

**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT**

**THÀNH PHỐ HỒ CHÍ MINH**

**KHOA: CÔNG NGHỆ THÔNG TIN**

**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT**

**THÀNH PHỐ HỒ CHÍ MINH**

**KHOA: CHÍNH TRỊ VÀ LUẬT**

Logo, icon

Description automatically generated

**Sinh viên thực hiện**

|  |  |  |
| --- | --- | --- |
| STT | Họ và tên | MSSV |
| 1 | Nguyễn Thái Văn | 21110939 |
| 2 | Huỳnh Nhật Nam | 21110903 |
| 3 | Nguyễn Trọng Phúc | 21110915 |

**FAST PATTERN MATCHING IN STRINGS**

Môn học: Cấu trúc dữ liệu và giải thuật

**TIỂU LUẬN cuối kỳ**

**môn học: TƯ TƯỞNG HỒ CHÍ MINH**

*Tp. Hồ Chí Minh, tháng 12 năm 2022*

*Tp. Hồ Chí Minh, tháng 11 năm 2022*

*GIỚI THIỆU:*

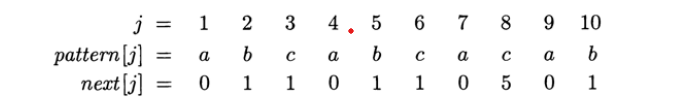
Trong lập trình, chắc hẳn ít nhiều chúng ta cũng nghe về vấn đề xét sự so khớp chuỗi, cụ thể là chúng ta xem thử chuỗi này có đang nằm trong chuỗi kia hay không. Và khi gặp những vấn đề như thế này, những bạn sinh viên CNTT ban đầu đều suy nghĩ ra một phương pháp đó chính là chúng ta sẽ trượt chuỗi cần xét trên chuỗi kia, và mỗi lần như thế chúng ta sẽ so sánh từng kí tự của hai chuỗi với nhau, nếu như chuỗi mẫu hoàn toàn nằm trong chuỗi kia thì chúng ta return true và kết thúc chương trình, còn nếu tại một vị trí mà kí tự của 2 chuỗi khác nhau thì chúng ta sẽ ngừng vòng lặp và tiến hành tịnh tiến chuỗi mẫu lên thêm 1 lần nữa cho đến khi nó khớp hoặc đến cuối vị trí của chuỗi kia. Tuy nhiên cách làm này thực sự không hiệu quả với những chuỗi có kích thước lớn, vì độ phức tạp của cách làm này là O(n \* m) trong đó n là độ dài chuỗi mẫu và m là độ dài chuỗi còn lại. Vì vậy trong chapter này chúng ta sẽ cùng nghiên cứu phương pháp để giảm thiểu độ phức tạp của thuật toán so khớp chuỗi này cụ thể độ phức tạp của thuật toán trong Chapter này sẽ là O(n + m).

*THUẬT TOÁN:*

Trước tiên, ta gọi+ pattern là chuỗi cần đi so khớp , gọi j là index của các kí tự chuỗi pattern

+ text là chuỗi mà để chuỗi pattern đi so khớp, gọi k là index của các kí tự trong chuỗi text

Trước khi chúng ta tiến hành so khớp chuỗi, chúng ta sẽ đi tạo ra mảng next trước, mảng next là mảng mà tại vị trí j của chuỗi next[j] chính là vị trí cần so sánh tiếp theo nếu như tại vị trí j này next[j] != text[k].

Bản chất của mảng next[j] này đó chính là tại một vị trí j mà pattern[j] != text[k] thì chúng ta không cần phải dịch pattern lên thêm 1 đơn vị để so sánh lại mà chúng ta có thể dịch pattern lên j – next[j] đơn vị, nhờ vậy mà độ phức tạp của thuật toán được giảm đi đáng kể. Đây chính là một ví dụ mẫu về mảng next[j].

Và cách để tính toán mảng next này chỉ tốn O(m) bước với m là độ dài của chuỗi pattern và trước khi đi tính mảng next chúng ta sẽ cùng nhau tạo ra mảng f, mảng f ở đây chính là tiền đề để tạo ra mảng next.

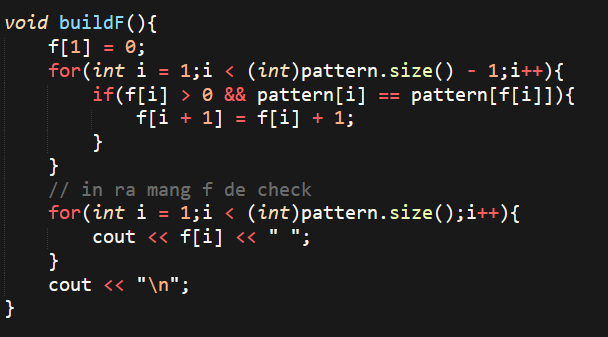
Bây giờ, mục đích chính của mảng next đó chính là lưu lại vị trí i lớn nhất mà chuỗi từ pattern[1] …pattern[i – 1] đã khớp với text để mà khi đó chúng ta sẽ so sánh text[k] với pattern[next[j]] thôi và hiệu quả là chúng ta đã bỏ qua i – 1 vị trí. Để dễ dàng hơn trong việc tạo ra mảng next, chúng ta sẽ đi tạo ra mảng f, mảng f ở đây chính là tại một vị trí j bất kì f[j] = i trong đó i là lớn nhất và nhỏ hơn j sao cho



Và chúng ta thấy điều kiện này hoàn toàn đúng với i bằng 1, vì vậy chúng ta sẽ có f[j] >= 1 với mọi j > 1. Theo quy ước, chúng ta sẽ để f[1] = 0. Và giờ chúng ta sẽ đi tạo ra mảng f



*CODE C++*

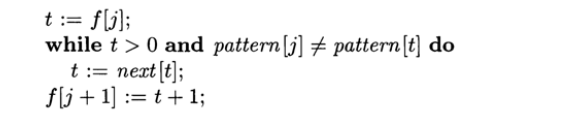


Đây chính là một vấn đề đã từng xuất hiện ở trước đây cụ thể đó chính là ta tính j lớn nhất nhỏ hơn k sao cho

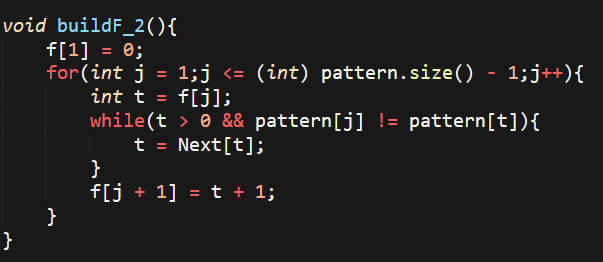
pattern[1] … pattern[j – 1] = text[k – j + 1] … text[k – 1]

vì vậy chúng ta sẽ sử dụng nó cho vấn đề hiện tại. Và khi chúng ta thực hiện thuật toán trên, chúng ta đã có thể tính toán được mảng f rồi.

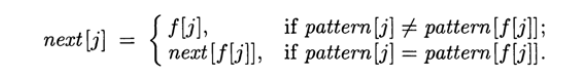
Và em bổ sung thêm một chương trình để tính mảng f, chương trình này sẽ tính f[j + 1] và giả sử rằng f[j] và next[1] …next[j – 1] đã được tính. Thuật toán:

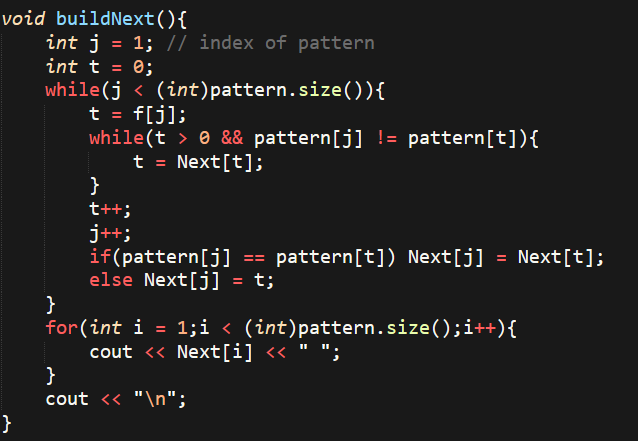


*CODE C++*



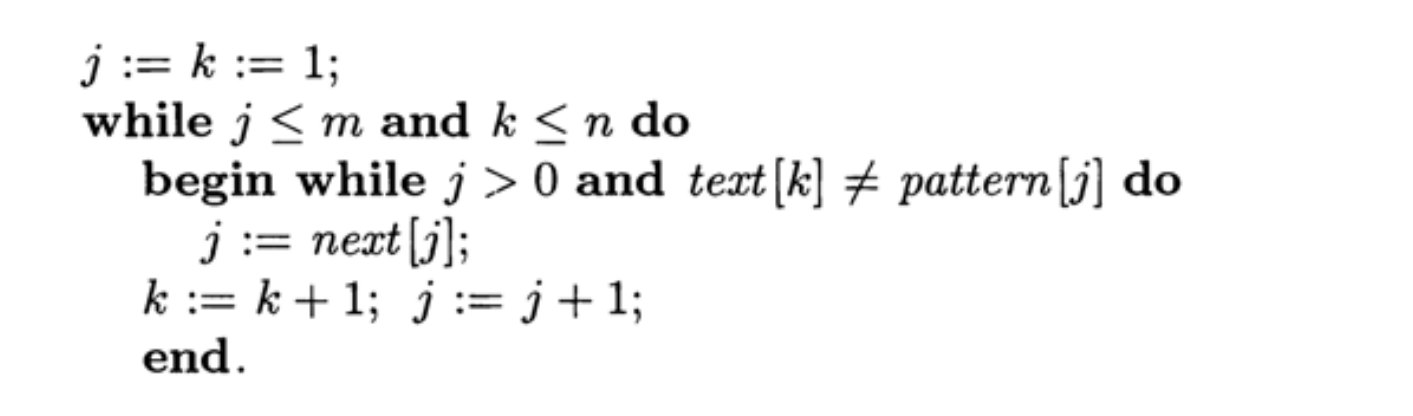
Và sau khi chúng ta đã tính toán được mảng f, chúng ta sẽ cùng nhau đi tạo mảng next, chúng ta có công thức sau:



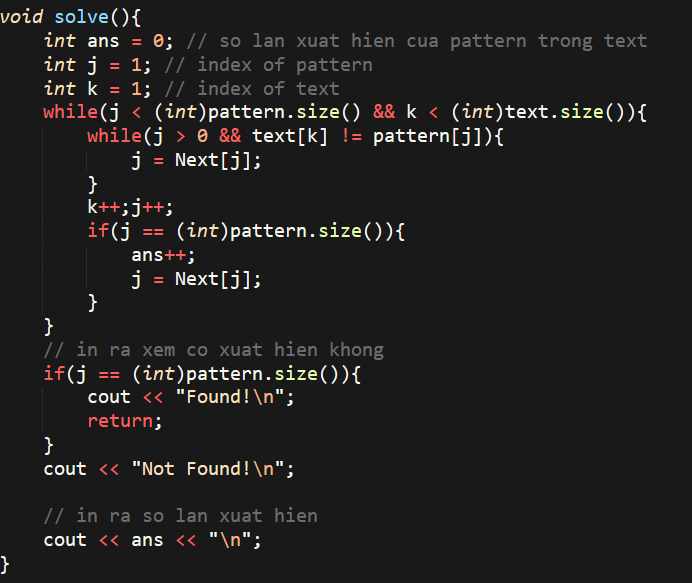
Thuật toán xây dựng mảng next: *CODE C++*

Độ phức tạp thuật toán: O(m) với m là độ dài của chuỗi pattern

Và sau khi chúng ta đã có mảng next, việc duyệt so khớp sẽ diễn ra một cách dễ dàng:



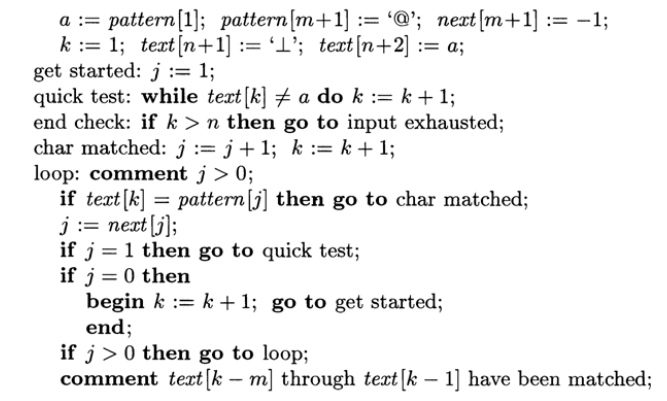
Đây là thuật toán trong bài báo, và thuật toán này chỉ để xét xem chuỗi pattern có xuất hiện trong chuỗi text không hay thôi, và em sẽ mở rộng ra để xét thêm chuỗi pattern xuất hiện trong chuỗi text bao nhiêu lần. Thuật toán:

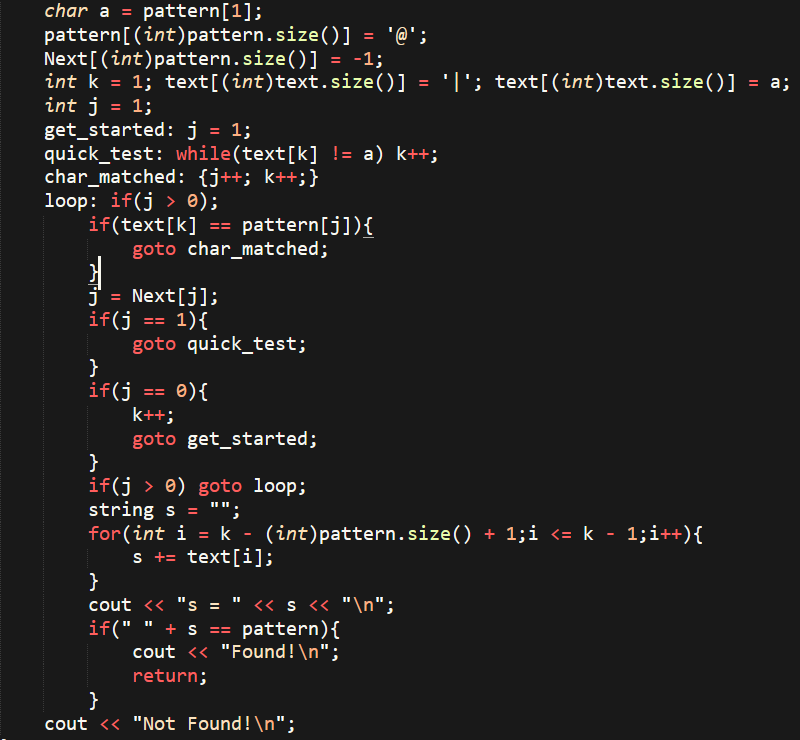
*CODE C++*

Độ phức tạp của thuật toán: O(m + n) với m là độ dài của chuỗi pattern và n là độ dài của chuỗi text.

*Cải tiến độ hiệu quả:*

Chúng em vừa mới trình bày thuật toán so khớp chuỗi ở trên, nhưng hình thức này dường như không hiệu quả lắm. Và bây giờ chúng ta sẽ đi tìm hiểu một thuật toán mới được triển khai tốt hơn, và đây chính là thuật toán:



*CODE C++*

*ỨNG DỤNG CỦA THUẬT TOÁN*

1. *Tìm kiếm văn bản:* Thuật toán Knuth-Morris-Pratt (KMP) có thể được sử dụng để tìm kiếm một mẫu trong chuỗi văn bản một cách hiệu quả bằng cách tránh phải quay lại và khớp lại các ký tự đã khớp trước đó.

2*. Nén dữ liệu:* Thuật toán KMP có thể được sử dụng trong các thuật toán nén dữ liệu như thuật toán LZ77 và LZ78 để tìm các mẫu lặp lại trong một chuỗi dữ liệu một cách hiệu quả.

3. *Truyền dữ liệu:* Thuật toán KMP có thể được sử dụng trong các giao thức sửa lỗi để truyền dữ liệu nhằm xác định và sửa lỗi hiệu quả trong dữ liệu được truyền.

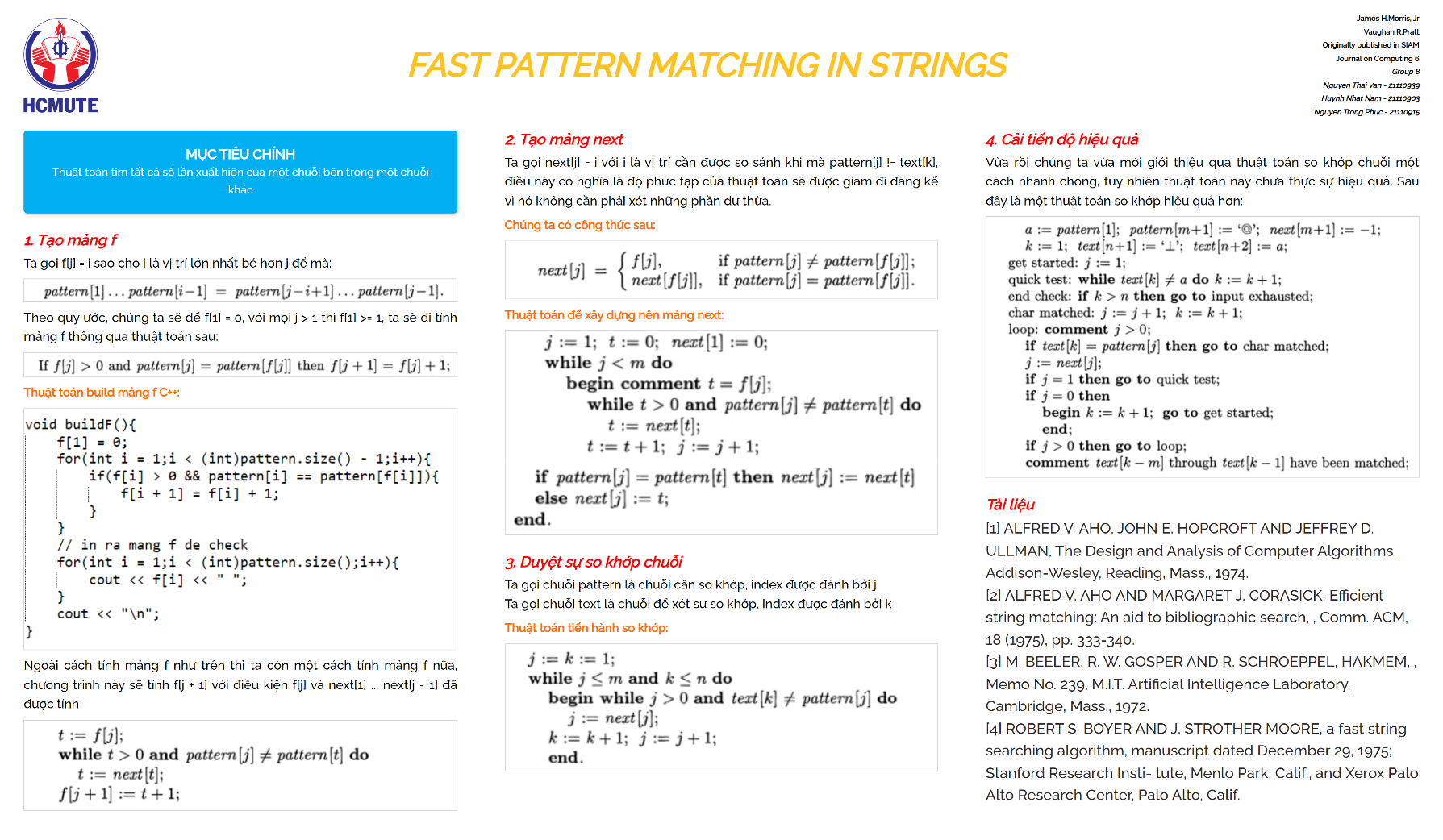
4. *Phân tích bộ gen:* Thuật toán KMP có thể được sử dụng trong tin sinh học để tìm kiếm hiệu quả các mẫu trong chuỗi DNA.

5. *Định tuyến mạng:* Thuật toán KMP có thể được sử dụng trong các thuật toán định tuyến cho mạng máy tính để tìm đường đi ngắn nhất giữa hai nút trong mạng một cách hiệu quả.

*HƯỚNG PHÁT TRIỂN CỦA THUẬT TOÁN*

Trước khi đi vào vấn đề so khớp chuỗi ở Chapter 9 này, việc đầu tiên chúng ta cần phải tạo ra mảng next, mảng next ở đây rất quan trọng vì nó giúp chúng ta có thể bỏ qua 1 vài vị trí không cần phải xét nữa vì lí do chúng ta đã biết nó chắc chắn khớp hoặc là chắc chắn không khớp rồi. Và để xây dựng mảng next 1 cách hiệu quả thì chúng ta sẽ tiến hành đi xây dựng mảng f trước, mảng f ở đây mỗi vị trí j nó sẽ lưu lại số i lớn nhất để mà chuỗi pattern[1] … pattern[i – 1] = pattern[j – i + 1] … pattern[j – 1]. Việc chúng ta xây dựng mảng f như vậy với lí do nếu như có sự bằng nhau giữa 2 chuỗi trên nghĩa là có một tiền tố đúng bằng với hậu tố đúng, mà ở đây vì tiền tố đúng này trùng với hậu tố đúng và chuỗi tới vị trí j – 1 đã hoàn toàn khớp nên vì thế chúng ta có thể bỏ qua xét các vị trí của tiền tố này mà chúng ta sẽ bắt đầu xét chữ cái tiếp theo đứng sau tiền tố ấy. Việc tạo ra mảng f chỉ cần một vòng lặp với độ phức tạp O(m) với m là độ dài của chuỗi pattern. Về phần thuật toán thì chúng em đã trình bày ở trên. Sau khi có được mảng f chính xác rồi thì chúng ta sẽ đi tạo mảng next, chúng ta có một công thức để tạo mảng next thông qua mảng f như trên và công thức này cũng để tìm vị trí tiếp theo cần so sánh, và thuật toán để tìm mảng next cũng chỉ có độ phức tạp O(m) với m là độ dài của chuỗi pattern. Sau khi có mảng next chúng em sẽ tiến hành đi so khớp, nếu gặp một vị trí mà pattern[j] != text[k] thì em sẽ so sánh text[k] với patter[next[j]] chứ chúng ta không cần dịch chuyển pattern lên 1 đơn vị và so sánh lại từ đầu. Thuật toán so khớp cũng diễn ra với độ phức tạp xấu nhất là O(n) với n là độ dài của chuỗi text khi mà chuỗi pattern không tìm thấy trong text hoặc chuỗi pattern ở cuối của text. Ở trong bài báo người ta chỉ xét là có xuất hiện chuỗi pattern trong text hay không, ở đây em mở rộng thêm là chuỗi pattern xuất hiện bao nhiêu lần và những vị trí mà chuỗi pattern đó xuất hiện. Để làm đc việc này thì khi mà j > m (m là độ dài của chuỗi pattern) thì tức là chuỗi pattern nó đã xuất hiện trong chuỗi text, khi đó em sẽ tăng biến đếm lên (biến đếm này để đếm số lần xuất hiện của chuỗi pattern trong text) và vị trí của chuỗi pattern xuất hiện sẽ là k – m + 1, bởi vì chúng ta cần in ra vị trí của pattern[1] nên vì thế chúng ta cần phải trừ đi độ dài của chuỗi pattern và cộng thêm 1

*POSTER*

**