CÁC THUẬT TOÁN TÌM KIẾM TRÊN MẢNG VÀ CHUỖI

Bùi Tiến Lên

01/01/2017

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Bài toán tìm kiếm

- **Tìm kiếm** (*search*) là một công việc hàng ngày trong cuộc sống.
- Tìm kiếm là một trong những bài toán quan trọng trong các ứng dụng tin học.
- Một số thuật toán tìm kiếm phổ biến trên cấu trúc dữ liệu **mảng**
  - Tìm kiếm tuần tự
  - Tìm kiếm nhị phân
- Tìm kiếm chuỗi cũng được xem là một bài toán tìm kiếm trên mảng

TÌM KIẾM TRÊN MẢNG

Bài toán

Định nghĩa 1

Cho một dãy a có n phần tử. Hãy tìm xem một phần tử x có trong dãy a hay không? Bài toán này có thể yêu cầu thêm những thông tin: Tìm vị trí phần tử x trong mảng, có bao nhiều phần tử x trong mảng

Tìm kiếm tuần tự

Ý tưởng

Duyệt tuần tự từng phần tử trong mảng a và so sánh với phần tử x.

Tìm kiếm tuần tự (cont.)

```
1  int LinearSearch(int a[], int n, int x)
2  {
3      for (int i = 0; i < n; i++)
4          if (a[i] == x)
5              return i;
6      return -1;
7  }
```

Đánh giá độ phức tạp

- Đánh giá chi phí thực hiện dựa tham số n (số phần tử)

Trường hợp	**Hàm ước lượng** $O(g(n))$
tốt nhất	
trung bình	
xấu nhất	

Tìm kiếm nhị phân

Ý tưởng

Nếu các phần tử trong dãy có quan hệ thứ tự; nghĩa là có thể so sánh bằng, lớn hơn, nhỏ hơn giữa chúng. Ta có thể tổ chức lại dãy a để có thể tìm kiếm hiệu quả hơn.

1. Sắp xếp lại dãy a theo thứ tự tăng dần
2. Xét dãy $\{a_0, a_1, ..., a_{n-2}, a_{n-1}\}$, so sánh phần tử a_{mid} với x
   - **2.1** Nếu $a_{mid} = x$ thì tìm thấy
   - **2.2** Nếu $a_{mid} < x$ thì tiếp tục tìm trong dãy con $\{a_{mid+1}, ..., a_{n-1}\}$
   - **2.3** Nếu $a_{mid} > x$ thì tiếp tục tìm trong dãy con $\{a_0, ..., a_{mid-1}\}$

Tìm kiếm nhị phân (cont.)

Chương trình 1: Hàm cài đặt tìm nhị phân với giả thiết là dãy *a* đã được sắp thứ tự tăng dần

```
1  int BinarySearch(int a[], int n, int x)
2  {
3      int left = 0, right = n - 1, mid;
4      do
5      {
6          mid = (left + right) / 2;
7          if (x == a[mid])
8              return mid;
9          else if (x < a[mid])
10             right = mid - 1;
11         else
12             left = mid + 1;
13     } while (left <= right);
14     return -1;
15 }
```

Đánh giá độ phức tạp

- Đánh giá chi phí thực hiện dựa trên tham số n (số phần tử)

Trường hợp	**Hàm ước lượng** $O(g(n))$
tốt nhất	
trung bình	
xấu nhất	

TÌM KIẾM TRÊN CHUỖI

Bài toán tìm kiếm chuỗi

- Đối sánh chuỗi là một trong những bài toán cơ bản và tự nhiên nhất trong tin học
- Có rất nhiều ứng dụng liên quan đến bài toán đối sánh chuỗi
  - Chức năng tìm kiếm trong các trình soạn thảo văn bản, hoặc trình duyệt Web
  - Truy vấn cơ sở dữ liệu
  - Sinh học phân tử

Bài toán tìm kiếm chuỗi (cont.)

Phát biểu bài toán

Cho trước một chuỗi T có chiều dài n và một chuỗi P có chiều dài m. Tìm vị trí xuất hiện của P trong T

- Hầu hết các ngôn ngữ đều có cung cấp hàm thư viện để giải quyết bài toán này
  - strstr(...) trong C++
  - pos(...) trong Pascal

Lịch sử phát triển của thuật toán tìm kiếm chuỗi

- Phương pháp Brute Force được biết đến nhiều nhất. Độ phức tạp của thuật toán cho trường hợp xấu nhất $O(m.n)$ và cho trường hợp tốt nhất là $O(m+n)$
- [Cook, 1971] đã chứng minh một quả lý thuyết đưa ra sự tồn tại của một giải thuật để giải bài toán với độ phức tạp $O(m+n)$ cho trường hợp xấu nhất
- [Knuth et al., 1977] đã dựa trên cơ sở lý thuyết của Cook đã tìm ra một thuật toán tương đối đơn giản. Đồng thời Morris cũng đưa ra một thuật toán tương tự. Tuy nhiên họ đã không công bố thuật toán cho đến năm 1977.
- [Boyer and Moore, 1977] trong thời gian này cũng đã tìm ra một thuật toán nhanh hơn. Một trong những đặc điểm chung của các thuật toán này là đều rất phức tạp và khó nắm bắt

Lịch sử phát triển của thuật toán tìm kiếm chuỗi (cont.)

- [Karp and Rabin, 1987] cuối cùng đã đưa ra một thuật toán đơn giản gần như thuật toán Brute Force và có chi phí là $O(m+n)$

Lịch sử phát triển của thuật toán tìm kiếm chuỗi (cont.)

- Các thuật toán tiêu biểu
  - Brute Force
  - Rabin-Karp
  - Knuth-Morris-Pratt
  - Boyer-Moore
  - Boyer-Moore-Horspool
  - Apostolico-Giancarlo
  - Aho-Corasick

Thuật toán Brute-force

Ý tưởng

Tại ví trí thứ i của chuỗi T ta so sánh với từng phần tử của P tương ứng từ trái sang phải; nghĩa là, $(P_0, T_i), (P_1, T_{i+1})$...

Thuật toán Brute-force (cont.)

```
1  int BF_StringSearch(char *P, char *T)
2  {
3      int n = strlen(T);
4      int m = strlen(P);
5      for (int i = 0; i <= n - m; i++)
6      {
7          int j = 0;
8          while (j < m)
9              if (T[i + j] == P[j])
10                 j++;
11             else
12                 break;
13         if (j == m)
14             return i; // tim thay
15     }
16     return -1; // khong tim thay
17 }
```

Ví dụ 1

- Xét chuỗi T="AAAAAAAAAAAAAAAAH" và chuỗi P="AAAAH"
- Lần lặp thứ nhất của vòng lặp for

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	H

A	A	A	A	H

- Lần lặp thứ hai của vòng lặp for

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	H

A	A	A	A	H

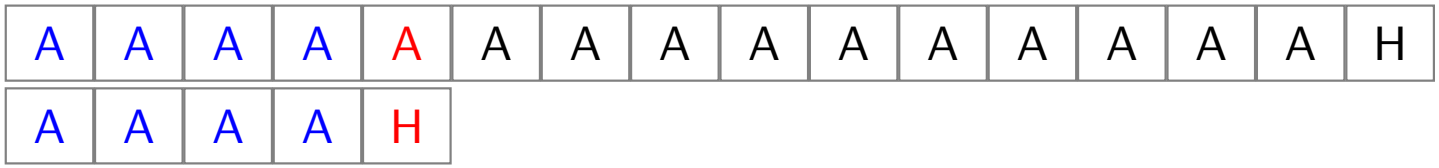
- ...
- Lần lặp cuối cùng của vòng lặp for

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	H

A	A	A	A	H

Ví dụ 1

- Xét chuỗi T="AAAAAAAAAAAAAAAH" và chuỗi P="AAAAH"
- Lần lặp thứ nhất của vòng lặp for

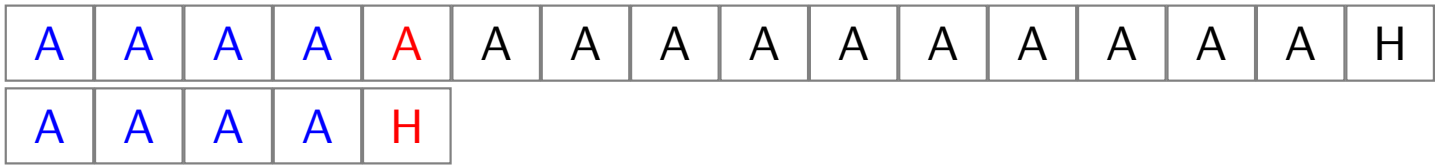


- Lần lặp thứ hai của vòng lặp for

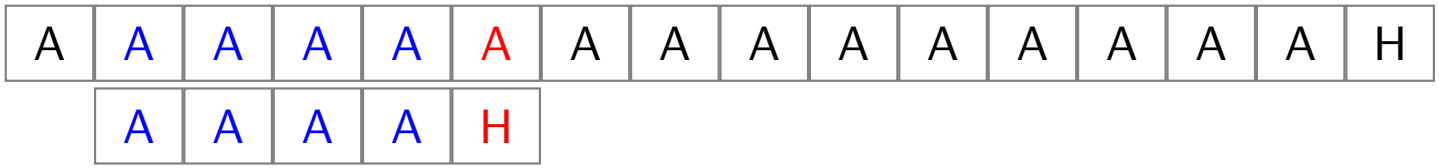
- ...
- Lần lặp cuối cùng của vòng lặp for

Ví dụ 1

- Xét chuỗi T="AAAAAAAAAAAAAAAH" và chuỗi P="AAAAH"
- Lần lặp thứ nhất của vòng lặp for



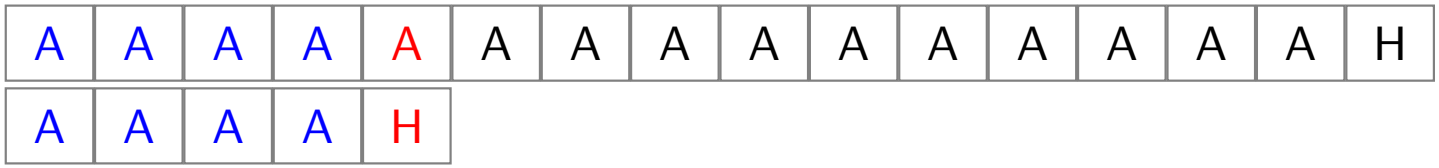
- Lần lặp thứ hai của vòng lặp for



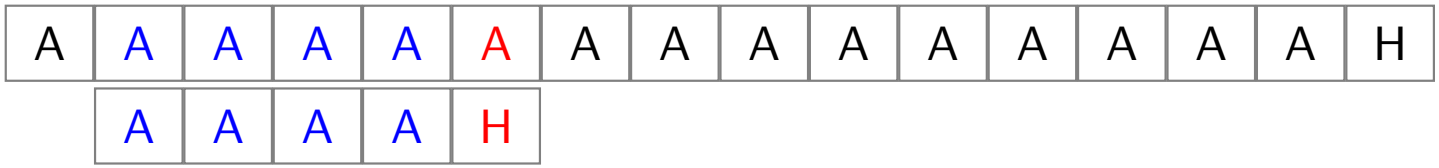
- ...
- Lần lặp cuối cùng của vòng lặp for

Ví dụ 1

- Xét chuỗi T="AAAAAAAAAAAAAAAH" và chuỗi P="AAAAH"
- Lần lặp thứ nhất của vòng lặp for



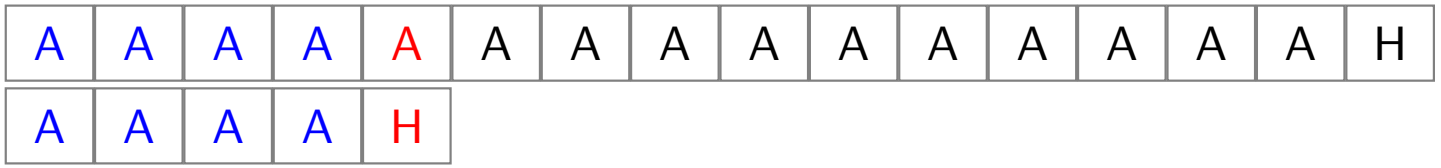
- Lần lặp thứ hai của vòng lặp for



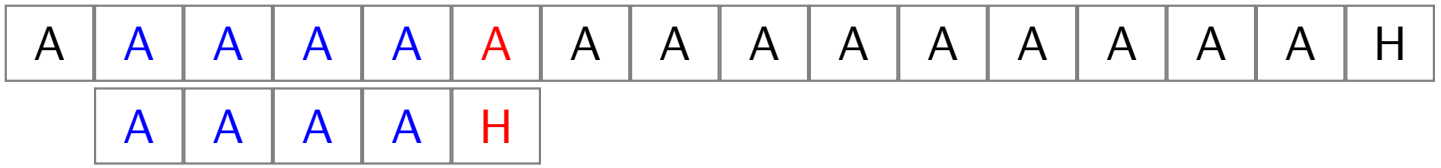
- ...
- Lần lặp cuối cùng của vòng lặp for

Ví dụ 1

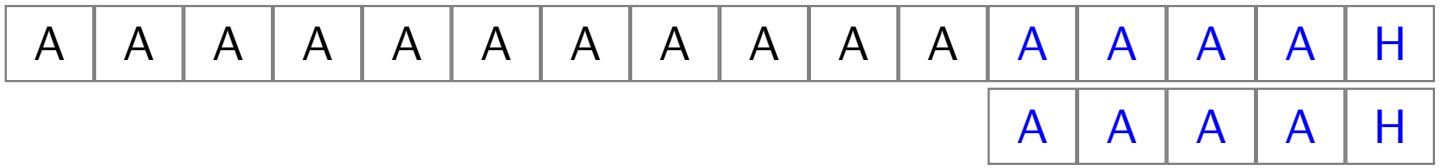
- Xét chuỗi T="AAAAAAAAAAAAAAAH" và chuỗi P="AAAAH"
- Lần lặp thứ nhất của vòng lặp for



- Lần lặp thứ hai của vòng lặp for



- ...
- Lần lặp cuối cùng của vòng lặp for



Ví dụ 2

- Xét chuỗi T="AAAAAAAAAAAAAAAH" và chuỗi P="OOOOH"
- Lần lặp thứ nhất của vòng lặp for

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	H
O	O	O	O	H											

- Lần lặp thứ hai của vòng lặp for

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	H
	O	O	O	O	H										

- ...
- Lần lặp cuối cùng của vòng lặp for

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	H
											O	O	O	O	H

Ví dụ 2

- Xét chuỗi T="AAAAAAAAAAAAAAAH" và chuỗi P="OOOOH"
- Lần lặp thứ nhất của vòng lặp for

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	H
O	O	O	O	H											

- Lần lặp thứ hai của vòng lặp for

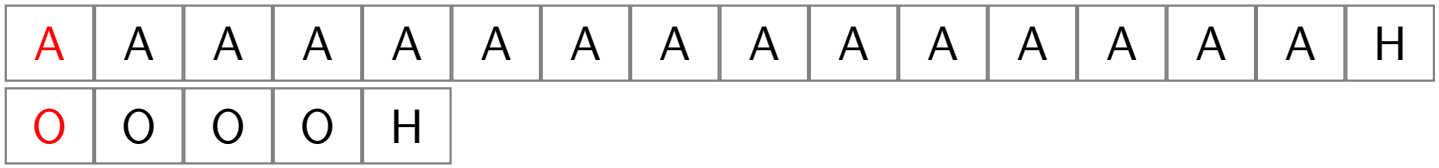
A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	H
	O	O	O	O	H										

- ...
- Lần lặp cuối cùng của vòng lặp for

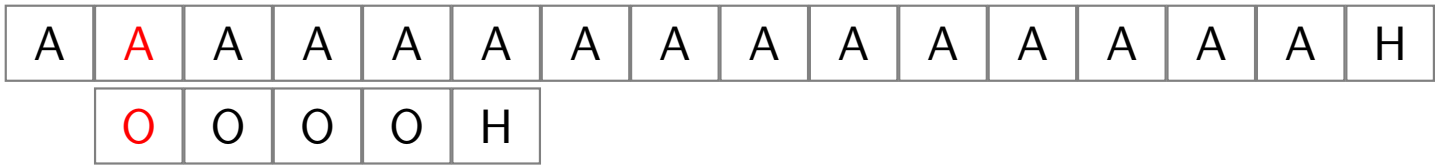
A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	H
											O	O	O	O	H

Ví dụ 2

- Xét chuỗi T="AAAAAAAAAAAAAAAH" và chuỗi P="OOOOH"
- Lần lặp thứ nhất của vòng lặp for



- Lần lặp thứ hai của vòng lặp for



- ...
- Lần lặp cuối cùng của vòng lặp for

Ví dụ 2

- Xét chuỗi T="AAAAAAAAAAAAAAAH" và chuỗi P="OOOOH"
- Lần lặp thứ nhất của vòng lặp for

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	H
O	O	O	O	H											

- Lần lặp thứ hai của vòng lặp for

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	H
	O	O	O	O	H										

- ...
- Lần lặp cuối cùng của vòng lặp for

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	H
											O	O	O	O	H

Ví dụ 2

- Xét chuỗi T="AAAAAAAAAAAAAAAH" và chuỗi P="OOOOH"
- Lần lặp thứ nhất của vòng lặp for

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	H
O	O	O	O	H											

- Lần lặp thứ hai của vòng lặp for

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	H
	O	O	O	O	H										

- ...
- Lần lặp cuối cùng của vòng lặp for

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	H
											O	O	O	O	H

Đánh giá độ phức tạp

- Đánh giá chi phí thực hiện dựa trên hai tham số n và m

Trường hợp	**Hàm ước lượng** $O(g(n,m))$
tốt nhất	
trung bình	
xấu nhất	

Thuật toán Morris-Pratt (MP)

Vấn đề của Brute-force

Trong thuật toán Brute-force: khi xảy ra không so khớp tại một vị trí nào đó, ta đã xóa bỏ tất cả các thông tin có được bởi các phép so sánh trước đó và bắt đầu lại việc so sánh tại vị trí đầu tiên của chuỗi P

- Hướng giải quyết của thuật toán MP
  - Sử dụng thông tin đã biết về các phần tử đã so sánh
  - Biến j thể hiện vị trí của phần tử hiện hành của mẫu P. Khi xảy không so khớp xảy ra, thay vì gán lại $j = 0$ để quay lại so sánh từ đầu chuỗi P thì ta sẽ gán j một giá trị thích hợp. Công thức xác định dựa trên bảng *Next N*

$$j \leftarrow N_j \tag{1}$$

Bảng Next

Khái niệm về chuỗi con tiền tố và hậu tố

Chuỗi con tiền tố và hậu tố của một chuỗi "ABCDE"

- Các chuỗi con tiền tố là "A", "AB", "ABC" và "ABCD"
- Các chuỗi con hậu tố là "E", "DE", "CDE" và "BCDE"

Bảng Next

Định nghĩa 2

Bảng Next

- Mỗi phần tử của chuỗi P sẽ tương ứng với một phần tử trong bảng N_j
- Nếu $j = 0$ thì $N_j = -1$
- Nếu $j > 0$ thì $N_j =$ chiều dài của chuỗi con chung dài nhất giữa các tập chuỗi con tiền tố và hậu tố của chuỗi $P_{0,2,\ldots,j-1}$

Cài đặt

Chương trình 2: Xây dựng bảng `Next` của thuật toán MP

```
1  void MP_CreateNext(char *P, int N[]) {
2      int m, i, j;
3      m = strlen(P);
4      N[0] = -1;
5      i = 0;
6      j = -1;
7      while (i < m-1) {
8          if ((j == -1) || (P[i] == P[j])) {
9              i++;
10             j++;
11             N[i] = j;
12         }
13         else
14             j = N[j];
15     }
16 }
```

Cài đặt (cont.)

Chương trình 3: Thuật toán MP

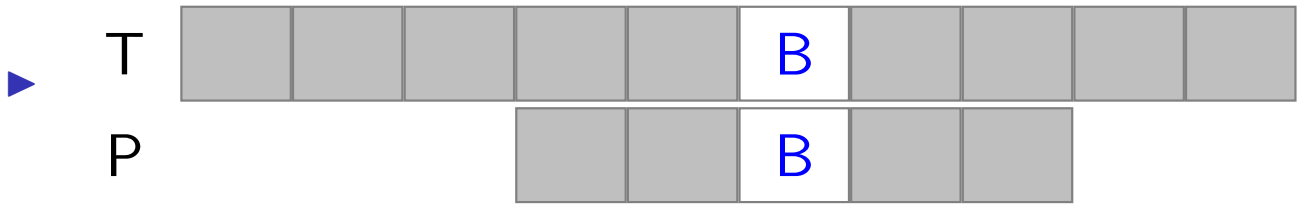
```
1  int MP_StringSearch(char *P, char *T)
2  {
3      // Tao bang Next N
4      MP_CreateNext(P, N);
5      // Tim
6      int n = strlen(T);
7      int m = strlen(P);
8      int i = 0, j=0;
9      while(i<n) {
10         if(T[i]==P[j]) {
11             i++; j++;
12             if(j==m) return i-j; // tim thay
13         } else {
14             if(j>0) j=N[j];
15             else i++;
16         }
17     }
```

Cài đặt (cont.)

```
18      return -1 ; // khong tim thay
19  }
```

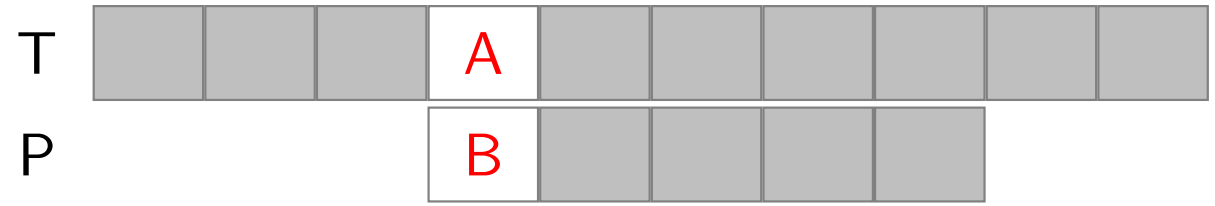
Giải thích thuật toán

Xét phần tử T_i và P_j

- Trường hợp 1: hai phần tử này khớp nhau thì dịch i và j sang phải một đơn vị
- 

Tiếp tục

- Trường hợp 2: hai phần tử không khớp nhau
  - Trường hợp 2.1: Nếu $j = 0$ thì i dịch sang bên phải một đơn vị và j giữ nguyên



  - Trường hợp 2.2: Nếu $j > 0$ thì i giữ nguyên j thay đổi theo công thức của bảng *Next*

Minh họa thuật toán

- Xét chuỗi T = "AATAAAATA" và mẫu P = "AAATA"
- Bảng *Next* của mẫu P là

-1	0	1	2	0

Tiếp tục

- Bảng `Next` của mẫu P là

-1	0	1	2	0

- Bắt đầu $i = 0, j = 0$

A	A	T	A	A	A	A	T	A
A	A	A	T	A				

- Trường hợp $1 \rightarrow i = 1, j = 1$

A	A	T	A	A	A	A	T	A
A	A	A	T	A				

- Trường hợp $1 \rightarrow i = 2, j = 2$

A	A	T	A	A	A	A	T	A
A	A	A	T	A				

Tiếp tục

- Bảng `Next` của mẫu P là

-1	0	1	2	0

- Trường hợp 2.2 $\rightarrow i = 2, j = 1$

A	A	T	A	A	A	A	T	A
	A	A	A	T	A			

- Trường hợp 2.2 $\rightarrow i = 2, j = 0$

A	A	T	A	A	A	A	T	A
		A	A	A	T	A		

- Trường hợp 2.1 $\rightarrow i = 3, j = 0$

A	A	T	A	A	A	A	T	A
			A	A	A	T	A	

Tiếp tục

- Bảng `Next` của mẫu P là

-1	0	1	2	0

- Trường hợp 1 $\rightarrow i = 4, j = 1$

A	A	T	A	A	A	A	T	A
			A	A	A	T	A	

- Trường hợp 1 $\rightarrow i = 5, j = 2$

A	A	T	A	A	A	A	T	A
			A	A	A	T	A	

- Trường hợp 1 $\rightarrow i = 6, j = 3$

A	A	T	A	A	A	A	T	A
			A	A	A	T	A	

Tiếp tục

- Bảng `Next` của mẫu P là

-1	0	1	2	0

- Trường hợp 2.2 $\rightarrow i = 6, j = 2$

A	A	T	A	A	A	A	T	A
				A	A	A	T	A

- Trường hợp 1 $\rightarrow i = 7, j = 3$

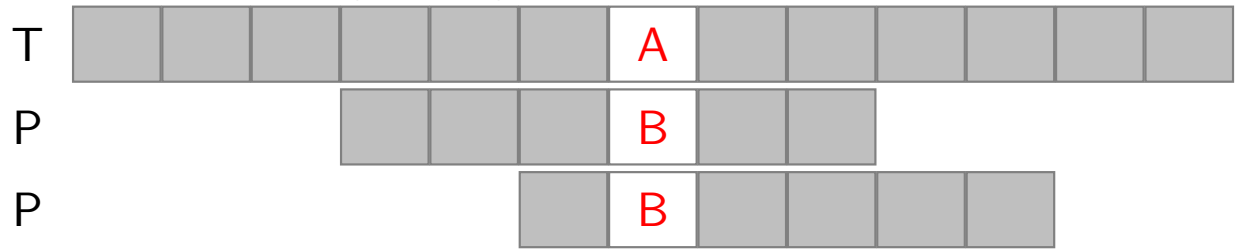
A	A	T	A	A	A	A	T	A
				A	A	A	T	A

- Trường hợp 1 $\rightarrow i = 8, j = 4$

A	A	T	A	A	A	A	T	A
				A	A	A	T	A

Thuật toán Knuth-Morris-Pratt (KMP)

- Điểm yếu trong bảng `Next` của MP



- Thuật toán Knuth-Morris-Pratt là sự cải tiến của Thuật toán Morris-Pratt
- Cải tiến được thực hiện trong việc tính bảng `Next`

Cài đặt

Cài đặt cải tiến cách tính bảng Next

```
1  void KMP_CreateNext(char *P, int N[]) {
2      int m, i, j;
3      m = strlen(P);
4      N[0] = -1;
5      i = 0;
6      j = -1;
7      while (i < m-1) {
8          if ((j == -1) || (P[i] == P[j])) {
9              i++; j++;
10             if(P[i] != P[j]) N[i] = j;
11             else N[i] = N[j];
12         }
13         else
14         j = N[j];
15     }
16 }
```

Đánh giá độ phức tạp

- Đánh giá chi phí thực hiện dựa trên hai tham số n và m

Trường hợp	**Hàm ước lượng** $O(g(n,m))$
tốt nhất	
trung bình	
xấu nhất	

Thuật toán Rabin-Karp (RK)

Nguyên lý

- Sử dụng kỹ thuật so sánh từng chuỗi con của T với mẫu P như thuật toán Brute-force
- Sử dụng hàm băm để so sánh chuỗi. Hai chuỗi giống nhau phải có giá trị hàm băm giống nhau, hai chuỗi khác nhau thì phải có giá trị hàm băm khác nhau

Hàm băm

Công thức

Công thức hàm băm phổ biến cho một chuỗi ký tự S có m ký tự

$$f(S, m) = S_0 d^{m-1} + S_1 d^{m-2} + ... + S_{m-1} \quad (2)$$

Với S_i là mã ASCII của ký tự chỉ số i trong chuỗi và d là thừa số của hàm băm thường là số nguyên tố.

Ví dụ 1

- Tính giá trị hàm băm của chuỗi "hi" với $d = 101$
- Mã ASCII của 'h' và 'i' là 104 và 105. Áp dụng công thức

$$f("hi", 2) = 104 * 101 + 105$$

- Giá trị hàm băm của chuỗi "hi" là 10609

Một cách tính hàm băm hiệu quả

Dựa trên công thức ta có cách tính hàm băm

- Cho chuỗi con m phần tử của T bắt đầu tại i là

$$f\left(T\left(i\right),m\right)=T_{i}d^{m-1}+T_{i+1}d^{m-2}+...+T_{i+m-1}$$

- Cho chuỗi con m phần tử của T bắt đầu tại $i+1$

$$f\left(T\left(i+1\right),m\right)=T_{i+1}d^{m-1}+T_{i+2}d^{m-2}+...+T_{i+m}$$

- Vậy ta có công thức

$$f\left(T\left(i+1\right),m\right)=f\left(T\left(i\right),m\right)d-T_{i}d^{m}+T_{i+m} \quad (3)$$

Đánh giá độ phức tạp

- Đánh giá chi phí thực hiện dựa trên hai tham số n và m

Trường hợp	**Hàm ước lượng** $O(g(n,m))$
tốt nhất	
trung bình	
xấu nhất	

Tài liệu tham khảo


Boyer, R. S. and Moore, J. S. (1977).
A fast string searching algorithm.
Communications of the ACM, 20(10):762–772.


Cook, S. A. (1971).
The complexity of theorem-proving procedures.
In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM.


Karp, R. M. and Rabin, M. O. (1987).
Efficient randomized pattern-matching algorithms.
IBM Journal of Research and Development, 31(2):249–260.


Knuth, D. E., Morris, Jr, J. H., and Pratt, V. R. (1977).
Fast pattern matching in strings.
SIAM journal on computing, 6(2):323–350.