x = 'GCAGAGAG'

y = 'GCATCGCAGAGAGTATACAGTACG'

m = len(x)

n = len(y)

MP(x,m,y,n)

## Thuật toán Knuth-Morris-Pratt

Đặc điểm

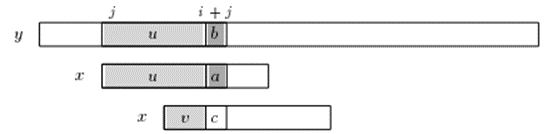
* Thực hiện các so sánh từ trái sang phải
* Giai đoạn tiền xử lý có độ phức tạp không gian O(m) và độ phức tạp thời gian là O(m)
* Giai đoạn tìm kiếm có độ phức tạp thời gian là O(m+n) (độc lập với kích thước của bảng chữ cái)
* Thực hiện tối đa 2n - 1 lần so sánh ký tự trong văn bản trong giai đoạn tìm kiếm
* Độ trễ được giới hạn bởi LogΦ(m), với tỉ lệ vàng Φ = (1 + √5)/2 (số lần so sánh tối đa cho một ký tự trong văn bản)

**Mô tả**

Thiết kế của thuật toán Knuth-Morris-Pratt tuân theo một phân tích chặt chẽ của thuật toán Morris Pratt.

Xem xét ở một vị trí j trên y, tức là khi cửa sổ được đặt trên phần văn bản y[j … j+m-1]. Giả sử sự không khớp đầu tiên xảy ra giữa x[i] và y[i+j] với 0 < i < m. Khi đó, x[0 … i-1] = y[j … i+j-1] = u và a = x[i] ≠ y[i+j]=b.

Khi dịch chuyển, hợp lý khi mong đợi rằng một tiền tố v của mẫu khớp với một hậu tố của phần u của văn bản. Hơn nữa, nếu chúng ta muốn tránh sự không khớp ngay lập tức khác, ký tự theo sau tiền tố v trong mẫu phải khác với a. Tiền tố dài nhất như vậy được gọi là biên gắn thẻ của u – tagged border (nó xảy ra ở cả hai đầu của u tiếp theo bởi các ký tự khác nhau trong x).



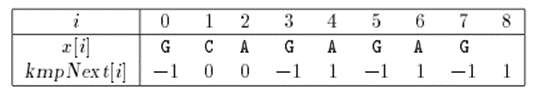
Điều này giới thiệu ký hiệu: hãy cho kmpNext[i] là độ dài của biên dài nhất của x[0 .. i-1] tiếp theo là một ký tự c khác với x[i] và -1 nếu không có biên gắn thẻ như vậy tồn tại, cho 0 < i ≤ m. Sau đó, sau mỗi dịch chuyển, các so sánh có thể tiếp tục giữa các ký tự x[kmpNext[i]] và y[i+j] mà không bỏ lỡ bất kỳ sự xuất hiện nào của x trong y, và tránh quay lại trên văn bản. Giá trị của kmpNext[0] được thiết lập là -1.

**Kiểm nghiệm**

Cho 2 xâu:

* X = GCAGAGAG, m = 8
* Y = GCATCGCAGAGAGTATACAGTACG, n = 24

Bảng KmpNext:



Tìm kiếm:

|  |  |  |
| --- | --- | --- |
| **Index bắt đầu** | **So sánh** | **Dịch (i – mpNext[i])** |
| 0 | y = **GCA**TCGCA  x = **GCA**GAGAG | 4 |
| 4 | y = **CG**CAGAGA  x = **GC**AGAGAG | 1 |
| 5 | y = **GCAGAGAG**  x = **GCAGAGAG** | 7 |
| 12 | y = **G**TATACAG  x = **G**CAGAGAG | 1 |
| 13 | y = TATACAGT  x = GCAGAGAG | 1 |
| 14 | y = ATACAGTA  x = GCAGAGAG | 1 |
| 15 | y = TACAGTAC  x = GCAGAGAG | 1 |
| 16 | y = ACAGTACG  x = GCAGAGAG | 1 |

**Cài đặt**

def preKMP(*x*,*m*,*KmpNext*):

i = 0

j = *KmpNext*[0]

while i < *m*:

while j > -1 and *x*[i] != *x*[j]:

j = *KmpNext*[j]

i += 1

j += 1

if i < *m* and *x*[i] == *x*[j]:

*KmpNext*[i] = *KmpNext*[j]

else:

*KmpNext*[i] = j

def KMP(*x*,*m*,*y*,*n*):

KmpNext = [-1 for i in range(*m* + 1)]

preKMP(*x*,*m*,KmpNext)

print(KmpNext)

i = j = 0

while j < *n*:

if i > -1 and *x*[i] != *y*[j] and j >= *n* - *m*:

# print(f"Dịch 1 khoảng = {i - KmpNext[i]} ({i} - {KmpNext[i]})")

break

while i > -1 and *x*[i] != *y*[j]:

# print(f"Dịch 1 khoảng = {i - KmpNext[i]} ({i} - {KmpNext[i]})")

i = KmpNext[i]

i += 1

j += 1

if i >= *m*:

print(f"Vị trí {j - i} xuất hiện vị trí của xâu x trong y")

# print(f"Dịch 1 khoảng = {i - KmpNext[i]} ({i} - {KmpNext[i]})")

i = KmpNext[i]

x = 'GCAGAGAG'

y = 'GCATCGCAGAGAGTATACAGTACG'

m = len(x)

n = len(y)

KMP(x,m,y,n)

## Thuật toán Not so Naive

**Đặc điểm**

* Giai đoạn tiền xử lý có độ phức tạp thời gian và không gian hằng số
* Giai đoạn tìm kiếm có độ phức tạp thời gian O(n x m)
* Hơi (theo hệ số) dưới tuyến tính (sub-linear) trong trường hợp trung bình

**Mô tả**

Trong giai đoạn tìm kiếm của thuật toán Not So Naive, các so sánh ký tự được thực hiện với các vị trí của mẫu theo thứ tự sau: 1, 2, ... , m-2, m-1, 0.

Đối với mỗi lần thử khi cửa sổ được đặt trên văn bản y[j ... j+m-1]:

* Nếu x[0] = x[1] và x[1] ≠ y[j+1] hoặc nếu x[0] ≠ x[1] và x[1] = y[j+1] thì mẫu sẽ được dịch đi 2 vị trí
* Ngược lại, sẽ dịch 1 vị trí

Do đó, giai đoạn tiền xử lý có thể được thực hiện trong thời gian và không gian hằng số. Giai đoạn tìm kiếm của thuật toán Not So Naive có trường hợp xấu nhất là bậc hai nhưng nó hơi (theo hệ số) dưới tuyến tính trong trường hợp trung bình.

**Kiểm nghiệm**

Cho 2 xâu:

* X = GCAGAGAG, m = 8
* Y = GCATCGCAGAGAGTATACAGTACG, n = 24

x[0] != x[1]

|  |  |  |
| --- | --- | --- |
| **Index bắt đầu** | **So sánh** | **Dịch (1: x[1] ≠ y[j+1], 2: x[1] = y[j+1])** |
| 0 | y = G**CA**TCGCA  x = G**CA**GAGAG | 2 |
| 2 | y = ATCGCAGA  x = GCAGAGAG | 1 |
| 3 | y = T**C**GCAGAG  x = G**C**AGAGAG | 2 |
| 5 | y = **GCAGAGAG**  x = **GCAGAGAG** | 2 |
| 7 | y = AGAGAGTA  x = GCAGAGAG | 1 |
| 8 | y = GAGAGTAT  x = GCAGAGAG | 1 |
| 9 | y = AGAGTATA  x = GCAGAGAG | 1 |
| 10 | y = GAGTATAC  x = GCAGAGAG | 1 |
| 11 | y = AGTATACA  x = GCAGAGAG | 1 |
| 12 | y = GTATACAG  x = GCAGAGAG | 1 |
| 13 | y = TATACAGT  x = GCAGAGAG | 1 |
| 14 | y = ATACAGTA  x = GCAGAGAG | 1 |
| 15 | y = TACAGTAC  x = GCAGAGAG | 1 |
| 16 | y = A**CAG**TACG  x = G**CAG**AGAG | 2 |

**Cài đặt**

def NSN(*x*,*m*,*y*,*n*):

j,k,ell = 0,0,0

if *x*[0] == *x*[1]:

k = 2

ell = 1

else:

k = 1

ell = 2

while j <= *n* - *m*:

if *x*[1] != *y*[j + 1]:

j += k

else:

if *x*[2 : *m*] == *y*[j + 2 : j + *m*] and *x*[0] == *y*[j]:

print(f"Vị trí {j} xuất hiện vị trí của xâu x trong y")

j += ell

x = 'GCAGAGAG'

y = 'GCATCGCAGAGAGTATACAGTACG'

m = len(x)

n = len(y)

NSN(x,m,y,n)

## Thuật toán Apostolico-Crochemore

**Đặc điểm**

* Giai đoạn tiền xử lý có độ phức tạp không gian O(m) và độ phức tạp thời gian là O(m)
* Giai đoạn tìm kiếm có độ phức tạp thời gian là O(n)
* Thực hiện tối đa 3n/2 lần so sánh ký tự trong văn bản trong trong trường hợp tệ nhất

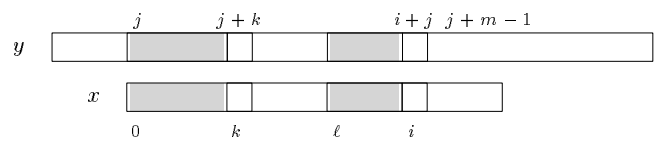
**Mô tả**

Apostolico-Crochemore sử dụng bảng dịch chuyển kmpNext để tính toán các lần dịch. Đặt l = 0 nếu x là lũy thừa của một ký tự đơn (x = c^m với c thuộc Σ) và l bằng vị trí của ký tự đầu tiên của x khác với x[0] ngược lại (x = a^l bu với a, b thuộc Σ , u thuộc Σ\* và a ≠ b). Trong mỗi lần thử, việc so sánh được thực hiện với mẫu các vị trí theo thứ tự sau: l, l + 1, …, m - 2, m - 1, 0, 1, …, l - 1.

Trong giai đoạn tìm kiếm, ta xem xét bộ ba có dạng (i; j; k) trong đó:

* cửa sổ được định vị trên hệ số văn bản y[j … j + m - 1];
* 0 <= k <= l và x[0 … k - 1] = y[j … j + k - 1];
* l <= i < m và x[l … i - 1] = y[j + l … i + j - 1].

Bộ ba ban đầu là (l; 0; 0)



Bây giờ chúng ta giải thích cách tính bộ ba tiếp theo sau khi (i; j; k) đã được tính toán. Ba trường hợp phát sinh tùy thuộc vào giá trị của i:

* i = l

Nếu x[i] = y[i + j] thì bộ ba tiếp theo là (i + 1; j; k).

Nếu x[i] ≠ y[i + j] thì bộ ba tiếp theo là (l; j + 1; max(0; k - 1)).

* l < i < m

Nếu x[i] = y[i + j] thì bộ ba tiếp theo là (i + 1; j; k).

Nếu x[i] ≠ y[i + j] thì phát sinh hai trường hợp tùy thuộc vào giá trị của kmpNext[i]:

* kmpNext[i] <= l: thì bộ ba tiếp theo là (l; i + j - kmpNext[i]; max(0; kmpNext[i])) ,
* kmpNext[i] > l: thì bộ ba tiếp theo là (kmpNext[i]; i + j - kmpNext[i]; l) .
* i = m

Nếu k < l và x[k] = y[j + k] thì bộ ba tiếp theo là (i; j; k + 1).

Nếu không thì k < l và x[k] ≠ y[j + k] hoặc k = l. Nếu k = l sự xuất hiện của x được báo cáo. Trong cả hai trường hợp, bộ ba tiếp theo là

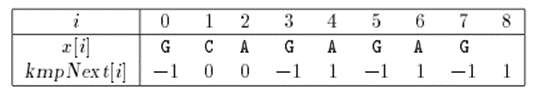
được tính theo cách tương tự như trong trường hợp l < i < m.

**Kiểm nghiệm**

Cho 2 xâu:

* X = GCAGAGAG, m = 8
* Y = GCATCGCAGAGAGTATACAGTACG, n = 24

Bảng kmpNext



l = 1

|  |  |  |
| --- | --- | --- |
| **Index bắt đầu** | **So sánh** | **Dịch (i - kmpNext(i))** |
| 0 | y = G**CA**TCGCA  x = G**CA**GAGAG | 4 |
| 4 | y = CGCAGAGA  x = GCAGAGAG | 1 |
| 5 | y = **GCAGAGAG**  x = **GCAGAGAG** | 7 |
| 12 | y = GTATACAG  x = GCAGAGAG | 1 |
| 13 | y = TATACAGT  x = GCAGAGAG | 1 |
| 14 | y = ATACAGTA  x = GCAGAGAG | 1 |
| 15 | y = TACAGTAC  x = GCAGAGAG | 1 |
| 16 | y = A**CAG**TACG  x = G**CAG**AGAG | 3 |

**Cài đặt**

def preKMP(*x*,*m*,*KmpNext*):

i = 0

j = *KmpNext*[0]

while i < *m*:

while j > -1 and *x*[i] != *x*[j]:

j = *KmpNext*[j]

i += 1

j += 1

if i < *m* and *x*[i] == *x*[j]:

*KmpNext*[i] = *KmpNext*[j]

else:

*KmpNext*[i] = j

def AXAMAC(*x*,*m*,*y*,*n*):

KmpNext = [-1 for i in range(*m* + 1)]

preKMP(*x*,*m*,KmpNext)

ell = 1

while ell < *m*:

if *x*[ell - 1] != *x*[ell]:

break

else:

ell += 1

if ell == *m*:

ell = 0

i = ell

j,k = 0,0

while j <= *n* - *m*:

while i < *m* and *x*[i] == *y*[i + j]:

i += 1

if i >= *m*:

while k < ell and *x*[k] == *y*[j + k]:

k += 1

if k >= ell:

print(f"Vị trí {j} xuất hiện vị trí của xâu x trong y")

j += i - KmpNext[i]

print(f"Dịch 1 khoảng = {i - KmpNext[i]} ({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]

x = 'GCAGAGAG'

y = 'GCATCGCAGAGAGTATACAGTACG'

m = len(x)

n = len(y)

AXAMAC(x,m,y,n)

# Tìm kiếm mẫu từ phải sang trái

## Thuật toán Boyer-Moore

**Đặc điểm**

* Thực hiện các so sánh từ phải sang trái
* Giai đoạn tiền xử lý có độ phức tạp thời gian và không gian O(m + σ)
* Giai đoạn tìm kiếm có độ phức tạp thời gian O(m x n)
* 3n lần so sánh ký tự trong văn bản trong trường hợp xấu nhất khi tìm kiếm cho một mẫu không chu kỳ
* Hiệu suất tốt nhất là O(n / m).

**Mô tả**

Thuật toán Boyer-Moore được coi là thuật toán tìm kiếm chuỗi hiệu quả nhất trong các ứng dụng thông thường.

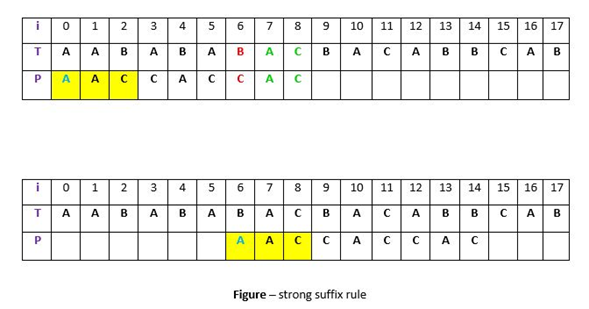
Thuật toán quét các ký tự của mẫu từ phải sang trái, bắt đầu từ ký tự bên phải nhất. Trong trường hợp không khớp (hoặc khớp hoàn toàn với toàn bộ mẫu), nó sử dụng hai hàm đã được tính toán trước để dịch cửa sổ sang phải. Hai hàm dịch này được gọi là hàm dịch hậu tố tốt - good-suffix shift và hàm dịch ký tự xấu - bad-character shift

*Good-suffix shift*

Hãy giả sử t là một chuỗi con của văn bản T khớp với chuỗi con của mẫu P. Bây giờ chúng ta dịch mẫu cho đến khi:

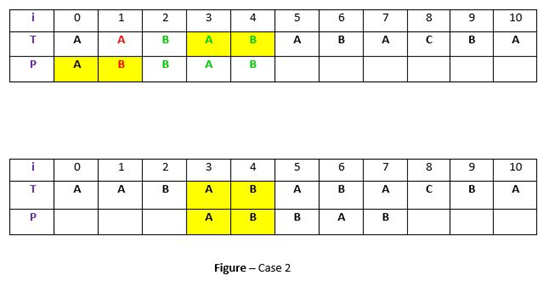
* Sự xuất hiện khác của t trong P khớp với t trong T.
* Một tiền tố của P, khớp với hậu tố của t.
* P di chuyển qua t

*Trường hợp 1*: Mẫu P có thể chứa nhiều sự xuất hiện khác nhau của t. Ta sẽ chọn xâu con gần bên phải nhất mà có ký tự liền trước ký tự đầu tiên khác với ký tự không khớp của mẫu

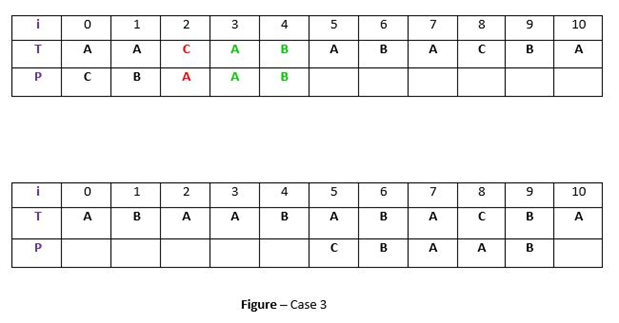


Trong hình, ta thấy 2 xâu không khớp ở ký tự C trong P, t = AC, ta cũng thấy có 3 sự xuất hiện của AC trong P. Chọn xâu AC thứ nhất để dịch vì nếu chọn xâu AC thứ 2 (ký tự liền trước ký tự đầu tiên là C trùng với ký tự không khớp của mẫu), ta sẽ lại lâm vào tình huống tương tự như trước.

*Trường hợp 2:* Không phải lúc nào cũng có khả năng rằng chúng ta sẽ tìm thấy sự xuất hiện của t trong P. Đôi khi không có sự xuất hiện nào.



*Trường hợp 3:* Nếu hai trường hợp trên không được thỏa mãn, chúng ta sẽ dịch mẫu qua t

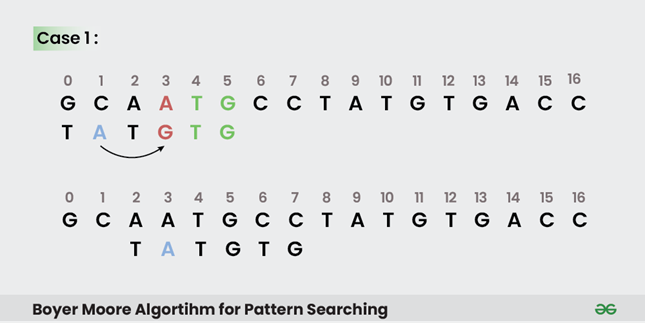


*Bad-character shift*

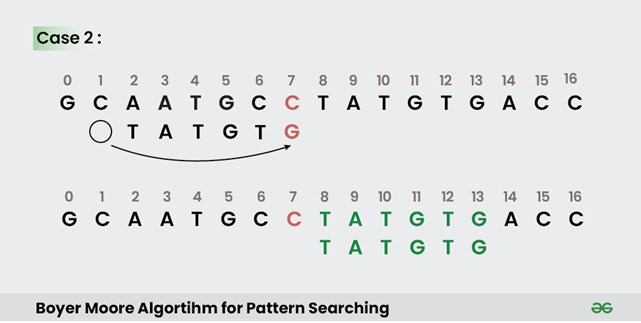
Ý tưởng của việc dịch ký tự xấu tương đối đơn giản. Ký tự của văn bản không khớp với ký tự hiện tại của mẫu được gọi là Ký tự Xấu. Trong trường hợp không khớp, chúng ta dịch mẫu cho đến khi:

* Sự không khớp trở thành một khớp .
* Mẫu P di chuyển qua ký tự không khớp.

*Trường hợp 1* (nếu ký tự có xuất hiện trong mẫu): Ta dịch sao cho ký tự không khớp trong y khớp với ký tự xuất hiện ngoài cùng bên phải trong x



*Trường hợp 2* (nếu ký tự không tồn tại trong mẫu)



Lưu ý rằng phép dịch ký tự xấu có thể âm (dịch lùi về) vì vậy để dịch cửa sổ, thuật toán Boyer-Moore áp dụng giá trị lớn nhất giữa dịch hậu tố tốt và dịch ký tự xấu. Cụ thể, hai hàm dịch được định nghĩa như sau.

Hàm dịch hậu tố tốt được lưu trữ trong một bảng bmGs có kích thước m+1.

Hãy xác định hai điều kiện:

* Cs(i, s): ∀ k | i < k < m, s >= k hoặc x[k-s]=x[k]
* Co(i, s): nếu s < i thì x[i-s] ≠ x[i]

Khi đó, ∀ i | 0 <= i < m: bmGs[i+1]=min(s), s>0 mà thỏa mãn: Cs(i, s) và Co(i, s) và chúng ta xác định bmGs[0] là độ dài của chu kỳ (period) của x.

Hàm dịch ký tự xấu được lưu trữ trong một bảng bmBc có kích thước σ. Đối ∀ c ∈ Σ: bmBc[c] = min(i) mà thỏa mãn : 1 <= i <m-1 và x[m-1-i]=c} nếu c xuất hiện trong x, m nếu không.

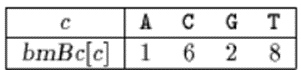
Cả hai bảng bmBc và bmGs có thể được tính toán trước trong thời gian O(m+ σ) và có độ phức tạp không gian O(m + σ). Độ phức tạp thời gian của giai đoạn tìm kiếm là bậc hai nhưng tối đa 3n lần so sánh ký tự văn bản được thực hiện khi tìm kiếm một mẫu không chu kỳ. Trên bảng chữ cái lớn (tương đối với độ dài của mẫu), thuật toán hoạt động rất nhanh.

**Kiểm nghiệm**

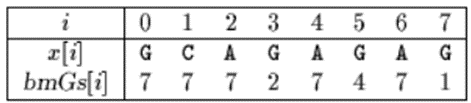
Cho 2 xâu:

* X = GCAGAGAG, m = 8
* Y = GCATCGCAGAGAGTATACAGTACG, n = 24

Bảng bmBc



Bảng bmGs



|  |  |  |
| --- | --- | --- |
| **Index bắt đầu** | **So sánh** | **Dịch (Max: bmGs[i], bmBc[c] – m + i)** |
| 0 | y = GCATCGCA  x = GCAGAGAG | Max(1, 1) |
| 1 | y = CATCGC**AG**  x = GCAGAG**AG** | Max(4, 4) |
| 5 | y = **GCAGAGAG**  x = **GCAGAGAG** | 7 (bmGs[0]) |
| 12 | y = GTATAC**AG**  x = GCAGAG**AG** | Max(4, 4) |
| 16 | y = ACAGTAC**G**  x = GCAGAGA**G** | Max(7, 5) |

**Cài đặt**

def preBmBc(*x*):

ASIZE = 256 # Assuming ASCII characters

m = len(*x*)

bmBc = [m] \* ASIZE

for i in range(m - 1):

bmBc[ord(*x*[i])] = m - i - 1

return bmBc

def preBmGs(*x*):

m = len(*x*)

period\_of\_x = ''

bmGs = [0] \* (m + 1)

# first element is the length of the period of x and will be removed when we returning the bmGs table

bmGs[0] = len(period\_of\_x)

# Helper function to check Cs condition

def Cs(*i*, *s*):

for k in range(*i* + 1, m):

if *s* >= k or *x*[k - *s*] == *x*[k]:

continue

else:

return False

return True

# Helper function to check Co condition

def Co(*i*, *s*):

if *s* >= *i*:

return True

if *x*[*i* - *s*] != *x*[*i*]:

return True

return False

# Calculate bmGs

for i in range(m):

for s in range(1, m + 1):

if Cs(i, s) and Co(i, s):

bmGs[i + 1] = s

break

return bmGs[1:]

def BM(*x*, *y*, *m*, *n*):

bmGs = preBmGs(*x*)

bmBc = preBmBc(*x*)

j = 0

while j <= *n* - *m*:

i = *m* - 1

while i >= 0 and *x*[i] == *y*[i + j]:

i -= 1

if i < 0:

print(f"Vị trí {j} xuất hiện vị trí của xâu x trong y")

j += bmGs[0]

print(f'Dịch {bmGs[0]}')

else:

print(f'Dịch theo Max({bmGs[i]}, {bmBc[ord(*y*[i + j])] - *m* + 1 + i})')

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

print()

x = 'GCAGAGAG'

y = 'GCATCGCAGAGAGTATACAGTACG'

m = len(x)

n = len(y)

BM(x, y, m, n)

## Thuật toán Turbo-Boyer-Moore

**Đặc điểm**

* Biến thể của thuật toán Boyer-Moore
* Không cần tiền xử lý bổ sung so với thuật toán Boyer-Moore
* Cần một không gian phụ cố định so với thuật toán Boyer-Moore
* Giai đoạn tiền xử lý có độ phức tạp thời gian và không gian O(m+σ)
* Giai đoạn tìm kiếm có độ phức tạp thời gian O(n)
* 2n lần so sánh ký tự văn bản trong trường hợp xấu nhất

**Mô tả**

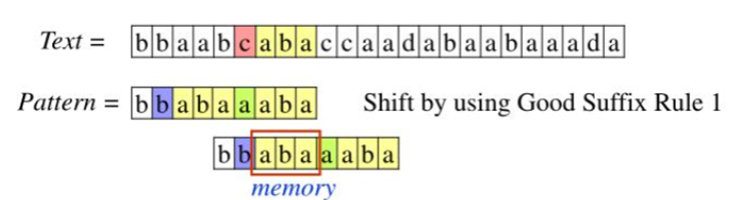
Thuật toán Turbo-BM là một sự cải tiến của thuật toán Boyer-Moore. Nó không cần tiền xử lý bổ sung và chỉ yêu cầu một không gian phụ cố định so với thuật toán Boyer-Moore gốc. Nó bao gồm việc ghi nhớ xâu con của văn bản đã khớp với một hậu tố của mẫu trong lần thử cuối cùng (và chỉ khi đã thực hiện một dịch hậu tố tốt).

Kỹ thuật này có hai lợi ích:

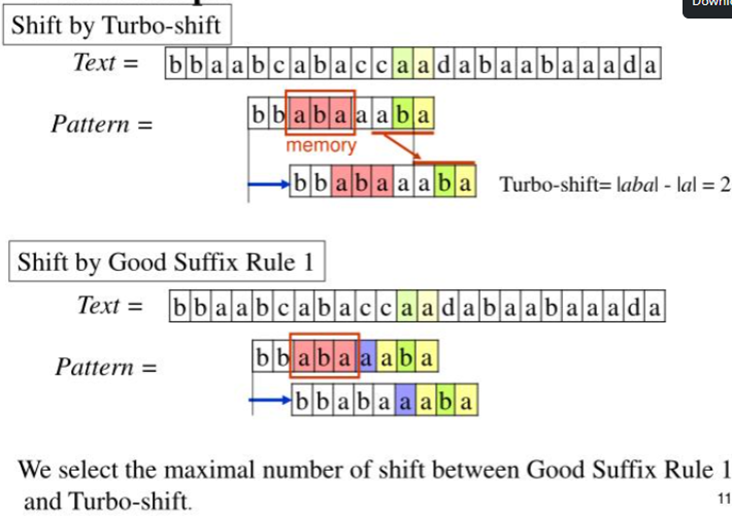
* Có thể bỏ qua xâu con này
* Nó có thể cho phép thực hiện một turbo-shift

Ta sẽ xem xét kỹ thuật này qua ví dụ sau:

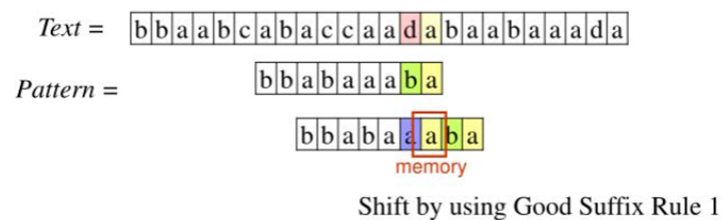
Phép dịch hậu tố tốt xảy ra:



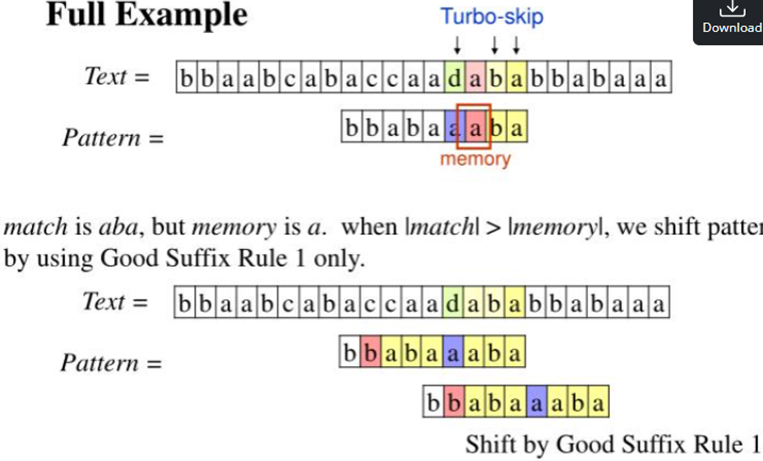
Điều này cho phép có thể thực hiện Turbo-Shift



Cả 2 đều cho kết quả là 2 nên ta chọn dịch hậu tố tốt



Chỉ dùng turbo-shift nếu |v| < |u|, nếu không dùng dịch hậu tố tốt

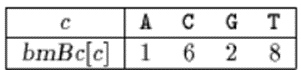


**Kiểm nghiệm**

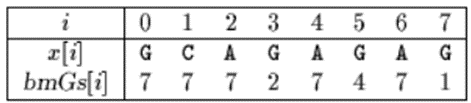
Cho 2 xâu:

* X = GCAGAGAG, m = 8
* Y = GCATCGCAGAGAGTATACAGTACG, n = 24

Bảng bmBc



Bảng bmGs



|  |  |  |
| --- | --- | --- |
| **Index bắt đầu** | **So sánh** | **Dịch** |
| 0 | y = GCATCGCA  x = GCAGAGAG | 1 |
| 1 | y = CATCGC**AG**  x = GCAGAG**AG**  => Mem = AG | 4 |
| 5 | y = **GC**AG**AGAG**  x = **GC**AG**AGAG**  Không cần so sánh AG vì Mem = AG | 7 |
| 12 | y = GTATAC**AG**  x = GCAGAG**AG** | 4 |
| 16 | y = ACAGTAC**G**  x = GCAGAGA**G** | 7 |

**Cài đặt**

def preBmBc(*x*):

ASIZE = 256 # Assuming ASCII characters

m = len(*x*)

bmBc = [m] \* ASIZE

for i in range(m - 1):

bmBc[ord(*x*[i])] = m - i - 1

return bmBc

def preBmGs(*x*):

m = len(*x*)

period\_of\_x = ''

bmGs = [0] \* (m + 1)

# first element is the length of the period of x and will be removed when we returning the bmGs table

bmGs[0] = len(period\_of\_x)

# Helper function to check Cs condition

def Cs(*i*, *s*):

for k in range(*i* + 1, m):

if *s* >= k or *x*[k - *s*] == *x*[k]:

continue

else:

return False

return True

# Helper function to check Co condition

def Co(*i*, *s*):

if *s* >= *i*:

return True

if *x*[*i* - *s*] != *x*[*i*]:

return True

return False

# Calculate bmGs

for i in range(m):

for s in range(1, m + 1):

if Cs(i, s) and Co(i, s):

bmGs[i + 1] = s

break

return bmGs[1:]

def TBM(*x*, *y*, *m*, *n*):

j = u = 0

shift = *m*

bmGs = preBmGs(*x*)

bmBc = preBmBc(*x*)

while j <= *n* - *m*:

i = *m* - 1

while i >= 0 and *x*[i] == *y*[i + j]:

i -= 1

if u != 0 and i == *m* - 1 - shift:

i -= u

if i < 0:

print(f"Vị trí {j} xuất hiện vị trí của xâu x trong y")

shift = bmGs[0]

u = *m* - shift

else:

v = *m* - 1 - i

turboShift = u - v

bcShift = bmBc[ord(*y*[i + j])] - *m* + 1 + i

shift = max(turboShift, bcShift, bmGs[i])

if shift == bmGs[i]:

u = min(*m* - shift, v)

else:

if turboShift < bcShift:

shift = max(shift, u + 1)

u = 0

j += shift

x = 'GCAGAGAG'

y = 'GCATCGCAGAGAGTATACAGTACG'

m = len(x)

n = len(y)

TBM(x, y, m, n)

## Thuật toán Zhu-Takaoka

**Đặc điểm**

* Biến thể của thuật toán Boyer-Moore
* Dùng 2 ký tự liên tiếp trong văn bản để tính toán phép dịch ký tự xấu
* Giai đoạn tiền xử lý có độ phức tạp thời gian và không gian O(m+σ2)
* Giai đoạn tìm kiếm có độ phức tạp thời gian O(n x m)

**Mô tả**

Zhu và Takaoka đã thiết kế một thuật toán thực hiện việc dịch bằng cách xem xét dịch ký tự xấu cho hai ký tự văn bản liên tiếp. Trong giai đoạn tìm kiếm, các so sánh được thực hiện từ phải sang trái và khi cửa sổ được đặt trên phần văn bản y[j … j+m-1] và một sự không khớp xảy ra giữa x[m-k] và y[j+m-k] trong khi x[m-k+1 … m-1]=y[j+m-k+1 … j+m-1], thì phép dịch được thực hiện với dịch ký tự xấu cho các ký tự văn bản y[j+m-2] và y[j+m-1]. Bảng dịch hậu tố tốt cũng được sử dụng để tính toán các phép dịch.

Giai đoạn tiền xử lý của thuật toán bao gồm tính toán sự xuất hiện ngoài cùng bên phải của ab trong x[0 .. m-2] cho mỗi cặp ký tự (a, b).

∀ a, b ∈ Σ, ztBc[a, b] = k (1 trong 3):

* k < m-2 và x[m-k ... m-k+1] = ab và ab không xuất hiện trong x[m-k+2 ... m-2]
* k = m-1 và x[0] = b và ab không xuất hiện trong x[0 ... m-2]
* k = m và x[0] ≠ b và ab không xuất hiện trong x[0 .. m-2]

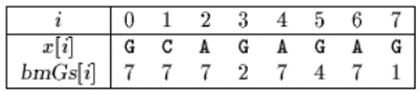
Thuật toán cũng bao gồm việc tính toán bảng bmGs. Giai đoạn tiền xử lý có độ phức tạp thời gian và không gian là O(m+σ2). Giai đoạn tìm kiếm có trường hợp xấu nhất là bậc hai.

**Kiểm nghiệm**

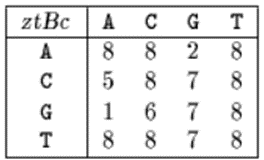
Cho 2 xâu:

* X = GCAGAGAG, m = 8
* Y = GCATCGCAGAGAGTATACAGTACG, n = 24

Bảng bmGs



Bảng ztBc



|  |  |  |
| --- | --- | --- |
| **Index bắt đầu** | **So sánh** | **Dịch (Max: bmGs[i], ztBc[y[j+m-2]][ y[j+m-1])** |
| 0 | y = GCATCGCA  x = GCAGAGAG | Max(1, 5) |
| 5 | x = GCAGAGAG  y = GCAGAGAG | 7 (bmGs[0[) |
| 12 | y = GTATAC**AG**  x = GCAGAG**AG** | Max(4, 2) |
| 16 | y = ACAGTAC**G**  x = GCAGAGA**G** | Max(7, 7) |

**Cài đặt**

def preZtBc(*x*):

ASIZE = 256 # Assuming ASCII characters

m = len(*x*)

# condition 3

ztBc = [[m for \_ in range(ASIZE)] for \_ in range(ASIZE)]

# condition 2

for i in range(ASIZE):

ztBc[i][ord(*x*[0])] = m - 1

# condition 1

for i in range(1, m - 1):

ztBc[ord(*x*[i - 1])][ord(*x*[i])] = m - 1 - i

return ztBc

def preBmGs(*x*):

m = len(*x*)

period\_of\_x = ''

bmGs = [0] \* (m + 1)

# first element is the length of the period of x and will be removed when we returning the bmGs table

bmGs[0] = len(period\_of\_x)

# Helper function to check Cs condition

def Cs(*i*, *s*):

for k in range(*i* + 1, m):

if *s* >= k or *x*[k - *s*] == *x*[k]:

continue

else:

return False

return True

# Helper function to check Co condition

def Co(*i*, *s*):

if *s* >= *i*:

return True

if *x*[*i* - *s*] != *x*[*i*]:

return True

return False

# Main loop to calculate bmGs

for i in range(m):

for s in range(1, m + 1):

if Cs(i, s) and Co(i, s):

bmGs[i + 1] = s

break

return bmGs[1:]

def ZT(*x*, *y*, *m*, *n*):

bmGs = preBmGs(*x*)

ztBc = preZtBc(*x*)

j = 0

while j <= *n* - *m*:

i = *m* - 1

while i >= 0 and *x*[i] == *y*[i + j]:

i -= 1

if i < 0:

print(f'Dịch {bmGs[0]}')

print(f"Vị trí {j} xuất hiện vị trí của xâu x trong y")

j += bmGs[0]

else:

print(f'Dịch theo Max({bmGs[i]}, {ztBc[ord(*y*[j + *m* - 2])][ord(*y*[j + *m* - 1])]})')

j += max(bmGs[i], ztBc[ord(*y*[j + *m* - 2])][ord(*y*[j + *m* - 1])])

print()

x = 'GCAGAGAG'

y = 'GCATCGCAGAGAGTATACAGTACG'

m = len(x)

n = len(y)

ZT(x,y,m,n)

## Thuật toán Berry-Ravindran

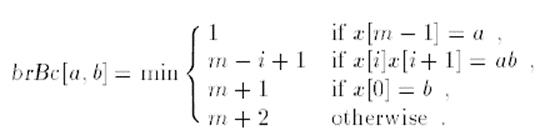
**Đặc điểm**

* Sự lai tạo giữa thuật toán Quick search và thuật toán Zhu - Takaota
* Giai đoạn tiền xử lý có độ phức tạp thời gian và không gian O(m+σ2)
* Giai đoạn tìm kiếm có độ phức tạp thời gian O(n x m)

**Mô tả**

Berry và Ravindran đã thiết kế một thuật toán thực hiện các phép dịch bằng cách xem xét dịch ký tự cho hai ký tự văn bản liên tiếp ngay bên phải của cửa sổ. Giai đoạn tiền xử lý của thuật toán bao gồm tính toán sự xuất hiện ngoài cùng bên phải của ab trong axb cho mỗi cặp ký tự (a, b).

∀ a, b ∈ Σ:



Giai đoạn tiền xử lý có độ phức tạp thời gian và không gian là O(m+σ2). Giai đoạn tìm kiếm có độ phức tạp thời gian O(n x m).

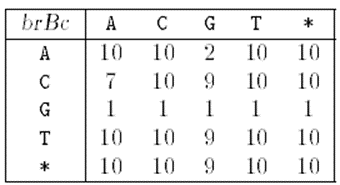
Sau một nỗ lực khi cửa sổ được đặt trên phần văn bản y[j .. j+m-1], một phép dịch có độ dài brBc[y[j+m],y[j+m+1]] được thực hiện. Ký tự văn bản y[n] bằng với ký tự null và y[n+1] được thiết lập thành ký tự null này để có thể tính toán các phép dịch cuối cùng của thuật toán.

**Kiểm nghiệm**

Cho 2 xâu:

* X = GCAGAGAG, m = 8
* Y = GCATCGCAGAGAGTATACAGTACG, n = 24

Bảng brBc



Ký tự \* đại diện cho tất cả ký tự còn lại trong bảng chữ cái

|  |  |  |
| --- | --- | --- |
| **Index bắt đầu** | **So sánh** | **Dịch (brBc[y[j+m]][ y[j+m+1])** |
| 0 | y = GCATCGCA**GA**  x = GCAGAGAG | 1 |
| 1 | y = CATCGCAG**AG**  x = GCAGAGAG | 2 |
| 3 | y = TCGCAGAG**AG**  x = GCAGAGAG | 2 |
| 5 | y = GCAGAGAG**TA**  x = GCAGAGAG | 10 |
| 15 | y = TACAGTA**Gnull**  x = GCAGAGAG | 1 |
| 16 | y = ACAGTACG**nullnull**  x = GCAGAGAG | 10 |

**Cài đặt**

def preBrBc(*x*):

ASIZE = 256 # Assuming ASCII characters

m = len(*x*)

# condition 4

brBc = [[m+2 for \_ in range(ASIZE)] for \_ in range(ASIZE)]

# condition 3

for i in range(ASIZE):

brBc[i][ord(*x*[0])] = m + 1

# condition 2

for i in range(m - 1):

brBc[ord(*x*[i])][ord(*x*[i + 1])] = m - i

# condition 1

for i in range(ASIZE):

brBc[ord(*x*[m - 1])][i] = 1

return brBc

def BR(*x*, *y*, *m*, *n*):

*y* += chr(0)

*y* += chr(0)

brBc = preBrBc(*x*)

j = 0

while j <= *n* - *m*:

if *x* == *y*[j : j + *m*]:

print(f"Vị trí {j} xuất hiện vị trí của xâu x trong y")

shift = brBc[ord(*y*[j + *m*])][ord(*y*[j + *m* + 1])]

j += shift

print(f'Dịch {shift}')

print()

x = 'GCAGAGAG'

y = 'GCATCGCAGAGAGTATACAGTACG'

m = len(x)

n = len(y)

BR(x, y, m, n)

# Tìm kiếm mẫu từ vị trí xác định

## Thuật toán Skip-Search

**Đặc điểm**

* Sử dụng các rổ (bucket) vị trí cho mỗi ký tự trong bảng chữ cái
* Giai đoạn tiền xử lý có độ phức tạp thời gian và không gian O(m + σ)
* Giai đoạn tìm kiếm có độ phức tạp thời gian O(m x n)
* Cần dự kiến O(n) lần so sánh ký tự văn bản

**Mô tả**

Đối với mỗi ký tự của bảng chữ cái, một rổ thu thập tất cả các vị trí của ký tự đó trong x. Khi một ký tự xuất hiện k lần trong mẫu, có k vị trí tương ứng trong rổ của ký tự đó. Khi từ ngắn hơn nhiều so với bảng chữ cái, nhiều rổ sẽ trống.

Giai đoạn tiền xử lý của thuật toán Skip Search bao gồm tính toán các rổ cho tất cả các ký tự của bảng chữ cái: ∀ c ∈ Σ, z[c] = {i : 0 <= i <= m-1 và x[i] = c}. Độ phức tạp về không gian và thời gian của giai đoạn tiền xử lý này là O(m + σ).

Vòng lặp chính của giai đoạn tìm kiếm bao gồm kiểm tra từng ký tự thứ m của văn bản, y[j] (do đó sẽ có n / m lần lặp chính). Đối với y[j], nó sử dụng mỗi vị trí trong rổ z[y[j]] để lấy được một vị trí khởi đầu (p) có thể của x trong y. Nó thực hiện một so sánh của x với y bắt đầu từ vị trí p, ký tự từng ký tự, cho đến khi có một không khớp, hoặc cho đến khi tất cả các ký tự khớp.

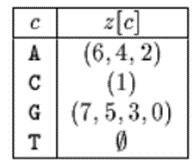
Thuật toán Skip Search có độ phức tạp thời gian xấu nhất là bậc hai nhưng số lần kiểm tra ký tự văn bản dự kiến là O(n).

**Kiểm nghiệm**

Cho 2 xâu:

* X = GCAGAGAG, m = 8
* Y = GCATCGCAGAGAGTATACAGTACG, n = 24

Bảng Z



|  |  |  |
| --- | --- | --- |
| **Index bắt đầu** | **So sánh** | |
| 7 | y = GCATCGC**A**  x = GCAGAGAG | + p = 7 - 6 = 1  y = CATCGCAG  x = GCAGAGAG    + p = 7 - 4 = 3  y = TCGCAGAG  x = GCAGAGAG    + p = 7 - 2 = 5  y = **GCAGAGAG**  x = **GCAGAGAG** |
| 15 | y = GAGAGTA**T**  x = GCAGAGAG |  |
| 23 | y = ACAGTAC**G**  x = GCAGAGAG | + p = 23 - 7 = 16  y = ACAGTACG  x = GCAGAGAG    + p = 23 - 5 = 18  y = AGTACG  x = GCAGAGAG    + p = 23 - 3 = 20  y = TACG  x = GCAGAGAG    + p = 23 - 0 = 23  y = G  x = GCAGAGAG |

**Cài đặt**

def build\_z\_table(*x*):

ASIZE = 256 # Assuming ASCII characters

m = len(*x*)

z = [[] for \_ in range(ASIZE)]

for i in range(m-1, -1, -1):

z[ord(*x*[i])].append(i)

return z

def search(*x*, *y*, *m*, *n*):

z = build\_z\_table(*x*)

for j in range(*m*-1, *n*, *m*):

print(f'Ở vị trí {j}')

print(f'y = {*y*[j - *m* + 1 : j + 1]}')

print(f'x = {*x*}')

for node in z[ord(*y*[j])]:

print(f'+ p = {j} - {node} = {j - node}')

print(f'y = {*y*[j - node : j - node + *m*]}')

print(f'x = {*x*}')

if *x* == *y*[j - node : j - node + *m*]:

print(f"Vị trí {j - node} xuất hiện vị trí của xâu x trong y")

print()

x = 'GCAGAGAG'

y = 'GCATCGCAGAGAGTATACAGTACG'

n, m = len(y), len(x)

search(x, y, m, n)

# Tìm kiếm mẫu từ vị trí ngẫu nhiên

## Thuật toán Horspool

**Đặc điểm**

* Sự đơn giản hóa của thuật toán Boyer-Moore
* Dễ triển khai
* Giai đoạn tiền xử lý có độ phức tạp thời gian O(m+σ) và không gian O(σ)
* Giai đoạn tìm kiếm có độ phức tạp thời gian O(m x n)
* Số lần so sánh trung bình cho một ký tự văn bản nằm giữa 1/σ và 2/( σ + 1)

**Mô tả**

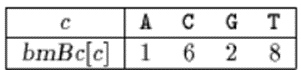
Dịch ký tự xấu trong thuật toán Boyer-Moore không hiệu quả lắm đối với các bảng chữ cái nhỏ, nhưng khi bảng chữ cái lớn so với độ dài của mẫu, như thường thấy trong bảng mã ASCII và các tìm kiếm thông thường được thực hiện dưới một trình soạn thảo văn bản, nó trở nên rất hữu ích. Sử dụng nó một mình tạo ra một thuật toán rất hiệu quả trong thực tế. Horspool đề xuất chỉ sử dụng phép dịch ký tự xấu của ký tự ngoài cùng bên phải của cửa sổ để tính toán các phép dịch trong thuật toán Boyer-Moore.

**Kiểm nghiệm**

Cho 2 xâu:

* X = GCAGAGAG, m = 8
* Y = GCATCGCAGAGAGTATACAGTACG, n = 24

Bảng bmBc



|  |  |  |
| --- | --- | --- |
| **Index bắt đầu** | **So sánh** | **Dịch (bmBc[y [j + m - 1]])** |
| 0 | y = GCATCGC**A**  x = GCAGAGAG | 1 |
| 1 | y = CATCGCA**G**  x = GCAGAGAG | 2 |
| 3 | y = TCGCAGA**G**  x = GCAGAGAG | 2 |
| 5 | y = GCAGAGA**G**  x = GCAGAGAG | 2 |
| 7 | y = AGAGAGT**A**  x = GCAGAGAG | 1 |
| 8 | y = GAGAGTA**T**  x = GCAGAGAG | 8 |
| 16 | y = ACAGTAC**G**  x = GCAGAGAG | 2 |

**Cài đặt**

def preBmBc(*x*):

ASIZE = 256 # Assuming ASCII characters

m = len(*x*)

bmBc = [m] \* ASIZE

for i in range(m - 1):

bmBc[ord(*x*[i])] = m - i - 1

return bmBc

def HS(*x*, *y*, *m*, *n*):

bmBc = preBmBc(*x*)

j = 0

while j <= *n* - *m*:

if *x* == *y*[j : j + *m*]:

print(f"Vị trí {j} xuất hiện vị trí của xâu x trong y")

print(f'Dịch {bmBc[ord(*y*[j + *m* - 1])]}\n')

j += bmBc[ord(*y*[j + *m* - 1])]

x = 'GCAGAGAG'

y = 'GCATCGCAGAGAGTATACAGTACG'

n, m = len(y), len(x)

HS(x, y, m, n)

## Thuật toán Quick search

**Đặc điểm**

* Sự đơn giản hóa của thuật toán Boyer-Moore
* Dễ triển khai
* Giai đoạn tiền xử lý có độ phức tạp thời gian O(m+σ) và không gian O(σ)
* Giai đoạn tìm kiếm có độ phức tạp thời gian O(m x n)
* Rất nhanh trong thực tế cho các mẫu ngắn và bảng chữ cái lớn

**Mô tả**

Thuật toán Quick Search chỉ sử dụng bảng dịch ký tự xấu. Sau một nỗ lực khi cửa sổ được đặt trên phần văn bản y[j ... j+m-1], độ dài của phép dịch ít nhất sẽ là một. Do đó, ký tự y[j+m] nhất định sẽ được sử dụng trong nỗ lực tiếp theo, và do đó có thể được sử dụng cho dịch chuyển ký tự xấu của nỗ lực hiện tại. Phép dịch ký tự xấu của thuật toán được điều chỉnh một chút để tính đến ký tự cuối cùng của x như sau:

∀ c ∈ Σ, qsBc[c]=min{i : 0 < i ≤ m và x[m-i]=c} nếu c xuất hiện trong x, ngược lại là m+1.

Giai đoạn tiền xử lý có độ phức tạp thời gian là O(m+ σ) và độ phức tạp không gian là O(σ).

Trong giai đoạn tìm kiếm, so sánh giữa các ký tự của mẫu và văn bản trong mỗi nỗ lực có thể được thực hiện theo bất kỳ thứ tự nào. Giai đoạn tìm kiếm có độ phức tạp thời gian xấu nhất là bậc hai nhưng nó có kết quả thực tế tốt.

**Kiểm nghiệm**

Cho 2 xâu:

* X = GCAGAGAG, m = 8
* Y = GCATCGCAGAGAGTATACAGTACG, n = 24

Bảng qsBc



|  |  |  |
| --- | --- | --- |
| **Index bắt đầu** | **So sánh** | **Dịch (qsBc[y [j + m]])** |
| 0 | y = GCATCGCA **G**  x = GCAGAGAG | 1 |
| 1 | y = CATCGCAG **A**  x = GCAGAGAG | 2 |
| 3 | y = TCGCAGAG **A**  x = GCAGAGAG | 2 |
| 5 | y = GCAGAGAG **T**  x = GCAGAGAG | 9 |
| 14 | y = ATACAGTA **C**  x = GCAGAGAG | 7 |

**Cài đặt**

def preQsBc(*x*):

ASIZE = 256 # Assuming ASCII characters

m = len(*x*)

qsBc = [m+1] \* ASIZE

for i in range(m):

qsBc[ord(*x*[i])] = m - i

return qsBc

def search(*x*, *y*, *m*, *n*):

qsBc = preQsBc(*x*)

j = 0

while j <= *n* - *m*:

if *x* == *y*[j : j + *m*]:

print(f"Vị trí {j} xuất hiện vị trí của xâu x trong y")

print(f'Dịch {qsBc[ord(*y*[j + *m*])]}\n')

j += qsBc[ord(*y*[j + *m*])]

x = 'GCAGAGAG'

y = 'GCATCGCAGAGAGTATACAGTACG'

n, m = len(y), len(x)

search(x, y, m, n)

## Thuật toán Tuned Boyer-Moore

**Đặc điểm**

* Sự đơn giản hóa của thuật toán Boyer-Moore
* Dễ triển khai
* Rất nhanh trong thực tế

**Mô tả**

Tuned Boyer-Moore là một phiên bản đơn giản hóa của thuật toán Boyer-Moore, nó rất nhanh trong thực tế. Phần tính toán mất thời gian nhất của một thuật toán so khớp chuỗi là kiểm tra xem ký tự của mẫu có khớp với ký tự của cửa sổ hay không. Để tránh làm phần này quá thường xuyên, có thể thực hiện một số dịch chuyển trước khi thực sự so sánh các ký tự. Thuật toán sử dụng hàm dịch ký tự xấu để tìm x[m-1] trong y và tiếp tục dịch cho đến khi tìm thấy nó, thực hiện ba lần dịch tù mù liên tiếp. Điều này yêu cầu lưu trữ giá trị của bmBc[x[m-1]] trong một biến dịch và sau đó đặt bmBc[x[m-1]] thành 0. Điều này cũng yêu cầu thêm m lần xuất hiện của x[m-1] vào cuối y. Khi x[m-1] được tìm thấy, m-1 ký tự cửa sổ khác được kiểm tra và áp dụng một dịch chuyển có độ dài là dịch

**Kiểm nghiệm**

Cho 2 xâu:

* X = GCAGAGAG, m = 8
* Y = GCATCGCAGAGAGTATACAGTACG, n = 24

Bảng bmBC



Shift = 2

|  |  |  |
| --- | --- | --- |
| **Index bắt đầu** | **So sánh** | **Dịch (shift)** |
| 0 | x = GCAGAGA**G**  y = GCATCGC**A**  Dịch bmBc[A]: y = CATCGCA**G**  Dịch bmBc[G]: y = CATCGCA**G** | 2 |
| 3 | x = GCAGAGA**G**  y = TCGCAGA**G**  Dịch bmBc[G]: y = TCGCAGA**G** | 2 |
| 5 | x = GCAGAGA**G**  y = GCAGAGA**G**  Dịch bmBc[G]: y = GCAGAGA**G** | 2 |
| 7 | x = GCAGAGA**G**  y = AGAGAGT**A**  Dịch bmBc[A]: y = GAGAGTA**T**  Dịch bmBc[T]: y = ACAGTAC**G**  Dịch bmBc[G]: y = ACAGTAC**G** | 2 |

**Cài đặt**

def preBmBc(*x*):

ASIZE = 256 # Assuming ASCII characters

m = len(x)

bmBc = [m] \* ASIZE

for i in range(m - 1):

bmBc[ord(x[i])] = m - i - 1

return bmBc

def TBM(*x*, *y*, *m*, *n*):

bmBc = preBmBc(x)

shift = bmBc[ord(x[m - 1])]

bmBc[ord(x[m - 1])] = 0

y += x[m - 1] \* m

j = 0

while j <= n-m:

print(f'index bắt đầu {j}, bắt đầu với y = {y[j : j + m]}')

k = bmBc[ord(y[j + m -1])]

if k == 0:

print(f'Dịch 0, y = {y[j : j + m]}')

while k != 0:

j += k

print(f'Dịch {k}, y = {y[j : j + m]}')

k = bmBc[ord(y[j + m -1])]

j += k

print(f'Dịch {k}, y = {y[j : j + m]}')

k = bmBc[ord(y[j + m -1])]

j += k

print(f'Dịch {k}, y = {y[j : j + m]}')

k = bmBc[ord(y[j + m -1])]

if x[ : m - 1] == y[j : j + m - 1]:

print(f"Vị trí {j} xuất hiện vị trí của xâu x trong y")

j += shift

print(f'Lần dịch cuối {shift}')

print()

x = 'GCAGAGAG'

y = 'GCATCGCAGAGAGTATACAGTACG'

n, m = len(y), len(x)

TBM(x, y, m, n)

## Thuật toán Raita

**Đặc điểm**

* Trước tiên so sánh ký tự cuối cùng của mẫu, sau đó là ký tự đầu tiên và cuối cùng là ký tự ở giữa trước khi thực sự so sánh các ký tự khác
* Thực hiện phép dịch như thuật toán Horspool
* Giai đoạn tiền xử lý có độ phức tạp thời gian O(m+σ) và độ phức tạp không gian O(σ)
* Giai đoạn tìm kiếm có độ phức tạp thời gian O(n x m)

**Mô tả**

Raita đã thiết kế một thuật toán mà ở mỗi lần thử trước hết so sánh ký tự cuối cùng của mẫu với ký tự văn bản nằm ở bên phải nhất của cửa sổ, sau đó nếu chúng khớp nhau thì so sánh ký tự đầu tiên của mẫu với ký tự văn bản nằm ở bên trái nhất của cửa sổ, sau đó nếu chúng khớp nhau thì so sánh ký tự ở giữa của mẫu với ký tự văn bản ở giữa của cửa sổ. Và cuối cùng nếu chúng khớp nhau thì thực sự so sánh các ký tự khác từ thứ hai đến gần cuối cùng, có thể so sánh lại ký tự ở giữa.

Raita quan sát thấy rằng thuật toán của mình có hành vi tốt trong thực tế khi tìm kiếm các mẫu trong văn bản tiếng Anh và ghi nhận hiện tượng này cho sự tồn tại của các phụ thuộc ký tự.

Smith đã tiến hành thêm một số thí nghiệm và kết luận rằng hiện tượng này có thể do các hiệu ứng của trình biên dịch.

Giai đoạn tiền xử lý của thuật toán Raita bao gồm tính toán hàm dịch ký tự xấu. Nó có thể được thực hiện trong thời gian O(m+σ) và không gian O(σ).

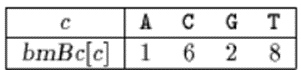
Giai đoạn tìm kiếm của thuật toán Raita có độ phức tạp thời gian xấu nhất là bậc hai.

**Kiểm nghiệm**

Cho 2 xâu:

* X = GCAGAGAG, m = 8
* Y = GCATCGCAGAGAGTATACAGTACG, n = 24

Bảng bmBc



|  |  |  |
| --- | --- | --- |
| **Index bắt đầu** | **So sánh** | **Dịch (bmBc[y[j+m-1]])** |
| 0 | y = GCATCGCA  x = GCAGAGAG | 1 |
| 1 | y = CATCGCA**G**  x = GCAGAGA**G** | 2 |
| 3 | y = TCGCAGA**G**  x = GCAGAGA**G** | 2 |
| 5 | y = **G**CAG**A**GA**G**  x = **G**CAG**A**GA**G** | 2 |
| 7 | y = AGAGAGTA  x = GCAGAGAG | 1 |
| 8 | y = GAGAGTAT  x = GCAGAGAG | 8 |
| 16 | y = ACAGTAC**G**  x = GCAGAGA**G** | 2 |

**Cài đặt**

def preBmBc(*x*):

ASIZE = 256 # Assuming ASCII characters

m = len(*x*)

bmBc = [m] \* ASIZE

for i in range(m - 1):

bmBc[ord(*x*[i])] = m - i - 1

return bmBc

def R(*x*, *y*, *m*, *n*):

bmBc = preBmBc(*x*)

firstC = *x*[0]

middleC = *x*[*m* // 2]

lastC = *x*[*m* - 1]

j = 0

while j <= *n* - *m*:

c = *y*[j + *m* - 1]

if *y*[j + *m* - 1] == lastC and *y*[j] == firstC and *y*[j + *m* // 2] == middleC:

if *x*[1 : *m* - 1] == *y*[j + 1 : j + *m* - 1]:

print(f"Vị trí {j} xuất hiện vị trí của xâu x trong y")

print(f'Dịch {bmBc[ord(c)]}\n')

j += bmBc[ord(c)]

x = 'GCAGAGAG'

y = 'GCATCGCAGAGAGTATACAGTACG'

n, m = len(y), len(x)

R(x, y, m, n)

## Thuật toán Smith

**Đặc điểm**

* Lấy giá trị lớn nhất của hàm dịch ký tự xấu của Horspool và hàm dịch ký tự xấu của Quick Search
* Giai đoạn tiền xử lý có độ phức tạp thời gian O(m+σ) và không gian O(σ)
* Giai đoạn tìm kiếm có độ phức tạp thời gian O(m x n)

**Mô tả**

Smith nhận thấy rằng tính toán dịch chuyển với ký tự văn bản ngay bên cạnh ký tự văn bản bên phải nhất của cửa sổ đôi khi cho phép dịch ngắn hơn so với việc sử dụng ký tự văn bản bên phải nhất của cửa sổ. Anh ấy đề xuất nên lấy giá trị lớn nhất giữa hai giá trị đó.

Giai đoạn tiền xử lý của thuật toán Smith bao gồm tính toán hàm dịch ký tự xấu và hàm dịch ký tự xấu của Quick Search.

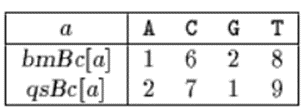
Giai đoạn tiền xử lý có độ phức tạp thời gian O(m+σ) và không gian O(σ). Giai đoạn tìm kiếm của thuật toán Smith có độ phức tạp thời gian xấu nhất là bậc hai.

**Kiểm nghiệm**

Cho 2 xâu:

* X = GCAGAGAG, m = 8
* Y = GCATCGCAGAGAGTATACAGTACG, n = 24

Bảng bmBc và qsBc



|  |  |  |
| --- | --- | --- |
| **Index bắt đầu** | **So sánh** | **Dịch (Max: bmBc[y[j + m - 1]], qsBc[y[j + m]])** |
| 0 | y = GCATCGC**A G**  x = GCAGAGAG | Max(1, 1) |
| 1 | y = CATCGCA**G A**  x = GCAGAGAG | Max(2, 2) |
| 3 | y = TCGCAGA**G A**  x = GCAGAGAG | Max(2, 2) |
| 5 | y = GCAGAGA**G T**  x = GCAGAGAG | Max(2, 9) |
| 14 | y = ATACAGT**A C**  x = GCAGAGAG | Max(1, 7) |

**Cài đặt**

def preBmBc(*x*):

ASIZE = 256 # Assuming ASCII characters

m = len(*x*)

bmBc = [m] \* ASIZE

for i in range(m - 1):

bmBc[ord(*x*[i])] = m - i - 1

return bmBc

def preQsBc(*x*):

ASIZE = 256 # Assuming ASCII characters

m = len(*x*)

qsBc = [m+1] \* ASIZE

for i in range(m):

qsBc[ord(*x*[i])] = m - i

return qsBc

def S(*x*, *y*, *m* ,*n*):

bmBc = preBmBc(*x*)

qsBc = preQsBc(*x*)

j = 0

while j <= *n* - *m*:

if *x* == *y*[j : j + *m*]:

print(f"Vị trí {j} xuất hiện vị trí của xâu x trong y")

print(f'Dịch theo Max({bmBc[ord(*y*[j + *m* - 1])]}, {qsBc[ord(*y*[j + *m*])]})\n')

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

x = 'GCAGAGAG'

y = 'GCATCGCAGAGAGTATACAGTACG'

n, m = len(y), len(x)

S(x, y, m, n)

## Thuật toán Brute Force

**Đặc điểm**

* không có giai đoạn tiền xử lý;
* không gian bổ sung hằng số cần thiết;
* luôn dịch chuyển cửa sổ sang bên phải đúng 1 vị trí;
* so sánh có thể được thực hiện theo bất kỳ thứ tự nào;
* pha tìm kiếm có độ phức tạp về thời gian O(m x n);
* 2n so sánh ký tự văn bản dự kiến.

**Mô tả**

Thuật toán Brute Force bao gồm việc kiểm tra, tại tất cả các vị trí trong văn bản từ 0 đến n - m, liệu sự xuất hiện của mẫu có bắt đầu ở đó hay không. Sau đó, sau mỗi lần thử, nó sẽ dịch chuyển mẫu đúng một vị trí sang phải.

Thuật toán Brute Force không yêu cầu giai đoạn tiền xử lý và một khoảng trống bổ sung không đổi ngoài mẫu và văn bản. Trong giai đoạn tìm kiếm, việc so sánh ký tự văn bản có thể được thực hiện theo bất kỳ thứ tự nào. Độ phức tạp về thời gian của giai đoạn tìm kiếm này là O(m x n). Số lượng so sánh ký tự văn bản dự kiến ​​là 2n

**Kiểm nghiệm**

Cho 2 xâu:

* X = GCAGAGAG, m = 8
* Y = GCATCGCAGAGAGTATACAGTACG, n = 24

Tìm kiếm:

|  |  |
| --- | --- |
| **Vị trí cửa sổ** | **So sánh** |
| 1 | y = **G C A** T C G C A  x = **G C A** G A G A G |
| 2 | y = C A T C G C A G  x = G C A G A G A G |
| 3 | y = A T C G C A G A  x = G C A G A G A G |
| 4 | y = T C G C A G A G  x = G C A G A G A G |
| 5 | y = C G C A G A G A  x = G C A G A G A G |
| 6 | y = **G C A G A G A G**  x = **G C A G A G A G** |
| 7 | y = C A G A G A G T  x = G C A G A G A G |
| 8 | y = A G A G A G T A  x = G C A G A G A G |
| 9 | y = **G** A G A G T A T  x = **G** C A G A G A G |
| 10 | y = A G A G T A T A  x = G C A G A G A G |
| 11 | y = **G** A G T A T A C  x = **G** C A G A G A G |
| 12 | y = A G T A T A C A  x = G C A G A G A G |
| 13 | y = **G** T A T A C A G  x = **G** C A G A G A G |
| 14 | y = T A T A C A G T  x = G C A G A G A G |
| 15 | y = A T A C A G T A  x = G C A G A G A G |
| 16 | y = T A C A G T A C  x = G C A G A G A G |
| 17 | y = A C A G T A C G  x = G C A G A G A G |

**Cài đặt**

def BF(*x*,*m*,*y*,*n*):

for j in range(0, *n* - *m* + 1):

if *x* == *y*[j:j+*m*]:

print(f"Vị trí {j} xuất hiện vị trí của xâu x trong y")

x = 'GCAGAGAG'

y = 'GCATCGCAGAGAGTATACAGTACG'

m = len(x)

n = len(y)

BF(x,m,y,n)