Thuật toán KMP:

**1. Ý tưởng thuật toán:**

Ta có bài toán như sau: Cho 1 chuỗi mẫu pattern P độ dài m và đoạn text T độ dài n, tìm tất cả các lần mà chuỗi P xuất hiện bên trong đoạn text T.

Với bài toán trên, khi sử dụng các giải đơn thuần (naive solution) thì ta phải duyệt lần lượt các ký tự trên T và so sánh các ký tự với P. Nếu ở vị trí i, P không xuất hiện trên T, thì ta tiếp tục duyệt lần lượt các ký tự của P trên T bắt đầu tại tại vị trí i + 1.

Tuy nhiên, việc làm như trên có thể mất thời gian và không hiệu quả vì khi P không xuất hiện tại vị trí i thì ta có thể kết luật nó cũng không thể xuất hiện ở vị trí i + 1 hoặc các vị trí tiếp theo.

*Ví dụ:*

T = “abcabcd”

P = “abcd”

P không thể xuất hiện trong T tại ví trí 0, nếu theo cách giải đơn thuần, ta tiếp tục xét vị trí 1, 2 điều này là không cần thiết vì P cũng không thể xuất hiện tại vị trí 1, 2 (P[0] T[1], P[0] T[2]).

Thuật toán KMP cải tiến bằng việc tiền xử lý chuỗi mẫu P để lưu lại thông tin về chuỗi tiền tố hậu tố dài nhất và sử dụng thông tin đó để đẩy nhanh quá trình tìm kiếm P xuất hiện trong T.

**2. Trình bày thuật toán:**

**2.1. Thuật toán:**

Lúc đầu, ta tìm chuỗi tiền tố hậu tố dài nhất (longest prefix suffix – LPS) tại vị trí i sao cho:

* Tiền tố độ dài k của P và hậu tố độ dài k của chuỗi [P0P1..Pi].
* Giá trị của k phải lớn nhất có thể.

Các giá trị trên được lưu trong mảng LSP[] trong đó LPS[i] là độ dài chuỗi tiền tố hậu tố lớn nhất của chuỗi [P0P1..Pi].

*Ví dụ:*

P = “AAAA” LPS[] = {0, 1, 2, 3}

P = “ABCDE” LPS[] = {0, 0, 0, 0, 0}

P = “AABAACAABAA” LPS[] = {0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5}

Ta có: LPS[0] = 0 vì với chuỗi 1 ký tự, chuỗi tiền tố hậu tố lớn nhất nhỏ hơn chính nó là chuỗi rỗng.

Với các phần tử LPS[i] (i > 0), ta xử lý như sau:

Trước hết ta đặt một biến tạm k = LPS[i-1]. Đó là độ dài của hậu tố đúng lớn nhất đang khớp ngay trước đó.

Chừng nào P[k] và P[i] chưa khớp nhau và tmp còn mang giá trị dương, ta rút ngắn hậu tố đúng cần đối chứng lại. Công thức rút ngắn sẽ là: k = LPS[k-1].

Do lệnh lặp ở trên nên sẽ xảy ra 2 khả năng:

* Nếu P[k] đã khớp với P[i], tức là ta đã tìm được hậu tố đúng thỏa mãn ở vị trí i. Ta tăng giá trị tmp thêm 1 đơn vị, và gán giá trị mới được xử lý này vào LPS[i].
* Nếu không, tức là k = 0, và không có hậu tố đúng thỏa mãn cho vị trí i. Như vậy rõ ràng LPS[i] = 0.

*Mã giả:*

lps[0] ← 0

k ← 0

i ← 1

while ( i < m )

if (P[i] == P[k])

k++

lps[i] = k

i++

else

if (k == 0)

lps[i]=0

i++

else

k = lps[k-1]

Tiếp theo, ta sẽ duyệt đoạn text T để tìm các vị trí P xuất hiện.

Ta đặt biến i = 0 và j = 0 rồi bắt đầu duyệt từ đầu đoạn T.

Khi P[j] và T[i] vẫn chưa khớp nhau và j còn mang giá trị dương, j = LPS[j-1].

Nghĩa là, ta đẩy xâu P sang phải cho tới khi nào tất cả các ký tự trước của P và T là trùng nhau, j là độ dài của hậu tố dài nhất của T{i} mà trùng khớp với tiền tố độ dài tương ứng của P.

Do đó lệnh lặp ở trên nên sẽ xảy ra 2 khả năng:

* Nếu P[j] đã khớp với T[i], tức là ta đã tìm được hậu tố đúng thỏa mãn ở vị trí i. Ta tăng giá trị j thêm 1 đơn vị.
  + Khi j = m, hậu tố độ dài |P| của T{i} đã khớp hoàn toàn với xâu P. Đặt lạ j = LPS[j-1] và tiếp tục quá trình tìm kiếm.
* Nếu không, tức là j = 0, và không có tiền tố nào của P khớp với hậu tố tương ứng cùng độ dài của xâu T{i}. Ta bỏ qua và duyệt tiếp tới i tiếp theo.

*Mã giả:*

i ← 0

j ← 0

result []

while ( i < n )

if ( P[j] == T[i] )

j++

i++

if ( i == m )

j = lps[j-1]

result.push(i-m)

else if ( i < n && P[j] != T[i] )

if ( j == 0 )

i++

else

j = lps[k-1]

return result

**2.2. Độ phức tạp thuật toán:**

Độ phức tạp của bước tiền xử lý là O(m)

Độ phức tạp của thuật toán là O(n).

Vậy độ phức tạp của thuật toán là O(m+n).

Tham khảo:

<https://codetube.vn/article/thuat-toan-kmp-tim-chuoi-con-ben-trong-chuoi-cha__mjzlIVUAGuOPGITwIPTAj/>

<https://thuytrangcoding.wordpress.com/2018/02/11/string-match-kmp/>