**Đối sánh chuỗi**: là thuật toán về các thao tác chuỗi

**Điều kiện**:

Mẫu cần tìm: P (pattern) ( chiều dài m )

Văn bản: T (text) ( chiều dài n )

m<=n

P và T có cùng tập hữu hạn kí tự. vd : {0,1} ; {A, B, C,…., Z}

**Ứng dụng:**

Trình soạn thảo văn bản

Input: Xâu mẫu P có độ dài m

Văn bản T có dộ dài n

Output: Tất cả vị trí xuất hiện của P trong T.

Sinh học phân tử

Input : T : CTCT Protein có m liên kết peptide, P : CTCT A.A

Output : Tất cả vị trí có CTCT A.A giống Input trong CTCT của protein

…

**Brute-force (Naive String-matching):**

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.

Lần lượt kiểm tra điều kiện P[0…m-1] = T[i...i+m-1] tại mọi vị trí có thể có i.

Vd: Tìm kiếm P = “aab” trong T = “acaabc”:

**Đặc điểm:**

* Là thuật toán tìm kiếm mẫu từ trái qua phải
* 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)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **a** | **C** | **a** | **a** | **b** | **c** |

**i=0 c !=a**

|  |  |  |
| --- | --- | --- |
| **a** | **A** | **b** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **a** | **C** | **a** | **a** | **b** | **C** |

**i=1, c!=a**

|  |  |  |
| --- | --- | --- |
| **A** | **a** | **b** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **a** | **C** | **a** | **a** | **b** | **c** |

**i=2 a=a b=b**

|  |  |  |
| --- | --- | --- |
| a | a | b |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **a** | **C** | **a** | **a** | **b** | **c** |

**i=3 b!=a**

|  |  |  |
| --- | --- | --- |
| a | a | b |

Input: P = “aab”, T = “acaabc”

Output: “aab” xuất hiện tại vị trí thứ i = 2

// không tìm thấy best case và worst case

Best case O(n)

Worst case O(n\*m)

Average case là O(n+m)

Hạn chế : Không ghi nhớ được thông tin đã trùng khớp (trước) khi xảy ra tình trạng không khớp.

Phải so sánh lại từ đầu (trên P) trong tất cả các trường hợp.

**Rabin-Karp**

Thuật toán Rabin-Karp giải quyết bài này bằng hàm băm (hash function): biểu diễn một xâu bất kỳ bằng một số nguyên (mã băm - hash value). Khi đó, việc so sánh hai xâu qui về so sánh hai mã băm tuơng ứng: PP xuất hiện trong TT tại vị trí ii nếu mã băm của PP bằng mã băm của T[i..(i+m−1)]T[i..(i+m−1)] (với i∈[0..(n−m)]i∈[0..(n−m)]).

Để tránh tràn số thì kết quả trên đuợc chia lấy dư cho một số qq, thuờng đuợc chọn là một số nguyên tố lớn. Nếu gọi tập các kí tự được sử dụng trong xâu là ΣΣ thì basebase thuờng đuợc chọn sao cho base=∣Σ∣base=∣Σ∣ hoặc cũng là một số nguyên tố lớn

Dễ thấy rằng độ phức tạp thời gian để tính mã băm của xâu độ dài kk mất O(k)O(k). Khi hiện thực thuật toán, ta sẽ “truợt” một thanh có độ dài mm trên xâu TT từ vị trí 00 đến n−mn−m để lấy ra tất cả các xâu con độ dài mm của TT và so sánh mã băm của chúng với mã băm của PP. Hàm băm đuợc sử dụng có một tính chất rất thú vị là nó giúp ta tính mã băm hihi của xâu hiện tại T[i..(i+m−1)]T[i..(i+m−1)] dựa trên mã băm h(i−1)h(i−1) của xâu ngay truớc đó T[(i−1)..(i+m)]T[(i−1)..(i+m)] chỉ trong thời gian O(1)O(1) thay vì tính lại trong thời gian O(m)O(m). Những hàm băm có tính chất này gọi là rolling hash:

***h­­i =(base×(hi−1−basem−1×T[i−1])+T[i+m−1])%q***

hash(‘426’) = 10 ( số thập phân)

hash(‘abc’) = 26 ( số chữ cái trong bảng chữ cái)

q: số nguyên tố lớn ( để tránh tràn số)

vd: P = ‘abc’

T = ‘aabcd’

hash(‘abc′)=97×262+98×261+99×260=68219.

|  |  |  |
| --- | --- | --- |
| a | b | c |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| a | a | b | c | d |

Hash(‘aab’) = 68193 (!=68219)

Hash(‘abc’) = 68219

Hash(‘bcd’) = 68922 (!=68219)

Input: P = ‘abc’ ,T = ‘aabcd’

Output :  ‘abc’ xuất hiện tại vị trí i =1.

Độ phức tạp: Best case và average case: O(m+n)

Worst case : O(m\*n): vd: P = ‘AAAA’ và T=’AAAAAAAAAAAAAAAA’

[**Thuật toán so khớp chuỗi**](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_so_kh%E1%BB%9Bp_chu%E1%BB%97i&action=edit&redlink=1)**Knuth–Morris–Pratt** ( thuật toán KMP) Thuật toán được xây dựng dựa vào quan sát rằng một xâu con chung của SS và TT sẽ đưa ra được thông tin SS có khớp với các vị trí sau của TT hay không. Bởi vì xâu con chung này đồng nghĩa với một phần của TT đã khớp với một phần của SS, nên bằng việc khởi tạo trước một số thông tin với xâu SS, ta sẽ thu được những kết luận về TT (nhờ xâu con chung) mà không cần quay ngược và so sánh lại những ký tự đã khớp.

Cụ thể hơn, ta muốn tính toán trước cách xâu SS tự khớp với chính nó. Nhờ vậy thuật toán sẽ không quay nhìn lại và chỉ duyệt qua TT một lần duy nhất.

Xây dựng bảng :

Gọi i là các vị trí vắt đầu sự đối sánh trên T, j là vị trí đang so sánh trên P.

i1 và j1 là vị trí mới trên T và và P thỏa: i+j = i1+j1

v = T[i1….i1+j1-1

Với mỗi vị trí j >0, giá trị của NEXT [j] (j1) là số lớn nhất (k<j) thỏa: k ký tự đầu khớp với k ký tự cuối cùng của chuỗi trước vị trí j ( nghĩa là: P(0..k-1) = P[j-k..j-1])

Nếu p[i] != p[j] thì NEXT [i] = [j]

Ngược lại: NEXT[i] = NEXT[j]

Vd: Input: P = ‘abcaby’, T=’abxabcabcaby’

Bảng Next:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
| KMP | 0 | 0 | 0 | 1 | 2 | 0 |

a b **x** a b c a b c a b y

a b **c** a b y

tại i = 2 và j = 2 thì ‘x’ != ‘c’, xét bảng: tại j = 1 thì KMP = 0 =>> xét lại tại j = 0

a b x a b c a b **c** a b y

a b c a b **y**

tại i = 8, j = 5 thì ‘c’ != ‘y’, xét bảng: tạo j = 4 thì KMP = 2 =>> xét lại tại j = 2

a b x a b c a b c a b y

c a b **y**

Input: P = ‘abcaby’, T=’abxabcabcaby’

Output: ‘abcaby’ xuất hiện tại vị trí thứ i = 6.

Độ phức tạp của thuật toán là O(n)

Bảng so sánh độ phúc tạp của 3 thuật toán:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Brute-force** | **Rabin-Karp** | **KMP** |
| Best case | O(n) | O(m+n) | O(n) |
| Worst case | O(n\*m) | O(m\*n) | O(n) |
| Average case | O(n+m) | O(m+n) | O(n) |

<https://nhannguyen95.github.io/rabin-karp-algorithm/>

Introduction to Algorithms (3 rdedition) chapter 32.