CHƯƠNG 4. KỸ THUẬT XỬ LÝ CHUỖI

4.1 Một số khái niệm

4.1.1 Chuỗi kí tự

Chuỗi kí tự, hay còn gọi là xâu kí tự, là một dãy các kí tự viết liền nhau. Trong đó, các kí tự được lấy từ bảng chữ cái ASCII. Chuỗi kí tự được hiểu là một mảng 1 chiều chứa các kí tự.

Cách khai báo chuỗi kí tự như sau:

```
char s[100];
```

hoặc

```
char *s = new char[100];
```

Ví dụ trên là khai báo một chuỗi kí tự s có độ dài tối đa 100 kí tự, trong đó chuỗi s có tối đa 99 bytes tương ứng 99 kí tự có ý nghĩa trong chuỗi, và byte cuối cùng lưu kí tự kết thúc chuỗi là '\0'. Kí hiệu '\0' là kí tự bắt buộc dùng để kết thúc một chuỗi.

Hằng xâu kí tự được ghi bằng cặp dấu nháy kép. Ví dụ, "Hello". Nếu giữa cặp dấu nháy kép không ghi kí tự nào thì ta có chuỗi rỗng và độ dài chuỗi rỗng bằng 0.

4.1.2 Nhập/ xuất chuỗi kí tự

Trong ngôn ngữ lập trình C, ta có thể sử dụng hàm scanf với kí tự định dạng là %s để nhập một chuỗi kí tự do người dùng nhập vào từ bàn phím vào chương trình.

```
char str[100];
scanf("%s", &str);
```

Nhược điểm của hàm scanf khi nhập nội dung chuỗi kí tự có khoảng trắng thì kết quả lưu trong chuỗi không đúng như người dùng mong muốn. Khi nhập chuỗi có chứa kí tự khoảng trằng thì biên kiểu chuỗi chỉ lưu được phần đầu chuỗi đến khi gặp khoảng trắng đầu tiên, phần còn lại được lưu vào vùng nhớ đệm để gán cho biến kiểu chuỗi tiếp sau khi gặp lệnh scanf dịnh dạng chuỗi lần kế tiếp.

Thông thường, để nhập một chuỗi ký tự từ bàn phím, ta sử dụng hàm gets().

Cú pháp: gets(<Biến chuỗi>)

Ví dụ 4.1:

```
char str[100];
gets(str);
```

Để xuất một chuỗi (biểu thức chuỗi) lên màn hình, ta sử dụng hàm puts().

Cú pháp: puts(<Biểu thức chuỗi>)

Ví dụ 4.2:

```
puts(str);
```

Một chương trình thực thi sử dụng nhiều biến lưu trữ dữ liệu. Trong khi đó, vùng nhớ chương trình thực thi thì hạn chế, do đó, người dùng thường lưu trữ dữ liệu trên file text để hỗ trợ cho chương trình thực thi tốt.

Cho f là input file dạng text thì dòng lệnh **f >> s** đọc dữ liệu vào đối tượng s đến khi *gặp dấu cách*.

Ví dụ 4.3:

Trong file input.txt chứa thông tin sau:

35 4 5 11

Đoạn lệnh sau đây sẽ gọi 4 lần dòng lệnh f>>s để thực hiện chức năng đọc thông tin trong file input.txt và in nội dung đọc được trong file ra màn hình.

```
ifstream f("input.txt");
char s[100];
for(int i=0; i<4; i++)
{
      f>>s;
      cout<<"\t"<<s;
}
```

Muốn đọc đầy đủ một dòng dữ liệu chứa cả dấu cách từ input file f vào một biến mảng kí tự s ta có thể dùng phương thức getline như ví dụ sau đây

```
char s[1001];
f.getline(s,1000,'\n');
```

Phương thức này đọc một dòng tối đa 1000 kí tự vào biến s, và thay dấu kết dòng '\n' trong input file bằng dấu kết xâu '/0' trong C.

Ta có thể sử dụng phương thức g<<s cho phép ghi chuỗi s vào file dạng text, trong đó g là output file dạng text.

```
ofstream g("output.txt");
g<< " Hello!";
```

Sau khi kết thúc thao tác, ta dùng phương thức đóng file

```
f.close();
g.close();
```

4.1.3 Xâu con

Cho 1 xâu kí tự s. Nếu xóa khỏi s một số kí tự và dồn các kí tự còn lại cho kề nhau, ta sẽ thu được một *xâu con* của xâu kí tự s.

Ví dụ 4.4:

s = "Ky thuat lap trinh";

S1 = "Ky lap trinh", S2 = "Ky thuat", S3 = "thuat" là xâu con của xâu kí tự s.

Cho 2 xâu kí tự s1, s2. Một xâu kí tự s vừa là xâu con của s1, và vừa là xâu con của s2 thì s được gọi là xâu con chung của 2 chuỗi s1 và s2.

Xét x = "xaxxbxcxd", y = "ayybycdy", xâu con chung của x, y là "abcd", "abd", "acd",… và chiều dài của xâu con chung dài nhất là 4.

Thuật toán xác định chiều dài xâu con chung dài nhất.

Xét hàm 2 biến s(i,j) là đáp số khi giải bài toán với 2 tiền tố i:x và j:y. Ta có:

- s(0,0) = s(i,0) = s(0,j) = 0: một trong hai xâu là rỗng thì xâu con chung là rỗng nên chiều dài là 0;
- Nếu x[i] = y[j] thì s(i,j) = s(i–1,j–1) + 1;
- Nếu x[i] ≠ y[j] thì s(i,j) = Max { s(i–1,j), s(i,j–1) }.

Để cài đặt, trước hết ta có thể sử dụng mảng hai chiều v với qui ước v[i][j] = s(i,j). Sau đó ta cải tiến bằng cách sứ dụng 2 mảng một chiều a và b, trong đó a là mảng đã tính ở bước thứ i–1, b là mảng tính ở bước thứ i, tức là ta qui ước a = v[i–1] (dòng i–1 của ma trận v), b = v[i] (dòng i của ma trận v). Ta có, tại bước i, ta xét kí tự x[i], với mỗi j = 0..len(y)–1,

- Nếu x[i] = y[j] thì b[j] = a[j–1] + 1;
- Nếu x[i] ≠ y[j] thì b[j] = Max { a[j], b[j–1] }.

Sau khi đọc dữ liệu vào hai xâu x và y ta gọi hàm XauChung để xác định chiều dài tối đa của xâu con chung của x và y. a,b là các mảng nguyên 1 chiều.

4.2 Các thuật toán tìm kiếm chuỗi

Các thuật toán này đều có cùng ý nghĩa là kiểm tra một chuỗi P có nằm trong một văn bản T hay không, nếu có thì nằm ở vị trí nào, và xuất hiện bao nhiêu lần. Ví dụ, kiểm tra chuỗi "lập trình" có nằm trong nội dung của file văn bản KTLT.txt hay không, xuất hiện tại vị trí nào và xuất hiện bao nhiêu lần.

4.2.1 Thuật toán Brute Force

Thuật toán Brute Force thử kiểm tra tất cả các vị trí trên văn bản từ 0 cho đến kí tự cuối cùng trong văn bản. Sau mỗi lần thử thuật toán Brute Force dịch mẫu sang phải một ký tự cho đến khi kiểm tra hết văn bản.

Thuật toán Brute Force không cần giai đoạn tiền xử lý cũng như các mảng phụ cho quá trình tìm kiếm. Độ phức tạp tính toán của thuật toán này là O(N*M).

Để mô phỏng quá trình tìm kiếm, ta thu nhỏ văn bản T thành chuỗi T. Ý tưởng của thuật toán này là so sánh từng kí tự trong chuỗi P với từng kí tự trong đoạn con của T. Bắt đầu từng kí tự trong P so sánh cho đến hết chuỗi P, trong quá trình so sánh, nếu thấy có sự sai khác giữa P và 1 đoạn con của T thì bắt đầu lại từ kí tự đầu tiên của P, và xét kí tự tiếp sau kí tự của lần so sánh trước trong T.

Thuật toán

```
//Nhập chuỗi chính T
Nhập chuỗi cần xét P
if ((  len (P) = 0) or ( len (T) = 0 ) or (  len (P) > len (T) )
 // len : chiều dài chuỗi
    P không xuất hiện trong T
else
    Tính vị trí dừng STOP =  len(T) – len(P)
    Vị trí bắt đầu so sánh START = 0
// ta cần vị trí dừng vì ra khỏi STOP
//chiều dài P lớn hơn chiều dài đoạn T còn lại thì dĩ nhiên P không thuộc T
```

So sánh các ký tự giữa P và T bắt đầu từ START

IF (các ký tự đều giống nhau)

P có xuất hiện trong T

ELSE

Tăng START lên 1

Dừng khi P xuất hiện trong T hoặc START > STOP

Ví dụ 4.5:

T = "I LIKE COMPUTER" n = len(T) = 14

P = "LIKE" m = len(P) = 4

start = 0

stop = n-m = 10

Bước 1: I LIKE COMPUTER i = start = 0

LIKE j = 0

P[j] = 'L' != T[i+j] = 'I'

----> tăng i lên 1 (ký tự tiếp theo trong T)

j = 0;(trở về đầu chuỗi P so sánh lại từ đầu P hay P dịch sang phải 1 ký tự.

Bước 2: I LIKE COMPUTER i = 1

LIKE j = 0

p[j] = 'L' != T[i] = ' '

---> tăng i lên một, j = 0

Bước 3: I LIKE COMPUTER i = 2

LIKE j = 0

p[j] == T[i+j] ='L'

----> tăng j lên một,

không tăng i vì ta đang xét T[i+j].

(chỉ khi nào có sự sai khác mới tăng i lên một)

Bước 4: I LIKE COMPUTER i = 2

LIKE j = 1

p[j] == T[i+j] = 'I'

-----> tăng j lên một

Bước 5: I LIKE COMPUTER i = 2

LIKE j = 2

p[j] == T[i+j] = 'K'

-----> tăng j lên một

Bước 6: I LIKE COMPUTER i = 2

LIKE j = 3

p[j] == T[i+j] = 'I'

-----> tăng j lên một -----> j = 4 = m : hết chuỗi P ------> P có xuất hiện trong T

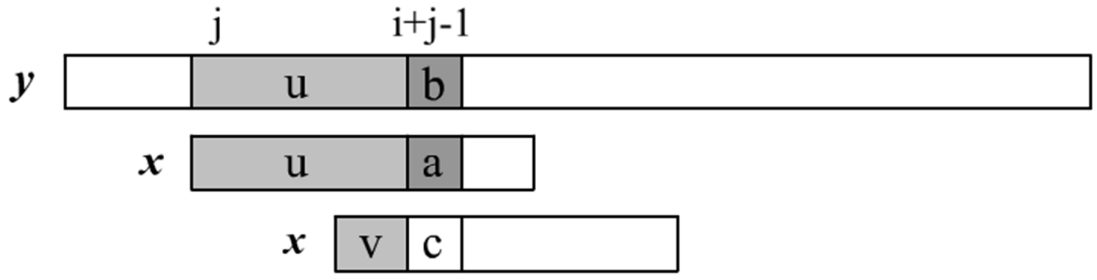
Kết quả: Xuất ra vị trí i = 2 là vị trí xuất hiện đầu tiên của P trong T

4.2.2 Thuật tóan Knuth – Morris – Pratt

Thuật toán Knuth-Morris-Pratt là thuật toán có độ phức tạp tuyến tính đầu tiên được phát hiện ra, nó dựa trên thuật toán brute force với ý tưởng lợi dụng lại những thông tin của lần thử trước cho lần sau. Trong thuật toán brute force vì chỉ dịch cửa sổ đi một ký tự nên có đến m-1 ký tự của cửa sổ mới là những ký tự của cửa sổ vừa xét. Trong đó có thể có rất nhiều ký tự đã được so sánh giống với mẫu và bây giờ lại nằm trên cửa sổ mới nhưng được dịch đi về vị trí so sánh với mẫu. Việc xử lý những ký tự này có thể được tính toán trước rồi lưu lại kết quả. Nhờ đó lần thử sau có thể dịch đi được nhiều hơn một ký tự, và giảm số ký tự phải so sánh lại.

Xét lần thử tại vị trí j, khi đó cửa sổ đang xét bao gồm các ký tự *y[j…j+m-1]* giả sử sự khác biệt đầu tiên xảy ra giữa hai ký tự *x[i]* và *y[j+i-1]*.

Khi đó *x[1…i]=y[j…i+j-1]=u* và *a=x[i]¹y[i+j]=b*. Với trường hợp này, dịch cửa sổ phải thỏa mãn *v* là phần đầu của xâu x khớp với phần đuôi của xâu u trên văn bản. Hơn nữa ký tự *c* ở ngay sau *v* trên mẫu phải khác với ký tự *a*. Trong những đoạn như *v* thoả mãn các tính chất trên ta chỉ quan tâm đến đoạn có độ dài lớn nhất.



Dịch cửa sổ sao cho *v* phải khớp với *u* và *c ≠ a*

Dịch cửa sổ sao cho *v* phải khớp với *u* và *c ¹ a*

Thuật toán Knuth-Morris-Pratt sử dụng mảng Next[i] để lưu trữ độ dài lớn nhất của xâu v trong trường hợp xâu *u=x[1…i-1]*. Mảng này có thể tính trước với chi phí về thời gian là ***O(m)*** (việc tính mảng Next thực chất là một bài toán qui hoạch động một chiều).

Thuật toán Knuth-Morris-Pratt có chi phí về thời gian là ***O(m+n)*** với nhiều nhất là 2n-1 lần số lần so sánh ký tự trong quá trình tìm kiếm.

Ví dụ

Để minh họa chi tiết thuật toán, chúng ta sẽ tìm hiểu từng quá trình thực hiện của thuật toán. Ở mỗi thời điểm, thuật toán luôn được xác định bằng hai biến kiểu nguyên, m và i, được định nghĩa lần lượt là vị trí tương ứng trên S bắt đầu cho một phép so sánh với W, và *chỉ số* trên W xác định kí tự đang được so sánh. Khi bắt đầu, thuật toán được xác định như sau:

m: 0

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: 0

Chúng ta tiến hành so sánh các kí tự của W tương ứng với các kí tự của S, di chuyển lần lượt sang các chữ cái tiếp theo nếu chúng giống nhau. S[0] và W[0] đều là ‘A’. Ta tăng i :

m: 0

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: _1

S[1] và W[1] đều là ‘B’. Ta tiếp tục tăng i :

m: 0

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: __2

S[2] và W[2] đều là ‘C’. Ta tăng i lên 3 :

m: 0

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: ___3

Nhưng, trong bước thứ tư, ta thấy S[3] là một khoảng trống trong khi W[3] = 'D', không phù hợp. Thay vì tiếp tục so sánh lại ở vị trí S[1], ta nhận thấy rằng không có kí tự 'A' xuất hiện trong khoảng từ vị trí 0 đến vị trí 3 trên xâu S ngoài trừ vị trí 0; do đó, nhờ vào quá trình so sánh các kí tự trước đó, chúng ta thấy rằng không có khả năng tìm thấy xâu dù có so sánh lại. Vì vậy, chúng ta di chuyển đến kí tự tiếp theo, gán m = 4 và i = 0.

m: ____4

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: 0

Tiếp tục quá trình so sánh như trên, ta xác định được xâu chung "ABCDAB", với W[6] (S[10]), ta lại thấy không phù hợp. Nhưng từ kết quả của quá trình so sánh trước, ta đã duyệt qua "AB", có khả năng sẽ là khởi đầu cho một đoạn xâu khớp, vì vậy ta bắt đầu so sanh từ vị trí này. Như chúng ta đã thấy các kí tự này đã trùng khớp với hau kí tự trong phép so khớp trước, chúng ta không cần kiểm tra lại chúng một lần nữa; ta bắt đầu với m = 8, i = 2 và tiếp tục quá trình so khớp.

m: _____8

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: __2

Quá trình so khớp ngay lập tức thất bại, nhưng trong W không xuất hiện kí tự ' ',vì vậy, ta tăng m lên 11, và gán i = 0.

m: _____11

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: 0

Một lần nữa, hai xâu trùng khớp đoạn kí tự "ABCDAB" nhưng ở kí tự tiếp theo, 'C', không trùng với 'D' trong W. Giống như trước, ta gán m = 15, và gán i = 2, và tiếp tục so sánh.

m: _____15

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: __2

Lần này, chúng ta đã tìm được khớp tương ứng với vị trí bắt đầu là S[15].

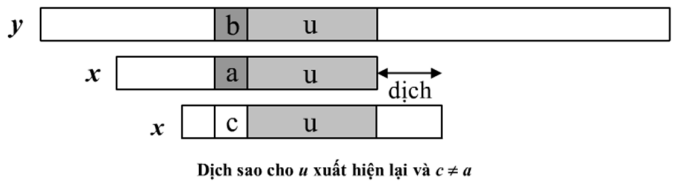
4.2.3 Thuật tóan Boyer Moore

Thuật toán Boyer Moore là thuật toán có tìm kiếm chuỗi rất có hiệu quả trong thực tiễn, các dạng khác nhau của thuật toán này thường được cài đặt trong các chương trình soạn thảo văn bản.

Khác với thuật toán Knuth-Morris-Pratt, thuật toán Boyer-Moore kiểm tra các ký tự của mẫu từ phải sang trái và khi phát hiện sự khác nhau đầu tiên thuật toán sẽ tiến hành dịch cửa sổ đi. Trong thuật toán này có hai cách dịch của sổ:

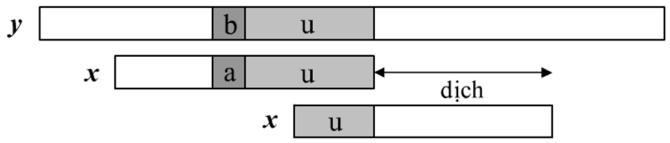
Cách thứ 1: gần giống như cách dịch trong thuật toán KMP, dịch sao cho những phần đã so sánh trong lần trước khớp với những phần giống nó trong lần sau.

Trong lần thử tại vị trí *j*, khi so sánh đến ký tự *i* trên mẫu thì phát hiện ra sự khác nhau, lúc đó $x[i+1\ldots m]=y[i+j\ldots j+m-1]=u$ và $a=x[i]^1y[i+j-1]=b$ khi đó thuật toán sẽ dịch cửa sổ sao cho đoạn $u=y[i+j\ldots j+m-1]$ giống với một đoạn mới trên mẫu (trong các phép dịch ta chọn phép dịch nhỏ nhất)



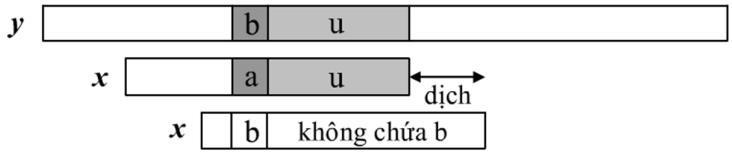
Dịch sao cho *u* xuất hiện lại và $c \neq a$

Nếu không có một đoạn nguyên vẹn của u xuất hiện lại trong x, ta sẽ chọn sao cho phần đôi dài nhất của u xuất hiện trở lại ở đầu mẫu.



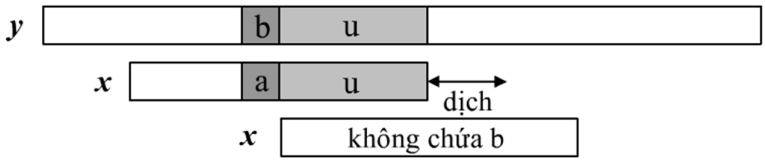
Dịch để một phần đôi của u xuất hiện lại trên x

Cách thứ 2: Coi ký tự đầu tiên không khớp trên văn bản là *b=y[i+j-1]* ta sẽ dịch sao cho có một ký tự giống *b* trên xâu mẫu khớp vào vị trí đó (nếu có nhiều vị trí xuất hiện *b* trên xâu mẫu ta chọn vị trí phải nhất)



Dịch để ký tự b ăn khớp với văn bản.

Nếu không có ký tự b nào xuất hiện trên mẫu ta sẽ dịch cửa sổ sao cho ký tự trái nhất của cửa sổ vào vị trí ngay sau ký tự *y[i+j-1]=b* để đảm bảo sự ăn khớp



Dịch khi b không xuất hiện trong *x*

Trong hai cách dịch thuật toán sẽ chọn cách dịch có lợi nhất.

Trong cài đặt ta dùng mảng bmGs để lưu cách dịch 1, mảng bmBc để lưu phép dịch thứ 2(ký tự không khớp). Việc tính toán mảng bmBc thực sự không có gì nhiều để bàn. Nhưng việc tính trước mảng bmGs khá phức tạp, ta không tính trực tiếp mảng này mà tính gián tiếp thông qua mảng suff. Có suff[i]=max{k | x[i-k+1…i]=x[m-k+1…m]}

Các mảng bmGs và bmBc có thể được tính toán trước trong thời gian tỉ lệ với ***O****(m+d)*. Thời gian tìm kiếm (độ phức tạp tính toán) của thuật toán Boyer-Moore là ***O****(m*n)*. Tuy nhiên với những bản chữ cái lớn thuật toán thực hiện rất nhanh. Trong trường hợp tốt chi phí thuật toán có thể xuống đến ***O****(n/m)* là chi phí thấp nhất của các thuật toán tìm kiếm hiện đại có thể đạt được.

BÀI TẬP CHƯƠNG 4

Bài 1. Cho file input.txt chứa nội dung sau:

Môn học: Kỹ thuật lập trình nâng cao

Số tiết: 30

Viết chương trình đọc nội dung trong file input.txt ra màn hình theo đúng định dạng trong file.

Bài 2. Cho file input.txt chứa nội dung sau:

6

35 2 6 4 12 9

Viết chương trình đọc nội dung trong file input.txt lưu vào mảng 1 chiều, sau đó in ra màn hình tổng các phần tử trong mảng.

Bài 3. Cho file input.txt chứa nội dung sau:

Số phần tử là: 5

Giá trị mảng la: 4 2 7 13 9

Viết chương trình đọc nội dung trong file input lưu vào biến mảng 1 chiều, sắp xếp mảng 1 chiều tăng dần, sau đó lưu kết quả mảng đã sắp xếp vào file output.txt.

Bài 4. Cho file input chứa nội dung sau:

Số dòng: 4

Số cột: 5

3 6 8 0 -5

5 12 7 21 -45

2 6 19 3 4

1 6 -7 3 9

Viết chương trình đọc nội dung trong file input lưu vào biến mảng 2 chiều, tìm dòng có tổng lớn nhất, sau đó lưu kết quả vào file output.txt.

Bài 5. Cho file input.txt chứa nội dung sau:

Số phần tử: 4

3 6 2 9

4 7 9 1

5 2 6 8

4 2 7 3

Viết chương trình đọc nội dung trong file input, đếm số phần tử chẵn trên biên của ma trận, sau đó in kết quả ra màn hình.

Bài 6. Cho file input.txt chứa nội dung sau:

Số phần tử: 3

Ma trận A

3	6	3
4	3	1
2	5	2

Ma trận B

1	4	9
8	3	7
2	5	6

Viết chương trình đọc nội dung trong file input, tính ma trận tổng S = A+B, ma trận tích P = AxB, sau đó lưu 2 ma trận kết quả vào file output.txt

Bài 7. Viết chương trình cài đặt thuật toán tìm chiều dài xâu con chung dài nhất trong 2 chuỗi ký tự cho trước.

Bài 8. Viết chương trình cài đặt thuật toán tìm kiếm chuỗi Brute Force.

Bài 9. Viết chương trình cài đặt thuật toán tìm kiếm chuỗi Knuth – Morris – Pratt

Bài 10. Viết chương trình cài đặt thuật toán tìm kiếm chuỗi Boyer – Moore