CHƯƠNG V: QUI HOẠCH ĐỘNG

1. Chiến lược qui hoạch động

Qui hoạch động (DP – Dynamic Programming), một thuật ngữ được nhà toán học Rechard Bellman đưa ra vào năm 1957, là một phương pháp giải bài toán bằng cách kết hợp các lời giải cho các bài toán con của nó giống như phương pháp chia để trị (devide-and-conquer). Các thuật toán chia để trị phân hoạch bài toán cần giải quyết thành các bài toán con độc lập với nhau, sau đó giải quyết chúng bằng phương pháp đệ qui (recursive) và kết hợp các lời giải lại để nhận được lời giải cho bài toán ban đầu. Ngược lại qui hoạch động là phương pháp được áp dụng khi mà các bài toán con của bài toán ban đầu (bài toán gốc) là không độc lập với nhau, chúng có chung các bài toán con nhỏ hơn. Trong các trường hợp như vậy một thuật toán chia để trị sẽ thực hiện nhiều việc hơn những gì thực sự cần thiết, nó sẽ lặp lại việc giải quyết các bài toán con nhỏ hơn đó. Một thuật toán qui hoạch động sẽ chỉ giải quyết tất cả các bài toán con nhỏ một lần duy nhất sau đó lưu kết quả vào một bảng và điều này giúp nó tránh không phải tính toán lại các kết quả mỗi khi gặp một bài toán con nhỏ nào đó.

Qui hoạch động thường được áp dụng với các bài toán tối ưu. Trong các bài toán tối ưu đó thường có nhiều nghiệm (lời giải). Mỗi lời giải có một giá trị được lượng giá bằng cách sử dụng một hàm đánh giá tùy thuộc vào các bài toán cụ thể và yêu cầu của bài toán là tìm ra một nghiệm có giá trị của hàm đánh giá là tối ưu (lớn nhất hoặc nhỏ nhất).

Qui hoạch động là một phương pháp chung rất hiệu quả để giải quyết các vấn đề tối ưu chẳng hạn như trên các đối tượng sắp thứ tự từ trái qua phải, vấn đề tìm đường đi ngắn nhất, vấn đề điều khiển tối ưu … Khi đã hiểu rõ về qui hoạch động việc ứng dụng vào giải các bài toán tối ưu không phải là quá khó khăn nhưng rất nhiều lập trình viên ban đầu phải mất rất nhiều thời gian mới có thể hiểu được. Bài báo này sẽ trình bày các bước cơ bản để áp dụng phương pháp qui hoạch động giải một số các bài toán tối ưu hay được ra trong các kỳ thi Olympic, học sinh giỏi Tin học cấp trường, cấp quốc gia … thông qua từng ví dụ cụ thể.

2. Bài toán 1: Dãy Fibonaci

Bài toán tìm số Fibonaci thứ n là một trong các bài toán quen thuộc đối với những người đã học lập trình, bài toán phát biểu như sau:

Dãy số Fibonaci được cho bởi công thức đệ qui và các điều kiện ban đầu như sau:

$$\begin{cases} F_n = F_{n-1} + F_{n-2}, \forall n \geq 2 \\ F_0 = 0 \\ F_1 = 1 \end{cases}$$

Xây dựng thuật toán tính F_n với n là một số nguyên nhập từ bàn phím.

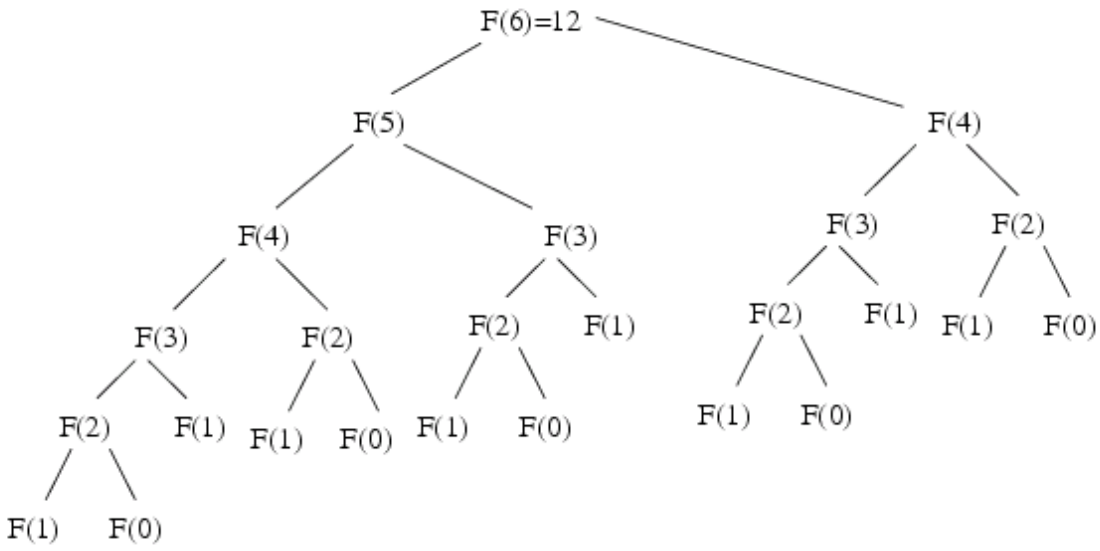
Cách 1: Sử dụng phương pháp đệ qui

Với định nghĩa dãy Fibonaci trên chúng ta dễ dàng thấy thuật toán đơn giản nhất để giải quyết bài toán là sử dụng một hàm đệ qui để tính F_n, chẳng hạn như sau:

```
int fibonaci(int k)
```

```
{
  if(k>=2)
        return (fibonaci(k-1)+fibonaci(k-2));
  return 1;
}
```

Thuật toán trên là đúng và hoàn toàn có thể cài đặt với một ngôn ngữ lập trình bất kỳ có hỗ trợ việc sử dụng các hàm đệ qui. Tuy vậy đây là một thuật toán không hiệu quả, giả sử chúng ta cần tính F_6:



Ta có $F_{n+1}/F_n \approx (1+\sqrt{5})/2 \approx 1.61803$ suy ra $F^n \approx 1.61803^n$ và do cây nhị phân tính F_n chỉ có hai nút là F_0 và F_1 nên chúng ta sẽ có xấp xỉ 1.61803^n lần gọi tới hàm fibonaci (trên thực tế n=6 là 13 lần) hay có thể nói độ phức tạp của thuật toán là hàm mũ.

Cách 2: Sử dụng phương pháp qui hoạch động

Chúng ta hoàn toàn có thể sử dụng một thuật toán có độ phức tạp tuyến tính để tính F_n bằng cách sử dụng một mảng để lưu các giá trị dùng để tính F_n:

$F_0 = 0$

$F_1 = 1$

For i = 2 to n

$F_i = F_{i-1} + F_{i-2}$

Tuy nhiên giảm được thời gian tính toán thì lại mất thêm bộ nhớ để chứa các kết quả trung gian trong quá trình tính toán.

Qua ví dụ 1 chúng ta có một số nhận xét sau:

+ Qui hoạch động là một kỹ thuật tính toán đệ qui hiệu quả bằng cách lưu trữ các kết quả cục bộ.

+ Trong qui hoạch động kết quả của các bài toán con thường được lưu vào một mảng.

3. Bài toán 2: Bài toán nhân dãy các ma trận

Giả sử chúng ta cần nhân một dãy các ma trận: $A \times B \times C \times D \times \ldots$. Hãy xây dựng thuật toán sao cho số phép nhân cần sử dụng là ít nhất.

Chúng ta biết rằng để nhân một ma trận kích thước $X \times Y$ với một ma trận kích thước $Y \times Z$ (sử dụng thuật toán nhân ma trận bình thường) sẽ cần $X \times Y \times Z$ phép nhân.

$$\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix} = \begin{bmatrix} 13 & 18 & 23 \\ 18 & 25 & 32 \\ 23 & 32 & 41 \end{bmatrix}$$

Để giảm số phép nhân cần dùng chúng ta sẽ tránh tạo ra các ma trận trung gian có kích thước lớn và do phép nhân ma trận có tính kết hợp nên có thể đạt được điều này bằng cách sử dụng các dấu đóng mở ngoặc để chỉ ra thứ tự thực phép nhân giữa các ma trận. Bên cạnh đó phép nhân ma trận không phải là đối xứng nên không thể hoán vị thứ tự của chúng mà không làm thay đổi kết quả.

Giả sử chúng ta có 4 ma trận A, B, C, D với các kích thước tương ứng là 30×1, 1×40, 40×10, 10×25. Số giải pháp sử dụng các dấu ngoặc có thể là 3:

((AB)C)D = $30 \times 1 \times 40 + 30 \times 40 \times 10 + 30 \times 10 \times 25 = 20,700$

(AB)(CD) = $30 \times 1 \times 10 + 40 \times 10 \times 25 + 30 \times 40 \times 25 = 41,200$

A((BC)D) = $1 \times 40 \times 10 + 1 \times 10 \times 25 + 30 \times 1 \times 25 = 1400$

Các kết quả trên cho thấy thứ tự thực hiện các ma trận tạo ra một sự sai khác rất lớn về hiệu quả tính toán (số phép nhân cần dùng để tính ra kết quả cuối cùng sai khác nhau rất lớn). Vậy làm thế nào để tìm ra kết quả tối ưu nhất?

Gọi M(i,j) là số lượng phép nhân nhỏ nhất cần thiết để tính $\prod_{k=i}^{j} A_k$ ta có nhận xét sau:

+ Các dấu ngoặc ngoài cùng phân hoạch dãy các ma trận (i,j) tại một vị trí k nào đó.

+ Cả hai nửa của dãy các ma trận (i,j) tại điểm phân hoạch k sẽ đều có thứ tự các dấu ngoặc là tối ưu.

Từ đó ta rút ra công thức sau truy hồi sau:

+ M(i,j) = $Min_{i \le k \le j-1} \left[M(i,k) + M(k+1,j) + d_{i-1} d_k d_j \right]$

+ M(i,i) = 0

Trong đó ma trận A_i có kích thước là (d_{i-1}, d_i).

Cũng giống như trong trường hợp của bài toán Fibonaci nếu chúng ta sử dụng một cài đặt đệ qui thì độ phức tạp của thuật toán sẽ là hàm mũ vì sẽ có rất nhiều lời gọi hàm giống nhau được thực hiện.

Nếu có n ma trận suy ra sẽ có n+1 số nguyên là kích thước của chúng và sẽ có tổ hợp chập 2 của n phần tử các xâu con (mỗi xâu tương ứng với mỗi thứ tự của các dấu ngoặc) giữa 1 và n nên chỉ cần dùng $\boldsymbol{O}(n^2)$ không gian nhớ để lưu kết quả cả các bài toán con.

Lại do k nằm giữa i,j theo phân tích trên nên chúng ta có thể lưu trữ các kết quả trung gian của bài toán trên một ma trận tam giác, đồng thời để chỉ ra nghiệm đạt được kết quả tối ưu cho bài toán chúng ta có thể lưu giá trị k vào một ma trận tam giác cùng kích thước khác.

Thuật toán tính M(1,n):

```
int matrixOrder
{
  for(i=1;i<=n;i++)
        M[i][j] = 0;
  for(b=1;b<n;b++)
        for(i=1;i<=n-b;i++)
        {
              j = i+b;
              M[i][j] = Min_{i≤k≤j-1}[ M(i,k) + M(k+1,j) + d_{i-1} d_k d_j ]
              faster[i][j] = k;
        }
  return M[1][n];
}
```

Thuật toán in nghiệm:

```
void showOrder(i,j)
{
  if(i==j)
        printf(A[i]);
  else
  {
        k = faster[i][j];
        printf("(");
```

```
            showOrder(i,k);
            printf("*");
            showOrder(k+1,j);
            printf(")");
    }
}
```

4. Phương pháp qui hoạch động

Qua hai ví dụ trên chúng ta có thể thấy quá trình phát triển của một thuật toán qui hoạch động có thể được chia làm 4 bước như sau:

1. *Xác định đặc điểm cấu trúc của giải pháp tối ưu của bài toán*
2. *Tìm công thức truy hồi (đệ qui) xác định giá trị của một giải pháp tối ưu*
3. *Tính giá trị tối ưu của bài toán dựa vào các giá trị tối ưu của các bài toán con của nó (bottom-up).*
4. *Xây dựng nghiệm đạt giá trị tối ưu từ các thông tin đã tính*.

Các bước 1-3 là các bước cơ bản trong việc giải bất cứ bài toán tối ưu nào bằng phương pháp qui hoạch động. Bước 4 có thể bỏ qua nếu như bài toán chỉ yêu cầu tìm ra giá trị tối ưu chứ không cần chỉ ra nghiệm cụ thể. Thông thường 2 bước đầu là quan trọng và cũng là khó khăn hơn cả, việc xác định cấu trúc nghiệm cũng như công thức truy hồi cần dựa vào kinh nghiệm và sự quan sát các trường hợp cụ thể của bài toán. Do vậy trong quá trình xây dựng thuật toán qui hoạch động cho các bài toán tối ưu chúng ta cần khảo sát các bộ giá trị thực tế của bài toán, giá trị tối ưu và nghiệm của bài toán ứng với các bộ giá trị đó.

Để có thể hiểu rõ hơn quá trình áp dụng các bước của phương pháp qui hoạch động giải bài toán tối ưu chúng ta xét ví dụ thứ 3 sau:

5. Bài toán dãy con chung dài nhất

Cho 2 dãy ký tự X[n] và Y[m] hãy xây dựng thuật toán tìm dãy con chung lớn nhất của hai dãy trên, với dãy con của một dãy được định nghĩa là một tập con các ký tự của dãy (giữ nguyên thứ tự).

Ví dụ với: X = A L G O R I T H M

Y = L O G A R I T H M

Thì dãy con chung dài nhất là Z = LORITHM

Bước 1: Xác định đặc điểm của dãy con chung dài nhất (giải pháp tối ưu của bài toán)

+ Giả sử chúng ta đã có lời giải là Z[1..k]

+ Nếu hai ký tự cuối cùng của X và Y trùng nhau thì đó cũng là ký tự cuối cùng của Z.

+ Phần còn lại của Z khi đó sẽ là xâu con chung dài nhất của X[1..n-1] và Y[1..m-1].

+ Nếu hai ký tự cuối của X và Y không trùng nhau thì một trong số chúng sẽ không nằm trong Z (có thể cả hai).

+ Giả sử ký tự không nằm trong Z trong trường hợp đó là ký tự của X

+ Thế thì Z sẽ là dãy con dài nhất của X[1..n-1] và Y[1..m].

+ Ngược lại nếu ký tự không nằm trong Z là ký tự của Y thì Z sẽ là dãy con dài nhất của X[1..n] và Y[1..m-1].

Bước 2: Xây dựng công thức truy hồi tính độ dài lớn nhất của dãy con của 2 dãy

Từ bước 1 ta có thể tiến hành xây dựng công thức truy hồi như sau:

+ Gọi C[i][j] là độ dài dãy con lớn nhất của hai dãy X[1..i] và Y[1..j]

+ C[i][0] = C[0][j] với mọi i, j.

+ Lời giải của bài toán chính là C[n][m].

+ Công thức truy hồi $C[i][j] = \begin{cases} C[i-1][j-1]+1 (1) \\ \max(C[i-1][j], C[i][j-1]) (2) \end{cases}$

+ Trường hợp 1 là khi X[i] = Y[j], còn trường hợp (2) là khi X[i] khác Y[j]

		A	L	G	O	R	I	T	H	M
	0	0	0	0	0	0	0	0	0	0
L	0	0	1	1	1	1	1	1	1	1
O	0	0	1	1	2	2	2	2	2	2
G	0	0	1	2	2	2	2	2	2	2
A	0	1	1	2	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3	3
I	0	1	1	2	2	3	4	4	4	4
T	0	1	1	2	2	3	4	5	5	5
H	0	1	1	2	2	3	4	5	6	6
M	0	1	1	2	2	3	4	5	6	7

Bước 3: Xây dựng thuật toán tìm dãy con chung dài nhất của 2 dãy X[1..n] và Y[1..m].

Thứ tự tính toán diễn ra như sau: ban đầu chúng ta tính C[1][1], C[1][2], ..., C[1][n], C[2][1], C[2][2], ..., C[2][n], ... Độ phức tạp để tính C[i][j] bằng $\boldsymbol{O}(1)$ với i=1,n và j=1,m nên độ phức tạp của thuật toán là $\boldsymbol{O}(mn)$.

```
int longest_common_sequence(X, Y)
{
```

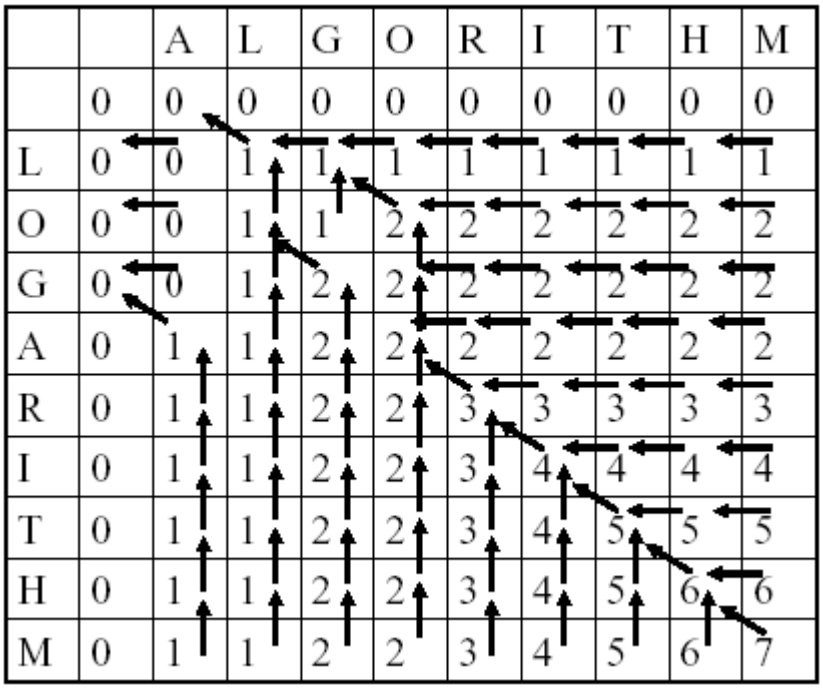
```
    for(i=0;i≤m;i++)
        C[i][0] = 0;
    for(j=0;j≤n;j++)
        C[0][j] = 0;
    for(i=1;i≤m;i++)
        for(j=1;j≤n;j++)
            if(X[i]==Y[j])
                C[i][j] = C[i-1][j-1] + 1;
            else
                C[i][j] = max(C[i-1][j],C[i][j-1]);
    return C[m][n];
}
```

		A	L	G	O	R	I	T	H	M
	0	0	0	0	0	0	0	0	0	0
L	0	0	1	1	1	1	1	1	1	1
O	0	0	1	1	2	2	2	2	2	2
G	0	0	1	2	2	2	2	2	2	2
A	0	1	1	2	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3	3
I	0	1	1	2	2	3	4	4	4	4
T	0	1	1	2	2	3	4	5	5	5
H	0	1	1	2	2	3	4	5	6	6
M	0	1	1	2	2	3	4	5	6	7

Bước 4: Tìm dãy con dài nhất của X[1..n] và Y[1..m](Xây dựng nghiệm đạt giá trị tối ưu từ các thông tin đã tính)

Để tìm lại được nghiệm chúng ta sử dụng một bảng D[i][j] trỏ ngược tới (i, j-1) hoặc (i-1, j) hoặc (i-1, j-1) và lần ngược từ D[m][n] như sau: nếu D[i][j] trỏ tới (i-1, j-1) thì X[i] = Y[j] là ký tự này sẽ nằm trong dãy con dài nhất của 2 xâu. Việc D[i][j] trỏ tới (i-1,j-1), (i-1,j) hoặc (i,j-1) phụ thuộc vào giá trị của mảng C tại vị trí nào được sử dụng để tính C[i][j]. Các giá trị của mảng D sẽ được tính như sau:

D[i][j] bằng 1 (trên trái) nếu C[i][j] = 1 + C[i-1][j-1], bằng 2 (trên) nếu C[i][j] = C[i-1][j] và bằng 3 (trái) nếu C[i][j] = C[i][j-1].



		A	L	G	O	R	I	T	H	M
	0	0	0	0	0	0	0	0	0	0
L	0	0	1	1	1	1	1	1	1	1
O	0	0	1	1	2	2	2	2	2	2
G	0	0	1	2	2	2	2	2	2	2
A	0	1	1	2	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3	3
I	0	1	1	2	2	3	4	4	4	4
T	0	1	1	2	2	3	4	5	5	5
H	0	1	1	2	2	3	4	5	6	6
M	0	1	1	2	2	3	4	5	6	7

Thuật toán tìm nghiệm: Nếu D[i][j] trỏ tới (i-1, j-1) (1 – trên trái) thì X[i] = Y[j] và ký tự này sẽ nằm trong dãy con là nghiệm.

```
char *  findSolution()
{
        row = m, col = n, lcs="";
        while((row>0)&&(col>0))
        {
          if(D[row][col]  == 1)
          {
                lcs = lcs + X[row];
                row = row – 1;
                col = col - 1;
          }else{
                    if (D[row][col]==2)
                      row = row – 1;
                else if (D[row][col]  = 3)
                      col = col – 1;
          }
```

```
        }
        reverse lcs; // đảo ngược xâu lcs
        return lcs;
}
```

		A	L	G	O	R	I	T	H	M
	0	0	0	0	0	0	0	0	0	0
L	0	0	1	1	1	1	1	1	1	1
O	0	0	1	1	2	2	2	2	2	2
G	0	0	1	2	2	2	2	2	2	2
A	0	1	1	2	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3	3
I	0	1	1	2	2	3	4	4	4	4
T	0	1	1	2	2	3	4	5	5	5
H	0	1	1	2	2	3	4	5	6	6
M	0	1	1	2	2	3	4	5	6	7

Qui hoạch động là một phương pháp rất hiệu quả để giải rất nhiều các vấn đề tối ưu hóa thuộc nhiều lĩnh vực khác nhau (Tin học, Kinh tế, Điều khiển, ...) tuy nhiên trong khuôn khổ của bài báo này tác giả chỉ trình bày việc ứng dụng qui hoạch động vào giải một số các bài toán tối ưu mà dạng của chúng thường được ra trong các kỳ thi học sinh giỏi hay Olympic Tin học thông qua 3 ví dụ cụ thể. Hy vọng bài báo giúp được ít nhiều cho độc giả trong việc bước nắm bắt và sử dụng phương pháp qui hoạch động để giải các bài toán tối ưu.

6. Bài tập

Bài 1: ContestSchedule

Vào năm 3006 hầu hết các đều được thực hiện liên quốc gia và hoàn toàn online. Tất cả các cuộc thi lập trình này đều được lên lịch một cách hoàn hảo. Thời gian của mỗi cuộc thi được xác định bởi hai số nguyên s và t tương ứng với thời gian bắt đầu và kết thúc (trước thời điểm t). Vì thế nếu một cuộc thi kết thúc vào thời điểm t=10 và bắt đầu vào thời điểm s=10 thì một lập trình viên có thể tham gia vào cả hai cuộc thi.

Là một lập trình viên có kinh nghiệm nên đối với bất cứ cuộc thi nào Tom cũng có thể dự đoán chính xác tỉ lệ thắng cuộc của mình. Cho trước một danh sách các cuộc thi, hãy giúp Tom tính xem nên tham gia và các cuộc thi nào để tổng tỉ lệ thắng của anh ta là lớn nhất có thể được.

Input

Dữ liệu chương trình được cho trong một file text theo định dạng sau: trên mỗi dòng là dữ liệu về một cuộc thi gồm 3 số nguyên tương ứng là s, t và p ($1 \le s, t \le 1000000$, $1 \le p \le 100$) là thời gian bắt đầu, kết thúc và tỉ lệ thắng của cuộc thi.

Output

Kết quả xử lý của chương trình ghi vào 1 file text với độ chính xác đến 5 chữ số sau dấu phẩy.

Ví dụ

Input	Output
1 10 100 **10 20 100** **20 30 100** **30 40 100**	**4.0**
10 20 20 **30 40 60** **15 35 90**	**0.9**
1 100 85 **99 102 100** **101 200 60**	**1.45**

Bài 2: JoinedString

Cho một dãy các từ thành lập từ các chữ cái tiếng Anh, hãy tìm xâu có độ dài nhỏ nhất chứa tất cả các từ trong dãy đã cho. Nếu có nhiều xâu như vậy, hãy đưa ra xâu đầu tiên theo thứ tự Alphabet.

Input

Dữ liệu của chương trình được cho trong một file text, mỗi từ được ghi trên một dòng (số từ nhỏ hơn 13), độ dài của các từ nhỏ hơn 51 và chỉ chứa các ký tự tiếng Anh viết hoa.

Output

Kết quả xử lý của chương trình ghi vào một file text.

Ví dụ

Input	Output
BAB **ABA**	**ABAB**
ABABA **AKAKA** **AKABAS** **ABAKA**	**ABABAKAKABAS**

Bài 3: JumpyNum

Một số Jumpy là một số nguyên dương và các chữ số liền kề của nó khác nhau ít nhất 2 đơn vị. Ví dụ:

Các số thường	28459	28549	1091919	97753
Các số Jumpy	290464	13131313	9753	5

Hãy viết chương trình xác định xem giữa hai số nguyên low, high có bao nhiêu số Jumpy.

Input

Dữ liệu của chương trình được cho trong file text với 2 số low, high (nhỏ hơn 2000000) được ghi trên 2 dòng khác nhau.

Output

Kết quả xử lý của chương trình ghi vào một file text.

Ví dụ

Input	Output
1 **10**	**9**
9 **23**	**9**
8000 **20934**	**3766**