2.3 Dynamic Programming (Thuật toán quy hoạch động)

Created by TUNG DUC NGUYEN tung2.nguyen, last modified on 2021/05/18

Các thuật toán đệ quy có ưu điểm dễ cài đặt, tuy nhiên do bản chất của đệ quy, các chương trình thường kéo theo những đòi hỏi lớn về không gian bộ nhớ và một khối lượng tính toán khổng lồ.

Quy hoạch động là một kỹ thuật nhằm đơn giản hóa việc tính toán các công thức truy hồi bằng cách lưu trữ toàn bộ hay một phần kết quả tính toán tại mỗi bước với mục đích sử dụng lại. Bản chất của quy hoạch động là thay thế mô hình tính toán từ trên xuống (top-down) bằng mô hình từ dưới lên (bottom-up).

Từ programming ở đây không liên quan đến việc lập trình cho máy tính, mà đó là thuật ngữ mà các nhà toán học dùng để chỉ ra các bước chung trong việc giải quyết một dạng bài toán hoặc một lớp các vấn đề. Không có một thuật toán tổng quát để giải tất cả các bài toàn quy hoạch động. Mục đích của phần này là cung cấp một cách tiếp cận trong việc giải quyết các bài toàn **tối ưu** mang bản chất **đệ quy**.

1. Công thức truy hồi

1.1 Ví dụ

Cho số tự nhiên n ≤ 100. Hãy cho biết bao nhiêu cách phân tích số n thành tổng của các số nguyên dương, các hoán vị của nhau chỉ tính là một cách.

Ví dụ: n = 5 thì có 7 cách:

1. 5 = 1 + 1 + 1 + 1 +1
2. 5 = 1 + 1 + 1 + 2
3. 5 = 1 + 1 + 3
4. 5 = 1 + 2 + 2
5. 5 = 1 + 4
6. 5 = 2 + 3
7. 5 = 5

(Với n = 0, thì ta coi có 1 cách phân tích thành tổn các số nguyên dương - tập rỗng).

Để giải bài toán này, chúng ta có thể dùng phương pháp liệt kê và đếm số cấu hình. Ta thử suy nghĩ xem có cách nào khác để giải quyết bài toàn này ngoài cách liệt kê không?

Nhận xét:

Nếu gọi F[m,v] là số cách phân tích số v thành tổng các số nguyên dương ≤ m, khi đó cách phân tích số v thành số nguyên dương ≤ m có hai loại:

Loại 1: Không chưa số m trong phép phân tích, khi đó số các phân tích loại này chính là số cách phân tích số v thành tổng các số nguyên dương <m. Tức là phân tích số v thành tổng các số nguyên dương ≤ m - 1 và nó chính là **F[m-1,v]**

Loại 2: Có chứa ít nhất một số m trong phép phân tích. Khi đó nếu trong cách phân tích chúng ta bỏ số m đó đi, thì ta sẽ được cách phân tích số v-m thành các số nguyên dương ≤ m. Có nghĩa là về mặt số lượng, số các cách phân tích loại này chính là **F[m, v-m]**.

Trong trường hợp m > v thì rõ ràng chỉ có loại 1. Còn m ≤ v thì ta có cả loại 1 và loại 2. Vì thế:

F[m,v] = F[m-1,v] nếu m > v

F[m,v] = F[m-1, v] + F[m, v-m] nếu m ≤ v.

Ta có công thức xây dựng F[m,v] từ F[m-1,v] và F[m,v-m]. Công thức này gọi là **công thức truy hồi** đưa việc tính F[m,v] về việc tính các F[m',v'] với dữ liệu nhỏ hơn. Tất nhiên cuối cùng ta quan tâm đến F[n,n], số cách phân tích n thành tổng các số nguyên dương ≤ n.

Với n = 5, ta có bảng F sẽ là:

F	0	1	2	3	4	5
0	1	0	0	0	0	0
1	1	1	1	1	1	1
2	1	1	2	2	3	3
3	1	1	2	3	4	5
4	1	1	2	3	5	6
5	1	1	2	3	5	7

v

m

Nhìn vào bảng F, ta thấy rằng F[m,v] được tính bằng tổng của:

Một phần tử của hàng trên F[m-1,v]

Và một phần tử cùng hàng, bên trái F[m, v-m].

Do đó, để tính được F[m,v] thì thứ tự hợp lý để tính các phần tử trong bảng F phải tuân theo thứ tự thừ trên xuống, và trên mỗi hàng thì từ trái qua phải.

Điều đó có nghĩa ban đầu ta phải tính hàng 0 của bảng. Theo quy ước của đề bài thì F[0,0] = 1 và F[0,v] = 0 với mọi v > 0.

Ta có giải thuật để giải quyết bài toán trên:

```
const int MAX = 100;
int f[MAX +1][MAX + 1];
int n,m,v;
int solve() {
    memset(f, 0, (MAX + 1)*(MAX+1));// khởi tạo bảng F toàn số 0
    f[0][0] = 1;                    // set F[0,0] = 1
    for (m = 1; i < n; i ++) {
        for (v = 0; v < n; v ++) {
            if (v < m) f[m][v] = f[m-1][v];
            else       f[m][v] = f[m-1][v] + f[m][v-m];
        }
    }
    return f[n][n];
}
```

1.2 Cải tiến

Ta thấy rằng không gian bộ nhớ đang cần là n^2. Điều này sẽ gây khó khăn nếu n lớn.

Tuy nhiên dựa theo công thức thì khi tính hàng thứ n thì ta chỉ cần giá trị ở hàng thứ n-1. Do đó ta có thể cải tiến bộ nhớ bằng cách dùng hai mảng một chiều.

Việc dùng hai mảng một chiều có thể làm tốc độ chậm đi (trong việc gán lại giá trị cho hàng thứ n-1 thành hàng n khi chuẩn bị tính toán hàng n+1) hoặc phức tạp trong cài đặt bằng cách đổi tham chiếu của hai mảng.

=> Ta có thể cải tiến bằng cách dùng 1 mảng một chiều và lợi dụng việc tham chiếu giá trị của máy tính (ta dùng chính nó để tính giá trị kế tiếp). Khi đó công thưc sẽ được đổi thành f[v] = f[v] + f[v-m];

```
const int MAX = 100;
int f[MAX +1];
int n,m,v;
int solve() {
    memset(f, 0, MAX + 1);// khởi tạo bảng F toàn số 0
    f[0] = 1;             // set F[0] = 1
    for (m = 1; i < n; i ++) {
        for (v = m; v < n; v ++) { //Ta cũng không cần tính lại giá trị của f[m,v] khi m < v;
            f[v] = f[v] + f[v-m];
        }
    }
    return f[n];
}
```

1.4 Cài đặt đệ quy

Xét thấy việc tính F[m,v] ta cần phải biết giá trị chính xác của F[m-1,v] và F[m, v-m]. Sau đó xác định thứ tự tính toán (phần nào tính trước, phần nào tính sau) là rất quan trọng và đương nhiên cũng phải tính toán, suy nghẫm.

Tuy nhiên ta có thể dùng đệ quy mà không quan tâm đến thứ tự tính toán:

```
int getF(int m, int v) {
    if (m == 0) {
        if (v == 0) return 1;
        else        return 0;
    }
    if (m > v) return getF(m-1,v);
    else       return getF(m-1,v) + getF(m, v-m);
}
```

Tuy nhiên phương pháp này khá chậm vì phải gọi nhiều lần hàm getF(m,v). Ta có thể cải tiến bằng cách kết hợp một mảng hai chiều F.

Ban đầu giá trị của mảng là "chưa biết" (gán cho một giá trị đặt biệt), hàm getF(m,v) khi được gọi sẽ tra cứu đến F[m,v] nếu F[m,v] là chưa biết thì hàm getF(m,v) sẽ gọi đệ quy để tính F[m,v] rồi dùng giá trị này là kết quả của hàm. Còn nếu F[m,v] đã biết thì hàm chỉ việc dùng giá trị của F[m,v] là kết quả của hàm.

```
const int MAX = 100;
```

```
int f[MAX +1][MAX + 1];
int n,m,v;

int getF(int m, int v) {
    if (f[m][v] != -1) return f[m][v];
    if (m == 0) {
        if (v == 0) f[m][v] = 1;
        else        f[m][v] = 0;
    } else {
        if (m > v) f[m][v] = getF(m-1,v);
        else       f[m][v] = getF(m-1,v) + getF(m, v-m);
    }
    return f[m][v];
}
```

Việc sử dụng phương pháp đệ quy để giải công thức truy hồi là một kỹ thuật nên lưu ý, vì khi gặp một công thức truy hồi phức tạp, khó xác định thứ tự tính toán thì việc sử dụng đệ quy hiệu quả hơn.

2. Phương pháp quy hoạch động

2.1 Bài toán quy hoạch

Bàit toán quy hoạch là bài toán tối ưu: gồm có một hàm f gọi là hàm mục tiêu hay hàm đánh giá. Các hàm g1,g2,...,gn cho giá trị logic lọi là các hàm ràng buộc. Yêu cầu của bài toán là tìm một cấu hình x thỏa mãn tất cả các ràng buộc g1,g2,...,gn: gi(x) = TRUE (với mọi 1 ≤ i ≤ n) và x là tốt nhất, nghĩa là không tồn tại cấu hình y nào thỏa mãn tất cả ràng buộc mà f(y) tốt hơn f(x).

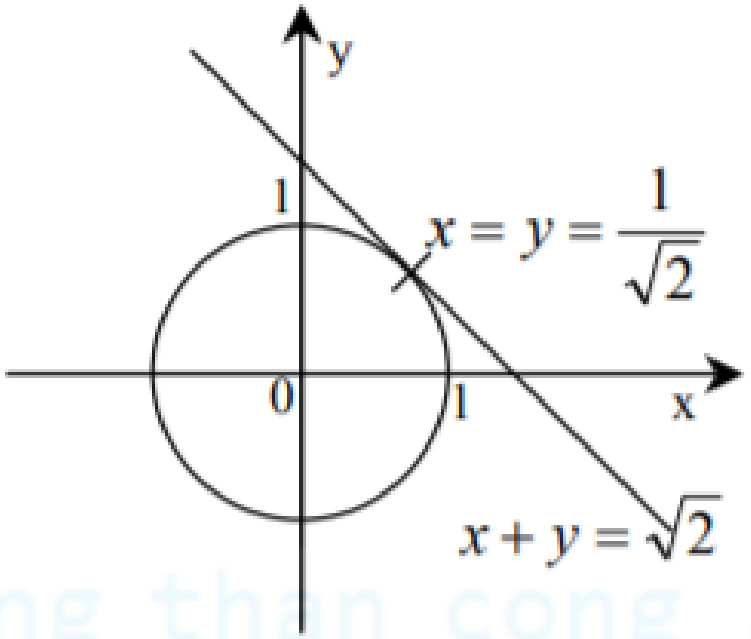
Ví dụ: Tìm tọa độ (x,y) trong mặt phẳng để:

hàm mục tiêu f: x + y → max

hàm ràng buộc: x^2 + y^2 ≤ 1

Xét trong mặt phẳng tọa độ, những cặp (x,y) thỏa mãn x^2 + y^2 ≤ 1 là những tọa độ nằm trong hình tròn có tâm O là gốc tọa độ, bán kính 1. Vậy nghiệm của bài toán chắn chắn nằm trong hình tròn.

Những đường thẳng có phương trình x + y = C (C là một hằng số) là đường thẳng vuông góc với phân giác góc phần tư thứ nhất. Ta phải tìm số C lớn nhất mà đường thẳng x + y = C vẫn có điểm chung với đường tròn tâm O bán kính 1. Đường thẳng đó là đường tiếp tuyến của đường tròn: x + y = sqrt(2).



Các bài toán quy hoạch rất phong phú và đa dạng, ứng dụng nhiều trong thực tế. Nhưng chúng ta cũng cần biết rằng, đa số các bài toán quy hoạch là không giải được hoặc chưa giải được.

2.2 Phương pháp quy hoạch động

Phương pháp quy hoạch động dùng để giải bài toán tối ưu có bản chất đệ quy, tức là tìm phương án tối ưu cho bài toán đó có thể đưa về phương án tối ưu của một số hữu hạn các bài toán con. Để giải quyết một bài toán lớn, ta chia nó thành nhiều bài toán con có dùng dạng với nó để có thể giải quyết độc lập. Khi không biết cần phải giải quyết những bài toán con nào, chúng ta sẽ đi **giải quyết tất cả các bài toán con** và l**ưu trữ lời giải của chúng** với mục đích **sử dụng lại** theo một sự phối hợp nào đó để giải quyết những bài toán tổng quát hơn.

Quy hoạch động bắt đầu từ việc **giải tất cả các bài toán nhỏ nhất** (bài toán cơ sở) để từ đó từng bước giải quyết những bài toán lớn hơn, cho tới khi giải được bài toán lớn nhất (bài toán ban đầu).

Trước khi áp dụng quy hoạch động, ta phải xem xét phương pháp đó có thỏa mãn những yêu cầu dưới đây hay không:

- Bài toán lớn phải phân rã được thành nhiều bài toán con, mà sự phối hợp lời giải của các bài toán con đó cho ta lời giải của bài toán lớn.
- Vì quy hoạch động phải đi giải tất cả các bài toán con, nên nếu không đủ không gian vật lý lưu trữ lời giải thì phương pháp quy hoạch động cũng không thể thực hiện được.
- Quá trình từ bài toán cơ sở đến tìm ra kết quả cho bài toán ban đầu phải hữu hạn bước

Các khái niệm:

Bài toán giải theo phương pháp quy hoạc động được gọi là **bài toán quy hoạch động**.

Công thức phối hợp nghiệm của các bài toán con để có nghiệm của bài toàn lớn gọi là **công thức truy hồi** của quy hoạch động.

Tập các bài toán nhỏ nhất có ngay lời giải để từ đó giải quyết các bài toán lớn hơn được gọi là **cơ sở quy hoạch động**.

Không gian lưu trữ các lời giải bài toán con để tìm cách phối hợp chúng được gọi là **bảng phương án** của quy hoạch động.

Các bước cài đặt:

- Giải tất các cả bài toán cơ sở (thông thường dất dễ), lưu các lời giải vào bảng phương án.
- Dùng công thức truy hồi phối hợp những lời giải của các bài toàn nhỏ lưu trong bảng phương án để tìm lời giải cho những bài toán lớn hơn và lưu chúng vào bảng phương án. Cho tới khi bài toán ban đầu tìm được lời giải.

- Dựa vào bảng phương án, truy vết tìm ra nghiệm tối ưu.

Cho đến nay, chưa có một định lý nào cho biết một cách chính xác những bài toàn nào có thể giải quyết hiệu quả bằng quy hoạch động. Tuy nhiên để biết một bài toán có thể giải bằng quy hoạch động hay không, ta có thể tự đặt câu hỏi:

- Một nghiệm tối ưu của bài toán lớn có phải là sự phối hợp các nghiệm tối ưu của các bài toàn con hay không?
- Liệu có thể nào lưu trữ được nghiệm của bài toán con dưới một hình thức nào đó để phối hợp tìm được nghiệm cho bài toán không?

Đôi khi, với một dạng bài có thể sử dụng quy hoạch động để giải quyết. Nhưng ta không thể xây dựng được công thức truy hồi hoặc không thể xây dựng được cách lưu trữ cho bảng phương án. Tin tốt là tất cả các lớp bài toán có thể giải quyết bằng quy hoạch động đều được công khai. Vậy cách tốt nhất là chúng ta nghiên cứu những lớp bài toán có thể giải quyết bằng quy hoạch động. Từ đó có thể dễ dàng đưa ra lời giải khi gặp những bài toán quy hoạch động.

3. Một số bài toán quy hoạch động

3.1 Dãy con đơn điệu tăng dài nhất

https://www.spoj.com/problems/LIS/

Cho dãy số nguyên A = a1, a2,... ,an (n ≤ 50000, -100000 ≤ ai ≤ 100000). Một dãy con của A là một cách chọn trong A một số phần tử giữ nguyên thứ tự.

Yêu cầu: hãy tìm dãy con đơn điệu tăng của A có độ dài lớn nhất.

Ví dụ: A = (1, -2, 2, 3, 4, 9, 10, 5, 6 ,7) thì dãy con đơn điệu tăng dài nhất là (1, 2, 3, 4, 5, 6 ,7).

Cách giải:

Bổ xung hai vào dãy A hai phần tử a0 = -∞ và a(n+1) = +∞. Khi đó dãy con đơn điệu tăng dài nhất chắc chắn sẽ **bắt đầu từ a0 và kết thúc ở a(n+1)** ← Kỹ thuật lính canh.

Với mọi i: 0 ≤ i ≤ n+1. Ta sẽ tính L[i] là độ dài dãy con đơn điệu dài nhất bắt đầu từ vị trí thứ i đến vị trí n+1.

- *Cơ sở quy hoạch động*:

  L[n+1] = 1. Dãy con này chỉ gồm một phần tử +∞.

- *Công thức truy hồi*:

  Giả sử ta đang cần tính L[i], độ dài dãy con tăng dài nhất bắt đầu a(i). Giả sử L[i] được tính khi đã biết L[i+1], L[i+2], ..., L[n+1].

  Như vậy dãy con đơn điệu tăng dài nhất bắt đầu từ a(i) sẽ được thành lập bằng cách lấy a(i) ghép vào đầu một trong số những dãy con đơn điệu tăng dài nhất bắt đầu tại vị trí a(j) đứng sau a(i). Ta sẽ chọn dãy nào?

  Tất nhiên là chỉ được ghép a(i) vào những dãy bắt đầu tại a(j) nào đó mà a(j) > a(i) để đảm bảo tính tăng dần và dĩ nhiên sẽ chọn dãy dài nhất để ghép vào trong những dãy thỏa mãn a(j) > a(i). Vậy L[i] sẽ được tính như sau:

  Xét tất cả chỉ số j từ i+1 đến n+1 mà a(j) > a(i), chọn ra chỉ số jmax mà có L[jmax] là lớn nhất. Đặt L[i] = L[jmax] +1.

- Truy vết:

  Do bài yêu cầu là tìm ra dãy con đơn điệu dài nhất, nên ta cần truy vết để đưa ra dãy này.

  Tại mỗi bước xây dựng L[i], mỗi ghi ta gán L[i] = L[jmax] +1, ta đặt T[i] = jmax. Để lưu lại rằng dãy con dài nhất bắt đầu tại a(i) sẽ có phần tử kế tiếp là a(jmax).

  Sau khi tính toán xong hay dãy L và T, ta bắt đầu từ 0:

  T[0] là phần tử đầu tiên được chọn.

  T[T[0]] là phần tử thứ hai được chọn.

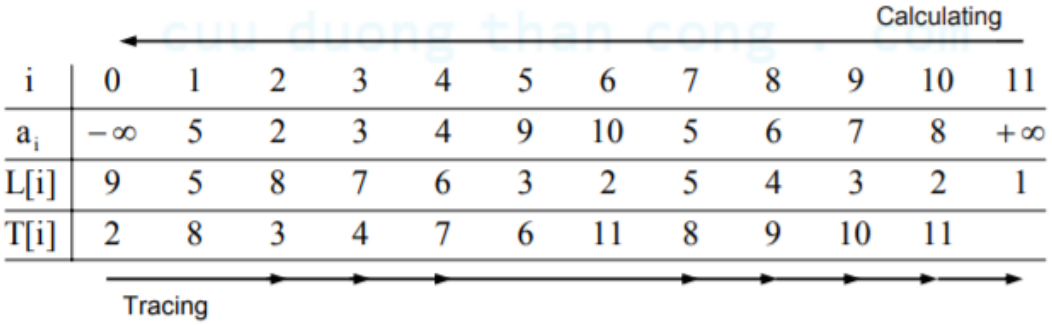
  T[T[T[0]]] là phần tử thứ ba được chọn.

  .....

  Quá trình truy vết có thể diễn ta như sau:

```
int i = T[0];
while (i != n + 1) {
    print(A[i]);
    i = T[i];
}
```

Ví dụ với dãy A = (5, 2, 3, 4, 9, 10, 5, 6, 7, 8). Hay dãy L và T sẽ được tính như sau:



i	0	1	2	3	4	5	6	7	8	9	10	11
a_i	$-\infty$	5	2	3	4	9	10	5	6	7	8	$+\infty$
L[i]	9	5	8	7	6	3	2	5	4	3	2	1
T[i]	2	8	3	4	7	6	11	8	9	10	11	

Thuật toán có thể triển khai như sau:

```
void resolve() {
    a[0] = INT_MIN; a[n+1] = INT_MAX; //Thêm hai phần tử canh đầu dãy
    L[n+1] = 1; //Điền cơ sở quy hoạch vào bảng phương án
    for (int i = n; i >=0; --i) { //Bắt đầu tính bảng phương án
        jmax = n +1;
        for (int j = i +1; j <= n+1; ++j) {
            if ((a[j] > a[i]) && (L[j] > L[jmax]))
                jmax = j;
        }
        L[i] = L[jmax] + 1; //Lưu độ dài dãy con tăng dài nhất bắt đầu tại a[i]
        T[i] = jmax;        // Lưu vết.
    }
}
```

Độ phức tạp tính toán của thuật toán trên là O(n^2).

Ta tìm cách cải thiện tốc độ bằng cách cải thiện cách tìm jmax bằng binary search.

Để ý thấy rằng độ dài của dãy là 50000. Ta có thể khởi tạo một mảng startOf[1...n+1].

Trong đó giá trị của startOf[k] là chỉ số x của phần tử a[x] thỏa mãn dãy đơn điệu dài nhất từ a[x] có độ dài là k. Nếu có nhiều x thỏa mãn điều kiện này thì ta sẽ chọn phần tử a[x] lớn nhất trong số các phần tử đó.

Do đó việc tìm jmax bằng cách sử dụng binary search trong mảng startOf.

Tất nhiên, mỗi khi xác định được L[i] thì ta cũng cập nhật lại mảng startOf.

3.2 Bài toán cái túi (knapsack)

https://www.spoj.com/problems/KNAPSACK/

Trong siêu thị có n (n ≤ 2000) gói hàng, gói hàng thứ i có trọng lượng là W(i) {1, 2, ..., i-1 2000 và giá trị V(i) ≤ 2000. Một tên trộm đột nhập vào siêu thị, tên trộm mang theo cái túi có thể mang được tối đa trọng lượng là M ≤ 2000. Hỏi tên trộm sẽ lấy đi những gói hàng nào để tổng giá trị là lớn nhất.

Cách giải:

Nếu gọi F[i,j] là giá trị lớn nhất có thể có bằng cachs chọn trong các gói {1, 2, ..., i} với giới hạn trọng lượng là j. Thì giá trị lớn nhất khi được chọn trong n gói với giới hạn trọng lượng M chính là F[n,M].

- *Công thức truy hồi:*

  Với giới hạn trọng lượng j, việc chọn tối ưu trong các gói {1, 2, ..., i-1, i} để có giá trị lớn nhất có hai khả năng:

  Nếu không chọn gói thứ i thì F[i,j] lớn nhất có thể chọn bằng cách chọn trong số các gói {1, 2, ..., i-1} với giới hạn trọng lượng là j. Tức là

  F[i,j] = F[i-1,j]

  Còn nếu chọn gói thứ i (trong điều kiện W(i) ≤ j) thì F[i,j] bằng giá trị gói thứ i là V(i) cộng với giá trị lớn nhất có thể chọn trong số các gói {1, 2, ..., i-1} với trọng lượng j - W(i). Tức là:

  F[i,j] = V[i] + F[i-1, j-W[i]].

  Vì theo cách xây dựng trên, F[i,j] sẽ là giá trị lớn nhất trong hai giá trị thu được ở trên.

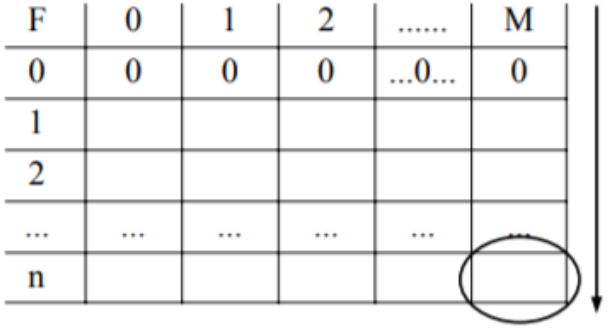
- *Cơ sở quy hoạch động:*

  Dễ thấy F[0,j] = 0. Do không có gói nào để lựa chọn.

- *Tính bảng phương án:*

  Bảng phương án F gồm n+1 dòng, M + 1 cột, trước tiên điền cơ sở quy hoạch động

  Dòng 0 gồm toàn số 0. Sử dụng công thức truy hồi, dùng dòng thứ 0 tính dòng thứ 1, dùng dòng thứ 1 tính dòng thứ 2,.... đến khi tính hết dòng thứ n.



- *Truy vết:*

  Tính xong bảng phương án thì ta quan tâm đến F[n,M], là giá trị lớn nhất thu được khi chọn trong n gói với giới hạn trọng lượng M.

  Nếu F[n,M] = F[n-1,M] thì tức là ta không chọn gói thứ n, ta truy vết tiếp từ F[n-1,M].

  Còn nếu F[n,M] ≠ F[n-1,M], thì ta thông báo rằng phương án tối ưu có chọn gói thứ n và tiếp tục truy vết tiêp từ F[n-1, M-W[n]].

  Cứ tiếp tục cho đến khi đến hàng 0 của bảng phương án.

```
void solve() {
    memset(F, 0, MAX * MAX); // Điền cơ sở quy hoạch động
```

```
    for (int i = 1; i <=n; i ++) {
        for (int j = 0; j <= M; j ++) {
            F[i][j] = F[i-1][j]; //Giả sử không chọn gói thứ i, sau đó đánh giá lại lựa chọn
            if ( (j > W[i]) && (F[i][j] < F[i-1][j-W[i]] + V[i]) )
                F[i][j] = F[i-1][j-W[i]] + V[i];
        }
    }
}

void trace() {
    println(F[n][M]); //in ra giá trị lớn nhất tìm được
    int k = n, l = M;
    while (k > 0) {   //Truy vết từ hàng n đến hàng thứ 0
        if (F[k][l] != F[k-1][l]) { //Chọn gói thứ k
            print(k);               //In kết quả
            l = l - W[k];           //Giảm trọng lượng có thể mang được
        }
        --k;
    }
}
```

3.3 Biến đổi xâu

https://www.spoj.com/problems/EDIST/

Cho xâu ký tự X, xét 3 phép biến đổi:

a. Insert(i, C): chèn ký tự C vào sau vị trí i của xâu X.

b. Replace(i, C): thay ký tự tại vị trí i của xâu X bởi ký tự C.

c. Delete(i): xóa ký tự tại vị trí i của xâu X.

Yêu cầu: cho trước xâu Y, hay tìm số phép biến đổi ít nhất để biến xâu X thành xâu Y với độ dài của xâu X và Y không quá 2000.

Ví dụ: X = "PBBCEFATZ" Y = "QABCDABEFA"

Thì kết quả là 7:

1. PBBCEFAT**Z** → delete(9) → PBBCEFAT
2. PBBCEFA**T** → delete(8) → PBBCEFA
3. PBBCEFA → insert(4,B) → PBBC**B**EFA
4. PBBCBEFA → insert(4,A) → PBBC**A**BEFA
5. PBBCABEFA → insert(4,D) → PBBC**D**ABEFA
6. P**B**BCDABEFA → replace(2,A) → P**A**BCDABEFA
7. **P**ABCDABEFA → replace(1,Q) → **Q**ABCDABEFA

Cách giải:

Đôi với xâu ký tự, việc chèn hay xóa sẽ làm cho các phần tử phía sau vị trí biến đổi bị đánh chỉ số lại, gây khó khăn cho việc quản lý vị trí. Để khắc phục điều này, ta sẽ tìm một thứ tự biến đổi thỏa mãn:

Phép biến đổi tại vị trí i bắt buộc phải thực hiện sau các phép biến đổi tại vị trí i+1, i+2,...

- *Công thức truy hồi*:

Giả sử m là độ dài của xâu X, n là độ dài của xâu Y. Gọi F[i,j] là số phép biến đổi tối thiểu để biên xâu gồm i ký tự đầu của xâu X: X1X2..Xi thành j ký tự đầu tiên của xâu Y: Y1Y2...Yj.

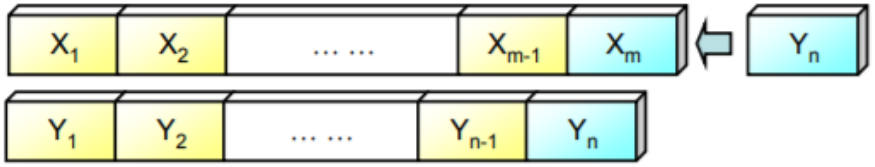
Quan sát hai dãy X, Y:

Ta nhận thấy nếu X(m) = Y(n) thì ta chỉ cần biến đoạn X1X2....X(m-1) thành Y1Y2...Yn-1:

Trong trường hợp này **F[m,n] = F[m-1, n-1]**

Nếu X(m) ≠ Y(n) thì tại vị trí X(m) ta có thể sử dụng một trong ba phép biến đổi:

1. Chèn vào sau vị trị m của X một ký tự đúng bằng Y(n):



Thì khi đó F[m,n] sẽ bằng phép chèn vừa rồi cộng với số phép biến đổi dãy X1...X(m) thành dãy Y1...Y(n-1):

F[m,n] = 1 + F[m, n-1]

2. Thay vị trí m của X bằng một ký tự đúng bằng Y(n):

Thì khi đó F[m,n] sẽ bằng phép 1 phép thay cộng với phép biến đổi dãy X1...X(m-1) thành dãy Y1...Y(n-1):

F[m,n] = 1 + F[m -1, n-1]

3. Xóa vị trí thứ m của X:

Thì khi đó F[m,n] sẽ bằng phép 1 phép thay cộng với phép biến đổi dãy X1...X(m-1) thành dãy Y1...Y(n):

F[m,n] = 1 + F[m -1, n]

Vì F[m,n] phải là nhỏ nhất nên trong trường hợp X(m) ≠ Y(n) thì

F[m,n] = min(F[m, n-1], F[m -1, n-1], F[m -1, n]) + 1

- *Cơ sở quy hoạch động*:

F[0,j] là số phép biến đổi xâu rỗng thành xâu j ký tự. Do đó nó cần tối thiểu j phép chèn:

F[0,j] = j

F[i,0] là số biến đổi xâu i ký tự ban đầu thành xâu rông. Do đó nó cần thối thiểu i phép xóa:

F[i,0] = i

Vậy thì đầu tiên bảng phương án F được khởi tạo hàng 0 và cột 0 là cơ sở quy hoạch động. Từ đó dùng công thức truy hồi để tính ra tất cả các phần tử bảng phương án F.

Sau khi tính xong F[m,n] cho ta biết số phép biến đổi tối thiểu.

- *Truy vết:*

Nếu X(m) = Y(n) thì chỉ việc xét tiếp F[m-1, n-1].

Nếu không, xét ba trường hợp:

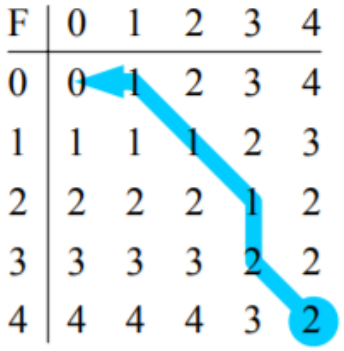
Nếu F[m,n] = F[m,n-1] + 1 thì phép biến đổi đầu tiên được sử dụng là: Insert(m, Y(n))

Nếu F[m,n] = F[m-1,n-1] + 1 thì phép biến đổi đầu tiên được sử dụng là: Replace(m, Y(n))

Nếu F[m,n] = F[m-1,n] + 1 thì phép biến đổi đầu tiên được sử dụng là: Delete(m)

Đưa về bài toán m,n nhỏ hơn và truy vết tiếp cho tới khi về F[0,0].

Ví dụ với X = "ABCD", Y = "EABD" thì bảng phương án là:



F	0	1	2	3	4
0	0	1	2	3	4
1	1	1	1	2	3
2	2	2	2	1	2
3	3	3	3	2	2
4	4	4	4	3	2

Thực hiện thuật toán:

```
void solve() {
    for (int j = 0; j <= n; ++j) F[0][j] = j; //Cơ sở quy hoạch động
    for (int i = 1; i <= m; ++i) F[i][0] = i; //Cơ sở quy hoạch động

    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
```

```
            if (X[i] == Y[j])
                F[i][j] = F[i-1][j-1];
            else
                F[i][j] = min(F[i][j-1], F[i-1][j-1], F[i-1][j]) + 1;
        }
    }
}

void trace() {
    int k =m, l = n;
    println(F[k][l]); //In ra số phép biến đổi ít nhất cần thực hiện.
    while ((k > 0) || (l > 0)) {
        if (X[k] == Y[l]) { //Hai ký tự cuối của xâu giống nhau
            --k; --l;
        } else {
            if (l > 0 && F[k][l] == F[k][l-1] + 1) { //Phép chèn
                println("Insert(" + m +"," + Y[l]);
                --k;
            } else if ( k > 0 && l > 0 && F[k][l] == F[k-1][l-1] + 1) { //Phép thay thế
                println("Replace(" + m +"," + Y[l]);
                --k; --l;
            } else {  //Phép xóa
                println("Delete(" + m +"," + Y[l]);
            }
        }
    }
}
```

4. Summarry

Một bài toán quy hoạch động có thể có nhiều cách tiếp cận khác nhau. Chọn cách nào là tùy theo yêu cầu bài toán sao cho dễ dàng cài đặt nhất. Phương pháp quy hoạch động thường không khó khăn trong việc tính bảng phương án, không khó khăn trong việc tìm cơ sở quy hoạch động, mà khó khắn chính là **nhìn nhận ra bài toán quy hoạch động** và **tìm công thức truy hồi**. Công việc này đòi hỏi sự nhanh nhạy, khôn khéo và chỉ từ sự **rèn luyện mới có được**.

5. Discussion questions

1. Sự giống và khác nhau của quy hoạch động và chia để trị như thế nào?

No labels