Chương 1. Quy hoạch động

Trong khoa học tính toán, quy hoạch động là một phương pháp hiệu quả để giải các bài toán tối ưu mang bản chất đệ quy.

Tên gọi "*Quy hoạch động*" – (*Dynamic Programming*) được Richard Bellman đề xuất vào những năm 1940 để mô tả một phương pháp giải các bài toán mà người giải cần đưa ra lần lượt một loạt các quyết định tối ưu. Tới năm 1953, Bellman chỉnh lại tên gọi này theo nghĩa mới và đưa ra nguyên lý giải quyết các bài toán bằng phương pháp quy hoạch động.

Mục đích của bài này là cung cấp một cách tiếp cận mới trong việc giải quyết các bài toán tối ưu mang bản chất đệ quy, đồng thời đưa ra các ví dụ để người đọc hình thành các kỹ năng trong việc tiếp cận và giải quyết các bài toán quy hoạch động.

1.1. Công thức truy hồi

1.1.1. Bài toán ví dụ

Chúng ta sẽ tìm hiểu về công thức truy hồi và phương pháp giải công thức truy hồi qua một ví dụ:

Cho số tự nhiên $n \leq 300$. Hãy cho biết có bao nhiêu cách phân tích số n thành tổng của dãy các số nguyên dương, các cách phân tích là hoán vị của nhau chỉ tính là một cách.

Ví dụ: $n = 5$ có 7 cách phân tích:

```
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
```

Ta coi trường hợp $n = 0$ cũng có 1 cách phân tích thành tổng các số nguyên dương (0 là tổng của dãy rỗng)

Chúng ta đã biết phương pháp liệt kê tất cả các cách phân tích và đếm số cấu hình. Bây giờ hãy thử nghĩ xem, có cách nào *tính ngay ra số lượng các cách phân tích mà không cần phải liệt kê* hay không. Bởi vì khi số cách phân tích tương đối lớn, phương pháp liệt kê tỏ ra khá chậm. (Ví dụ chỉ với $n = 100$ đã có 190.569.292 cách phân tích và khi $n = 300$ có tới 9.253.082.936.723.602 cách phân tích).

1.1.2. Đặt công thức truy hồi

Nếu gọi $f(m, v)$ là số cách phân tích số v thành tổng các số nguyên dương $\leq m$. Khi đó các cách phân tích số v thành tổng một dãy các số nguyên dương $\leq m$ có thể chia làm hai loại:

- Loại 1: Không chứa số m trong phép phân tích, khi đó số cách phân tích loại này chính là số cách phân tích số v thành tổng các số nguyên dương $\leq m-1$, tức là bằng $f(m-1, v)$.
- Loại 2: Có chứa ít nhất một số m trong phép phân tích. Khi đó nếu trong các cách phân tích loại này ta bỏ đi số m đó thì ta sẽ được các cách phân tích số $v-m$ thành tổng các số nguyên dương $\leq m$. Có nghĩa là về mặt số lượng, số các cách phân tích loại này bằng $f(m, v-m)$

Trong trường hợp $m > v$ thì rõ ràng chỉ có các cách phân tích loại 1, còn trong trường hợp $m \leq v$ thì sẽ có cả các cách phân tích loại 1 và loại 2. Vì thế:

$$f(m, v) = \begin{cases} f(m-1, v), \text{nếu } m > v \\ f(m-1, v) + f(m, v-m), \text{nếu } m \leq v \end{cases} \quad (1.1)$$

Ta có công thức xây dựng $f(m, v)$ từ $f(m-1, v)$ và $f(m, v-m)$. Công thức này có tên gọi là *công thức truy hồi* (*recurrence*) đưa việc tính $f(m, v)$ về việc tính các $f(m', v')$ với dữ liệu nhỏ hơn. Cuối cùng ta sẽ quan tâm đến $f(n, n)$: Số các cách phân tích n thành tổng các số nguyên dương $\leq n$.

Hàm f có thể biểu diễn bởi một mảng hai chiều, ta đồng nhất giá trị $f(m, v)$ với phần tử $f[m][v]$ của bảng f. Ví dụ với $n = 5$, các giá trị $f[m][v]$ có thể cho bởi bảng:

f	0	1	2	3	4	5	v
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	
m							

Công thức (1.1) cho thấy mỗi phần tử $f[m][v]$ được tính qua giá trị của một phần tử ở hàng trên ($f[m-1][v]$) và một phần tử ở cùng hàng, bên trái: ($f[m][v-m]$). Ví dụ $f[5][5]$ sẽ được tính bằng $f[4][5] + f[5][0]$, hay $f[3][5]$ sẽ được tính bằng $f[2][5] + f[3][2]$. Chính vì vậy để tính $f[m][v]$ thì phần tử ở hàng trên ($f[m-1][v]$) và phần tử cùng hàng bên trái ($f[m, v-m]$) phải được tính trước. Thứ tự tính hợp lý là tính lần lượt các hàng từ trên xuống và trên mỗi hàng thì tính theo thứ tự từ trái qua phải.

Điều đó có nghĩa là ban đầu ta phải tính hàng 0 của bảng. Theo định nghĩa $f[0][v]$ là số dãy có các phần tử là số nguyên dương ≤ 0 mà tổng bằng v, theo quy ước đã đặt ra thì $f[0][0] = 1$ và dễ thấy rằng $f[0][v] = 0, \forall v > 0$.

Bây giờ giải thuật đếm trở nên rất đơn giản:

- Khởi tạo hàng 0 của bảng f: $f[0][0] = 1$; $f[0][v] = 0, \forall v > 0$;
- Dùng công thức truy hồi tính ra tất cả các phần tử của bảng f
- Cuối cùng cho biết $f[n][n]$ là số cách phần tích cần tìm

Chúng ta sẽ cài đặt chương trình đếm số cách phân tích số với khuôn dạng nhập/xuất dữ liệu như sau:

Input

Số tự nhiên $n \leq 300$

Output

Số cách phân tích số n

Sample Input	Sample Output
300	9253082936723602 analyses

NUMBERPARTITIONING1_DP.cpp Đếm số cách phân tích số

```
#include <iostream>
#include <algorithm>
using namespace std;
const int maxN = 300;
int n;
long long f[maxN + 1][maxN + 1];

int main()
{
    cin >> n;
    //Khởi tạo dòng 0 của bảng
    f[0][0] = 1;
    fill(f[0] + 1, f[0] + n + 1, 0LL);
    //Giải công thức truy hồi
    for (int m = 1; m <= n; ++m)
        for (int v = 0; v <= n; ++v)
            f[m][v] = v < m ?
                      f[m - 1][v] : f[m - 1][v] + f[m][v - m];
    cout << f[n][n] << " analyses";
}
```

1.1.3. Cải tiến thứ nhất

Cách làm ở mục 1.1.2 có thể tóm tắt lại như sau: Khởi tạo hàng 0 của bảng, sau đó dùng hàng 0 tính hàng 1, dùng hàng 1 tính hàng 2, v.v... tới khi tính được hết hàng n. Có thể nhận thấy rằng khi đã tính xong hàng thứ m thì việc lưu trữ các hàng từ 0 tới $m-1$ là không cần thiết bởi vì việc tính hàng $m+1$ chỉ phụ thuộc các giá trị lưu trữ trên hàng m.

Cách tính này dựa vào quan sát trên để chỉ cần lưu trữ hai hàng của mảng f biểu diễn như hai mảng một chiều: Mảng một chiều x tương ứng với hàng vừa được tính của bảng f và mảng một chiều y tương ứng với hàng sắp tính của bảng f: Đầu tiên mảng x được gán các giá trị tương ứng trên hàng 0 của bảng f; sau đó, công thức truy hồi sẽ được áp dụng để tính mảng y từ mảng x. Tiếp theo, hai mảng x và y sẽ đổi vai trò cho nhau và công thức truy hồi tiếp tục dùng mảng x tính mảng y, mảng y sau khi tính sẽ gồm các giá trị tương ứng trên hàng 2 của bảng f, v.v... Vậy ta có một cách cài đặt tiết kiệm bộ nhớ hơn:

NUMBERPARTITIONING2_DP.cpp Đếm số cách phân tích số

```
#include <iostream>
#include <algorithm>
using namespace std;
const int maxN = 300;
int n;
long long x[maxN + 1], y[maxN + 1];

```

```
int main()
{
    cin >> n;
    //Khởi tạo mảng x: x[v] = f[0][v], ∀v
    x[0] = 1;
    fill(x + 1, x + n + 1, 0LL);
    //Giải công thức truy hồi
    for (int m = 1; m <= n; ++m)
    {   //Dùng mảng x tính mảng y
        for (int v = 0; v <= n; ++v)
            y[v] = v < m ?
                   x[v] : x[v] + y[v - m];
        swap(x, y); //Đảo vai trò mảng x và mảng y
    }
    cout << x[n] << " analyses"; //Cuối cùng, mảng x ứng với dòng n của bảng f
}
```

Chương trình còn có thể cải thiện tốc độ hơn nữa bằng cách dùng `x, y` là hai con trỏ tới hai mảng, khi đó thao tác `std:swap(x, y)` trở thành thao tác đảo hai con trỏ sẽ nhanh hơn so với đảo giá trị từng cặp phần tử trên hai mảng. Một cách khác là khai báo `x, y` dưới dạng mảng động (vector), khi đó phương thức `swap` của lớp mẫu `std::vector` cũng cho phép hoán đổi phần tử của hai mảng động trong thời gian O(1). Việc thử nghiệm những cách nâng cấp này ta coi như bài tập

1.1.4. Cải tiến thứ hai

Nhận xét sau cho phép một cách cài đặt tốt hơn nữa: hàng m của bảng f không những tính được chỉ dựa vào hàng liền trước mà các phần tử $f[m][v]$ còn giữ nguyên giá trị so với $f[m-1][v]$ nếu $v < m$. Trường hợp $v \geq m$ thì $f[m][v]$ tăng thêm so với $f[m-1][v]$ một lượng bằng $f[m][v-m]$, lượng tăng này đúng bằng một phần tử cùng hàng m đã tính. Dựa vào nhận xét này, ta chỉ cần lưu lại một hàng của bảng f như mảng một chiều, sau đó áp dụng công thức truy hồi lên mảng đó tính lại chính nó, để sau khi tính, mảng một chiều sẽ chứa các giá trị trên hàng kế tiếp của bảng f.

NUMBERPARTITIONING3_DP.cpp Đếm số cách phân tích số

```
#include <iostream>
#include <algorithm>
using namespace std;
const int maxN = 300;
int n;
long long f[maxN + 1]; //Chỉ cần lưu một hàng của bảng f

int main()
{
    cin >> n;
    //Khởi tạo mảng 1 chiều f ứng với hàng 0
    f[0] = 1;
    fill(f + 1, f + n + 1, 0LL);
    //Giải công thức truy hồi
    for (int m = 1; m <= n; ++m)
        for (int v = m; v <= n; ++v) //Chỉ cần tính lại f[v] với v ≥ m
            f[v] += f[v - m]; //f[v] tăng lên một lượng bằng f[v - m]
    cout << f[n] << " analyses";
}
```

1.1.5. Cài đặt đệ quy

Xem lại công thức truy hồi tính $f[m][v] = f[m-1][v] + f[m][v-m]$, ta nhận thấy rằng để tính $f[m][v]$ ta phải biết được chính xác $f[m-1][v]$ và $f[m][v-m]$. Như vậy việc xác

định thứ tự tính các phần tử trong bảng f (phần tử nào tính trước, phần tử nào tính sau) là quan trọng. Trong trường hợp thứ tự này khó xác định, ta có thể tính dựa trên một hàm đệ quy mà không cần phải quan tâm tới thứ tự tính toán. Trước khi trình bày phương pháp này, ta sẽ thử viết một hàm đệ quy theo cách quen thuộc để giải công thức truy hồi:

NUMBERPARTITIONING4_DC.cpp Đếm số cách phân tích số

```cpp
#include <iostream>
using namespace std;
int n;

long long f(int m, int v)
{
    if (m == 0)
        return v == 0 ? 1 : 0; //phần neo
    return v < m ?
        f(m - 1, v) : f(m - 1, v) + f(m, v - m); //phần đệ quy
}

int main()
{
    cin >> n;
    cout << f(n, n) << " analyses";
}
```

Phương pháp cài đặt này tỏ ra khá chậm do với mỗi giá trị m và v, hàm $f(m, v)$ sẽ bị gọi nhiều lần (có thể thấy qua thực nghiệm, ta sẽ giải thích rõ hơn trong bài sau). Có thể cải tiến bằng cách kết hợp hàm đệ quy $f(m, v)$ với một bảng lưu trữ giá trị g. Ban đầu các phần tử của g được coi là "chưa biết" (bằng cách dùng thêm một bảng đánh dấu hoặc gán một giá trị đặc biệt). Hàm $f(m, v)$ sẽ tra cứu tới phần tử $g[m][v]$ theo cách sau:

- Nếu $g[m][v]$ chưa biết thì hàm $f(m, v)$ sẽ gọi đệ quy để tính sau đó lưu kết quả vào $g[m][v]$
- Còn nếu $g[m][v]$ đã biết thì hàm $f(m, v)$ chỉ việc trả về kết quả là $g[m][v]$ mà không cần gọi đệ quy để tính toán nữa.

NUMBERPARTITIONING5_DP.cpp Đếm số cách phân tích số

```cpp
#include <iostream>
#include <algorithm>
using namespace std;
const int maxN = 300;
int n;
long long g[maxN + 1][maxN + 1];

long long f(int m, int v)
{
    if (g[m][v] == -1) //g[m][v] == -1, chưa biết, tính g[m][v] bằng đệ quy
        if (m == 0)
            g[m][v] = v == 0 ? 1 : 0; //Phần neo
        else
            g[m][v] = v < m ?
                f(m - 1, v) : f(m - 1, v) + f(m, v - m); //Phần đệ quy
    return g[m][v]; //Lấy giá trị g[m][v] trả về kết quả hàm
}

int main()
{
    cin >> n;
    fill(g[0], g[n + 1], -1LL); //Các giá trị bảng g đều bằng -1: chưa biết
    cout << f(n, n) << " analyses";
}
```

Việc sử dụng phương pháp đệ quy để giải công thức truy hồi là một kỹ thuật đáng lưu ý, vì khi gặp một công thức truy hồi phức tạp, khó xác định thứ tự tính toán thì phương pháp này tỏ ra rất hiệu quả, hơn thế nữa nó làm rõ hơn bản chất đệ quy của công thức truy hồi.

Thêm nữa, nếu khảo sát giá trị bảng g trong một trường hợp cụ thể, chẳng hạn $n = 5$:

g	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	-1	3
3	1	1	2	-1	-1	5
4	1	1	-1	-1	-1	6
5	1	-1	-1	-1	-1	7

Ta thấy còn rất nhiều phần tử mang giá trị -1 (chưa được tính), cài đặt đệ quy cho phép chúng ta bỏ qua những phần tử không tham gia vào quá trình tính $f(n, n)$, điều này trong một số trường hợp có thể cải thiện tốc độ đáng kể.

1.1.6. Tóm tắt kỹ thuật giải công thức truy hồi

Công thức truy hồi có vai trò quan trọng trong bài toán đếm, chẳng hạn đếm số cấu hình thoả mãn điều kiện nào đó, hoặc tìm tương ứng giữa một số thứ tự và một cấu hình.

Ví dụ trong bài này là thuộc dạng đếm số cấu hình thoả mãn điều kiện nào đó. Tuy nhiên nếu đặt lại bài toán: Nếu đem tất cả các dãy số nguyên dương không giảm có tổng bằng n sắp xếp theo thứ tự từ điển thì dãy thứ p là dãy nào?, hoặc cho một dãy số nguyên dương không giảm có tổng bằng n, hỏi dãy đó đứng thứ mấy?. Lúc này ta sẽ có bài toán tìm tương ứng giữa một số thứ tự và một cấu hình - Một dạng bài toán có thể giải quyết hiệu quả bằng công thức truy hồi.

Ta cũng đã khảo sát một vài kỹ thuật phổ biến để giải công thức truy hồi: Phương pháp tính trực tiếp bảng giá trị, phương pháp tính luân phiên (chỉ phải lưu trữ các phần tử tích cực), phương pháp tự cập nhật giá trị, và phương pháp đệ quy kết hợp với bảng lưu trữ. Những phương pháp này sẽ đóng vai trò quan trọng trong việc cài đặt chương trình giải công thức truy hồi - Hạt nhân của bài toán quy hoạch động.

1.1.7. ★ Nói thêm về bài toán phân tích số

Bài toán *phân tích số* (*integer partitioning*) là một bài toán quan trọng trong lĩnh vực số học và vật lý. Chúng ta không đi sâu vào chi tiết hai lĩnh vực này mà chỉ nói tới một kết quả đã được chứng minh bằng lý thuyết số học về các *số ngũ giác* (*pentagonal numbers*):

Những số ngũ giác là những số nguyên có dạng $p(k) = \frac{k(3k-1)}{2}$ với $k \in \mathbb{Z}_{>0}$. Bằng cách cho k nhận lần lượt các giá trị trong dãy $+1, -1, +2, -2, +3, -3, \ldots$ ta có thể liệt kê các số ngũ giác theo thứ tự tăng dần là:

$$1,2,5,7,12,15, \ldots$$

Gọi $f(n)$ là số cách phân tích số n thành tổng của các số nguyên dương (không tính hoán vị). Quy ước $f(0) = 1$ và $f(n) = 0$ với $\forall n < 0$, khi đó:

$$f(n) = \sum_{k=1}^{\infty} (-1)^{k+1} \left(f\left(n - p(k)\right) + f\left(n - p(-k)\right) \right) \\ = f(n-1) + f(n-2) - f(n-5) - f(n-7) + \cdots \tag{1.2}$$

Chúng ta có thể sử dụng công thức truy hồi (1.2) để đếm số cách phân tích số trong thời gian $O\left(n\sqrt{n}\right)$ thay vì $\Theta(n^2)$ nếu dùng công thức (1.1) như các chương trình trước.

NUMBERPARTITIONING6_DP.cpp Đếm số cách phân tích số

```cpp
#include <iostream>
using namespace std;
const int maxN = 300;
int n;
long long f[maxN + 1];

int p(int k) //Số ngũ giác p(k)
{
    return k * (3 * k - 1) / 2;
}

int main()
{
    cin >> n;
    f[0] = 1;
    for (int i = 1; i <= n; ++i)
    {   //Tính f[i]
        f[i] = 0;
        for (int k = 1; i >= p(k); ++k)
        {
            long long t = f[i - p(k)];
            if (i >= p(-k))
                t += f[i - p(-k)]; //t = f[i - p(k)] + f[i - p(-k)]
            if (k % 2 == 1)
                f[i] += t;
            else
                f[i] -= t;
        }
    }
    cout << f[n] << " analyses";
}
```

1.2. Phương pháp quy hoạch động

1.2.1. Bài toán quy hoạch động

Trong toán học và khoa học máy tính, *quy hoạch động* (*dynamic programming*) là một phương pháp hiệu quả giải những bài toán tối ưu có ba tính chất sau đây:

- Bài toán lớn có thể phân rã thành những bài toán con đồng dạng, những bài toán con đó có thể phân rã thành những bài toán nhỏ hơn nữa ...(*recursive form*).
- Lời giải tối ưu của các bài toán con có thể sử dụng để tìm ra lời giải tối ưu của bài toán lớn (*optimal substructure*)
- Hai bài toán con trong quá trình phân rã có thể có chung một số bài toán con khác (*overlapping subproblems*).

Tính chất thứ nhất và thứ hai là điều kiện cần của một bài toán quy hoạch động. Tính chất thứ ba (bài toán con gối nhau) nêu lên đặc điểm của một bài toán mà cách giải bằng phương pháp quy hoạch động hiệu quả hơn so với phương pháp chia để trị thuần túy.

1.2.2. So sánh với phương pháp chia để trị

Với những bài toán có hai tính chất đầu tiên nêu ở mục 1.2.1, chúng ta thường nghĩ đến các thuật toán chia để trị và đệ quy: Để giải quyết một bài toán lớn, ta chia nó ra thành nhiều bài toán con đồng dạng và giải quyết độc lập các bài toán con đó.

Khác với thuật toán đệ quy, phương pháp quy hoạch động thêm vào cơ chế lưu trữ nghiệm hay một phần nghiệm của mỗi bài toán khi giải xong nhằm mục đích *sử dụng lại*, hạn chế những thao tác thừa trong quá trình tính toán.

Xét một ví dụ ở chương trước: Dãy Fibonacci là dãy vô hạn các số nguyên dương $f_0, f_1, \ldots$ được định nghĩa bằng công thức truy hồi sau:

$$f_n = \begin{cases} n, \text{nếu } n < 2 \\ f_{n-1} + f_{n-2}, \text{nếu } n \geq 2 \end{cases}$$

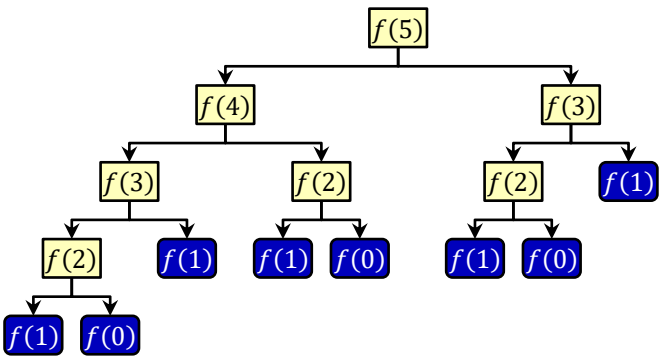
Ví dụ 10 phần tử đầu tiên của dãy Fibonacci: $f[0 \ldots 9] = (0,1,1,2,3,5,8,13,21,34)$

Hãy tính f_5.

Xét cách tính bằng phương pháp chia để trị:

```
01 #include <iostream>
02 using namespace std;
03
04 int f(int n)
05 {
06     if (n < 2) return n;
07     else return f(n - 1) + f(n - 2);
08 }
09
10 int main()
11 {
12     cout << f(5);
13 }
```

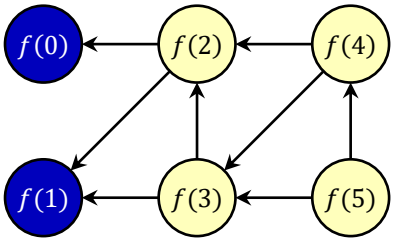
Trong cách giải này, hàm đệ quy $f(i)$ được dùng để tính số Fibonacci thứ i. Khi gọi hàm $f(5)$, nó sẽ gọi tiếp $f(4)$ và $f(3)$ để tính ... Quá trình tính toán có thể vẽ như cây trong Hình 1-1.



Hình 1-1. Hàm đệ quy tính số Fibonacci

Theo thuật toán này, để tính $f(5)$, máy phải tính một lần $f(4)$, hai lần $f(3)$, ba lần $f(2)$, năm lần $f(1)$ và ba lần $f(0)$. Sự thiếu hiệu quả có thể giải thích bởi tính chất: Hai bài toán trong quá trình phân rã có thể có chung một số bài toán con. Trong ví dụ này nếu coi lời gọi hàm $f(n)$ tương ứng với bài toán tính số Fibonacci thứ n, thì bài toán $f(3)$ là bài toán con chung của cả $f(5)$ và $f(4)$. Điều đó chỉ ra rằng nếu tính $f(5)$ và $f(4)$ một cách độc lập,

bài toán $f(3)$ sẽ phải giải hai lần. Tính chất bài toán con gối nhau được chỉ ra trong Hình 1-2.



Hình 1-2. Tính chất bài toán con gối nhau

Để khắc phục những nhược điểm trong việc giải độc lập các bài toán con, mỗi khi giải xong một bài toán trong quá trình phân rã, chúng ta sẽ *lưu trữ lời giải* của nó nhằm mục đích *sử dụng lại*. Chẳng hạn với bài toán tính số Fibonacci thứ 5, ta có thể dùng một mảng $f[0 \ldots 5]$ để lưu trữ các giá trị số Fibonacci, khởi tạo sẵn giá trị cho $f[0] = 0$ và $f[1] = 1$, từ đó tính tiếp $f[2]$, $f[3]$, $f[4]$, $f[5]$, đảm bảo mỗi giá trị Fibonacci chỉ phải tính một lần:

```
01 #include <iostream>
02 using namespace std;
03
04 int f[6];
05
06 int main()
07 {
08     f[0] = 0; f[1] = 1;
09     for (int i = 2; i <= 5; ++i)
10         f[i] = f[i - 1] + f[i - 2];
11     cout << f[5];
12 }
```

Kỹ thuật lưu trữ lời giải của các bài toán con nhằm mục đích sử dụng lại được gọi là *memoization**, nếu tới một bước nào đó mà lời giải một bài toán con không còn cần thiết nữa, chúng ta có thể bỏ lời giải đó đi khỏi không gian lưu trữ để tiết kiệm bộ nhớ. Ví dụ khi ta đang cần tính f_5 thì chỉ cần biết giá trị f_4 và f_3 là đủ nên việc lưu trữ các giá trị $f_{0\ldots2}$ là vô nghĩa. Những kỹ thuật này chúng ta đã bàn đến trong mục 1.1.3 và 1.1.4. Cụ thể bài toán tính số Fibonacci, ta có thể cài đặt dùng hai biến với vai trò luân phiên:

```
1 a = 0; b = 1;
2 for (i = 2; i <= 5; ++i)
3 {
4     b += a; //b = f[i]
5     a = b - a; //a = f[i - 1]
6 }
7 Output ← b;
```

Cách giải này bắt đầu từ việc giải bài toán nhỏ nhất, lưu trữ nghiệm và từ đó giải quyết những bài toán lớn hơn… Công thức truy hồi khi đó được giải từ dưới lên (*bottom-up*), phép giải từ dưới lên không cần có chương trình con đệ quy, vì vậy bớt đi một số đáng kể lời gọi chương trình con và tiết kiệm được bộ nhớ chứa tham số và biến địa phương của chương trình con đệ quy. Tuy nhiên, với phép giải từ dưới lên, chúng ta phải xác định rõ

* Từ này khá mới trong tiếng Anh, có thể coi như từ đồng nghĩa với *memorization*, gốc từ: *memo* (Bản ghi nhớ)

thứ tự giải các bài toán để khi bắt đầu giải một bài toán nào đó thì tất cả các bài toán con của nó phải được giải sẵn từ trước, điều này đôi khi khá phức tạp (xác định thứ tự tô pô trên tập các bài toán phân rã). Trong trường hợp khó xác định thứ tự hợp lý giải quyết các bài toán con, người ta sử dụng phép giải công thức truy hồi từ trên xuống (*top-down*) kết hợp với kỹ thuật lưu trữ nghiệm (*memoization*), kỹ thuật này chúng ta cũng đã bàn đến trong mục 1.1.5. Với bài toán tính số Fibonacci, ta có thể cài đặt bằng một hàm đệ quy:

```
01 #include <iostream>
02 using namespace std;
03 int g[6];
04
05 int f(int n)
06 {
07     if (g[n] == -1) //g[n] chưa biết, đi tính
08         if (n < 2) g[n] = n;
09         else g[n] = f(n - 1) + f(n - 2);
10     return g[n];
11 }
12
13 int main()
14 {
15     fill(g, g + 6, -1);
16     cout << f(5);
17 }
```

Trước khi đi vào phân tích cách tiếp cận một bài toán quy hoạch động, ta làm quen với một số thuật ngữ sau đây:

- Bài toán giải theo phương pháp quy hoạch động gọi tắt là *bài toán quy hoạch động**. Máy tính cần giải *một trường hợp* của bài toán quy hoạch động gọi là bài toán cụ thể xác định bởi dữ liệu là các tham số cho trường hợp cụ thể đó (ví dụ tham số n trong bài toán phân tích số hay tham số n trong bài toán tính số fibonacci). Khái niệm *bài toán lớn/bài toán con* là đề cập tới những *bài toán cụ thể*, trong đó tính lớn/nhỏ của bài toán cụ thể được xác định bởi tham số của bài toán đó.
- Công thức phối hợp nghiệm của các bài toán con để có nghiệm của bài toán 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 gọi là *cơ sở quy hoạch động*.
- Không gian lưu trữ lời giải các bài toán con để tìm cách phối hợp chúng gọi là *bảng phương án* của quy hoạch động.
- Trong lý thuyết tối ưu hóa, hàm đánh giá mức độ "tốt" của một lời giải cho một bài toán cụ thể gọi là *hàm mục tiêu* (*objective function*). Trong các bài toán quy hoạch động, khái niệm hàm mục tiêu được mở rộng ra cho cả các hàm đếm hoặc đơn thuần chỉ là hàm có tác dụng đánh dấu tùy theo bài toán cần giải quyết.

1.2.3. Cách giải một bài toán quy hoạch động

Việc giải một bài toán quy hoạch động phải qua khá nhiều bước, từ những phân tích ở trên, ta có thể tóm tắt:

---

* Có sự bất hợp lý khi tên gọi của bài toán lại phụ thuộc cách giải nó. Tuy nhiên đây là khái niệm dùng phổ biến thành quen, ta không nên sửa đổi.

Quy hoạch động = Chia để trị + Cơ chế lưu trữ nghiệm để sử dụng lại

(Dynamic Programming = Divide and Conquer + Memoization)

Không có một "thuật toán" nào cho chúng ta biết một bài toán có thể giải bằng phương pháp quy hoạch động hay không? Khi gặp một bài toán cụ thể, chúng ta phải phân tích bài toán, sau đó kiểm chứng ba tính chất của một bài toán quy hoạch động (mục 1.2.1) trước khi tìm lời giải bằng phương pháp quy hoạch động.

Nếu như bài toán đặt ra đúng là một bài toán quy hoạch động thì việc đầu tiên phải làm là phân tích xem một thể hiện của bài toán tổng quát xác định bởi những tham số gì, từ đó xác định xem các bài toán bài toán lớn có thể phân rã thành những bài toán con đồng dạng như thế nào. Việc tiếp theo là xây dựng cách tìm nghiệm của bài toán lớn trong điều kiện chúng ta đã biết nghiệm của những bài toán con - *tìm công thức truy hồi*. Đây là hai công đoạn khó nhất vì nó hoàn toàn dựa vào kinh nghiệm và độ nhạy cảm của người lập trình khi đọc và phân tích bài toán tổng quát.

Sau khi xây dựng công thức truy hồi, công đoạn tiếp theo là phải phân tích xem tại mỗi bước tính nghiệm của một bài toán, chúng ta *cần lưu trữ bao nhiêu bài toán con* và với mỗi bài toán con thì cần *lưu trữ toàn bộ nghiệm hay một phần nghiệm*. Từ đó xác định lượng bộ nhớ cần thiết để lưu trữ, nếu lượng bộ nhớ cần huy động vượt quá dung lượng cho phép, chúng ta cần tìm một công thức khác hoặc thậm chí, một cách giải khác không phải bằng quy hoạch động.

Một điều không kém phần quan trọng là phải chỉ ra được *bài toán nào cần phải giải trước bài toán nào*, hoặc chí ít là chỉ ra được khái niệm bài toán này nhỏ hơn bài toán kia nghĩa là thế nào. Nếu không, ta có thể rơi vào quá trình lòng vòng: Giải bài toán A cần phải giải bài toán B, giải bài toán B lại cần phải giải bài toán A. Cũng nhờ chỉ rõ quan hệ "nhỏ hơn" giữa các bài toán mà ta có thể xác định được một tập các bài toán nhỏ nhất có thể giải trực tiếp, nghiệm của chúng được dùng làm cơ sở giải những bài toán lớn hơn.

Trong đa số các bài toán quy hoạch động, nếu bạn tìm được đúng công thức truy hồi và xác định đúng tập các bài toán cơ sở thì coi như phần lớn công việc đã xong. Việc tiếp theo là cài đặt chương trình giải công thức truy hồi:

- Nếu giải công thức truy hồi theo cách tiếp cận từ dưới lên, trước hết cần giải tất cả các bài toán cơ sở, lưu trữ nghiệm vào bảng phương án, sau đó dùng công thức truy hồi tính nghiệm của những bài toán lớn hơn và lưu trữ vào bảng phương án cho tới khi bài toán ban đầu được giải. Cách này yêu cầu phải xác định được chính xác thứ tự giải các bài toán (từ nhỏ đến lớn). Ưu điểm của nó là cho phép dễ dàng loại bỏ nghiệm những bài toán con không dùng đến nữa để tiết kiệm bộ nhớ lưu trữ cho bảng phương án.
- Nếu giải công thức truy hồi theo cách tiếp cận từ trên xuống, bạn cần phải có cơ chế đánh dấu bài toán nào đã được giải, bài toán nào chưa. Nếu bài toán chưa được giải, nó sẽ được phân rã thành các bài toán con và giải bằng đệ quy. Cách này không yêu cầu phải xác định thứ tự giải các bài toán nhưng cũng chính vì vậy, khó khăn sẽ gặp phải nếu muốn loại bỏ bớt nghiệm của những bài toán con không dùng đến nữa để tiết kiệm bộ nhớ lưu trữ cho bảng phương án.

Công đoạn cuối cùng là chỉ ra nghiệm của bài toán ban đầu. Nếu bảng phương án lưu trữ toàn bộ nghiệm của các bài toán thì chỉ việc đưa ra nghiệm của bài toán ban đầu, nếu bảng phương án chỉ lưu trữ một phần nghiệm, thì trong quá trình tính công thức truy hồi, ta để lại một "manh mối" nào đó (gọi là vết) để khi kết thúc quá trình giải, ta có thể truy vết tìm ra toàn bộ nghiệm.

Chúng ta sẽ khảo sát một số bài toán quy hoạch động kinh điển để rút ra những kỹ thuật cơ bản trong việc phân tích bài toán quy hoạch động, tìm công thức truy hồi cũng như lập trình giải các bài toán quy hoạch động.

1.3. Một số bài toán quy hoạch động

1.3.1. Dãy con tăng dài nhất

Bài toán *dãy con tăng dài nhất* (*Longest Increasing Subsequence - LIS*) phát biểu như sau: Cho dãy số nguyên $A = a[0 \dots n)$. Một dãy con của A là một cách chọn ra trong A một số phần tử giữ nguyên thứ tự. Yêu cầu đặt ra là tìm dãy con tăng của A có độ dài lớn nhất.

Input

- Dòng 1 chứa số n ($n \leq 10^6$)
- Dòng 2 chứa n số nguyên $a_0, a_1, \dots, a_{n-1}$ ($\forall i: |a_i| \leq 10^9$)

Output

Dãy con tăng dài nhất của dãy A

Sample Input	Sample Output
12 1 2 3 8 9 4 6 2 7 3 9 5	Length = 7 a[10] = 9 a[8] = 7 a[6] = 6 a[5] = 4 a[2] = 3 a[1] = 2 a[0] = 1

Trong bài toán này và cả các bài toán về sau, ta cần phân biệt khái niệm dãy con (*subsequence*) và xâu con (*substring*). Dãy con của một dãy được tạo thành bằng cách chọn ra một số phần tử trong dãy ban đầu theo đúng thứ tự, các phần tử được chọn không nhất thiết phải liên tiếp, trong khi đó xâu con của một dãy được tạo thành bằng cách chọn một số phần tử liên tiếp trong dãy ban đầu, xâu con của một dãy còn có thể gọi là một đoạn con.

Thuật toán

Nhận định đầu tiên là phần tử cuối cùng của dãy con tăng dài nhất (LIS) chắc chắn phải là một phần tử a_i nào đó trong dãy A, nếu bỏ phần tử cuối này đi ta vẫn sẽ thu được một dãy con tăng mà phần tử cuối của dãy con tăng này phải là một phần tử a_j nào đó đứng trước a_i và $a_j < a_i$. Nhận định này cho phép ta xác định hàm mục tiêu dựa vào độ dài dãy con tăng dài nhất kết thúc tại một phần tử a_i nào đó.

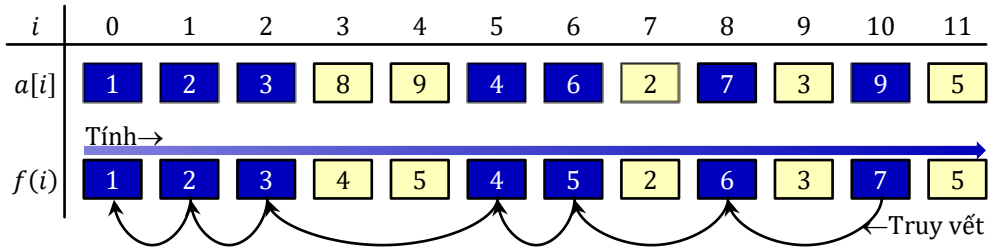
Với mỗi giá trị i $(0 \le i < n)$, gọi $f(i)$ là độ dài LIS kết thúc tại a_i*. Tức là trong tất cả các dãy con đơn điệu tăng có phần tử cuối là a_i, ta gọi $f(i)$ là độ dài của dãy dài nhất trong số đó.

Công thức truy hồi tính $f(i)$: Nếu LIS kết thúc tại a_i có nhiều hơn 1 phần tử, LIS này sẽ được thành lập bằng cách lấy a_i nối vào đuôi một trong số những dãy con tăng kết thúc tại phần tử a_j nào đó đứng trước a_i. Ta sẽ chọn dãy nào để nối thêm a_i vào đuôi?, tất nhiên ta chỉ được nối a_i vào đuôi những dãy con tăng kết thúc tại a_j nào đó nhỏ hơn a_i (để đảm bảo tính tăng) và dĩ nhiên ta sẽ chọn dãy dài nhất để nối a_i vào đuôi (để đảm bảo tính dài nhất). Vậy để tính $f(i)$, ta xét tất cả các phần tử a_j đứng trước và nhỏ hơn a_i, tìm $fmax$ là giá trị lớn nhất trong các $f(j)$ đó và đặt $f(i) = fmax + 1$. Công thức truy hồi là:

$$f(i) = \underbrace{\max\{f(j): j < i \text{ và } a_j < a_i\}}_{fmax} + 1 \qquad (1.3)$$

Nếu không tồn tại phần tử đứng trước a_i và nhỏ hơn a_i, tức là a_i không thể nối vào đuôi bất kỳ một dãy con tăng nào đứng trước nó để được dãy tăng, trong trường hợp này LIS kết thúc tại a_i chỉ có 1 phần tử là chính a_i, tức là $f(i) = 1$. Để tổng quát hóa công thức (1.3) cho trường hợp này, ta quy ước phép lấy max trên tập rỗng cho kết quả là 0. Đây có thể coi là cơ sở của thuật toán quy hoạch động.

Đồng nhất hàm mục tiêu f với mảng các giá trị $f[0 \dots n)$ $(f(i) \equiv f[i])$, ta nhận thấy rằng để tính $f[i]$ thì các giá trị $f[j]$ với $j < i$ phải được tính trước. Tức là thứ tự tính toán đúng phải là tính từ đầu mảng f tới cuối mảng f: $f[0], f[1], \dots, f[n-1]$



Hình 1-3. Giải công thức truy hồi và truy vết

Sau khi tính xong các giá trị $f[0 \dots n)$ thì giá trị lớn nhất trong mảng f chính là độ dài LIS cần tìm. Việc chỉ ra LIS được thực hiện bằng một kỹ thuật gọi là *truy vết* (*tracing*), tức là lần ngược lại quá trình tính toán để xác định xem giá trị lớn nhất trong mảng f được hình thành như thế nào, từ đó đưa ra sự lựa chọn phần tử vào LIS.

* Thực ra phải gọi là độ dài LIS kết thúc ở vị trí i mới chính xác (do dãy A có thể có nhiều phần tử bằng nhau). Nhưng ta muốn nhấn mạnh về phần tử cuối của mỗi LIS nên có thể hiểu khái niệm a_i trong bài toán này là tham chiếu tới phần tử chỉ số i của dãy A chứ không phải giá trị phần tử chỉ số i trong dãy A.

Xét quá trình tìm một dãy con tăng độ dài m của $a[0 \dots n-1]$ mà phần tử cuối cùng được chọn phải nhỏ hơn b. Ban đầu m là giá trị lớn nhất trong mảng f và $b = +\infty$, quá trình này tương đương với việc tìm LIS trên dãy A. Khi tìm thấy một chỉ số i thỏa mãn $a[i] < b$ và $f[i] = m$, ta chọn luôn $a[i]$ và quá trình lặp lại với việc tìm dãy con tăng độ dài $m-1$ của dãy $a[0 \dots i-1]$ mà phần tử đầu tiên được chọn phải nhỏ hơn $a[i]$... Ý tưởng đó thể hiện qua một thuật toán lặp:

```
m = max{f[0 … n)};
b = +∞;
for (i = n – 1; i ≥ 0; --i)
    if (a[i] < b && f[i] == m) //Tìm thấy i: a[i] < b và f[i] = m
    {
        «Thông báo chọn a[i]»;
        --m;      // Quy về tìm dãy con tăng của a[0 … i) độ dài --m
        b = a[i]; // mà phần tử cuối < a[i]
    }
```

❋ Cài đặt

Ta sẽ cài đặt chương trình giải bài toán tìm dãy con đơn điệu tăng dài nhất theo những phân tích trên.

LIS_DP.cpp Tìm dãy con đơn điệu tăng dài nhất

```cpp
#include <iostream>
#include <algorithm>
using namespace std;
const int maxN = 1e6;
const int infty = 1e9 + 1;

int n, a[maxN], f[maxN];

void ReadInput() //Nhập dữ liệu
{
    cin >> n;
    for (int i = 0; i < n; ++i) cin >> a[i];
}

void SolveRecurrence() //Quy hoạch động
{
    for (int i = 0; i < n; ++i)
    {
        //Tính fmax = max{f[j]: j < i và a[j] < a[i]}
        int fmax = 0; //Quy ước max trên tập Ø bằng 0
        for (int j = 0; j < i; ++j)
            if (a[j] < a[i] && f[j] > fmax) fmax = f[j];
        f[i] = fmax + 1; //Tính f[i] theo công thức truy hồi
    }
}

void Print() //In kết quả
{
    int m = *max_element(f, f + n);
    int b = infty;
    cout << "Length = " << m << '\n';
    for (int i = n - 1; i >= 0; --i)
        if (a[i] < b && f[i] == m) //Tìm thấy i: a[i] < b và f[i] = m
        {
            cout << "a[" << i << "] = " << a[i] << '\n'; //Chọn a[i] vào LIS
            --m;      // Quy về tìm dãy con tăng của a[0...i) độ dài --m
            b = a[i]; // mà phần tử cuối < a[i]
        }
}
```

```
40
41  int main()
42  {
43      ReadInput();
44      SolveRecurrence();
45      Print();
46  }
```

Việc đưa ra chỉ số các phần tử được chọn vào LIS bị ngược, tức là chương trình đưa ra LIS theo thứ tự từ phần tử cuối cùng về phần tử đầu tiên của LIS. Để in ra theo thứ tự xuôi, từ phần tử đầu tiên tới phần tử cuối cùng của LIS, ta có thể lưu trữ lại danh sách các phần tử được chọn sau đó in ra danh sách theo thứ tự ngược lại. Nếu khéo hơn, ta có thể đặt lại hàm mục tiêu: $f[i]$ là độ dài LIS dài nhất bắt đầu tại a_i và quy hoạch động từ cuối mảng f về đầu mảng f, khi đó phép truy vết sẽ cho ta thứ tự đúng của LIS. Vấn đề này ta coi như bài tập.

❋ **Cải tiến**

Với dãy $A = a[0 \dots n)$, gọi $f[i]$ là độ dài LIS kết thúc tại a_i. Nhắc lại công thức truy hồi tính các $f[i]$ là:

$$f[i] = \max\left\{f[j]: j < i \text{ và } a_j < a_i\right\} + 1 \qquad (1.4)$$

Thuật toán đã trình bày ở phần trên cần thời gian $\Omega(n^2)$ để tính mảng f. Với kích thước dữ liệu lớn ($n \approx 10^6$) thì chương trình này chạy rất chậm. Mục này sẽ đề cập tới một cách khác giải công thức truy hồi với độ phức tạp tính toán là $O(n \log m)$ với n là độ dài dãy A và m là độ dài LIS tìm được.

Ý tưởng của phương pháp là mỗi khi tính xong $f[0], f[1], \dots, f[i-1]$, nếu có nhiều dãy con tăng cùng độ dài k của dãy $a[0 \dots i)$ thì chỉ cần quan tâm tới một dãy có phần tử cuối nhỏ nhất (để dễ nối thêm một phần tử khác vào đuôi dãy). Thuật toán sẽ lưu thông tin về phần tử cuối nhỏ nhất của các dãy con tăng độ dài 1, phần tử cuối nhỏ nhất của các dãy con tăng độ dài 2, … và với một quan sát về thứ tự đặc biệt của các thông tin này, khi xét tới phần tử a_i, ta có thể xác định nhanh a_i nên nối vào đuôi dãy con tăng độ dài bao nhiêu là có lợi nhất.

Với một chỉ số i, gọi:

- m là độ dài dãy con tăng dài nhất đứng trước a_i, tức là độ dài LIS của $a[0 \dots i)$.
- Với mỗi độ dài k ($1 \le k \le m$), gọi $e[k]$ là phần tử cuối nhỏ nhất của một dãy con tăng độ dài k đứng trước a_i. Tức là nếu có nhiều dãy con tăng độ dài k đứng trước a_i, ta lấy dãy có phần tử cuối nhỏ nhất và gán $e[k]$ bằng giá trị phần tử cuối của dãy này.

Bổ đề 1-1

Các giá trị $e[1 \dots m]$ được xếp theo thứ tự tăng ngặt:

$$e[1] < e[2] < \dots < e[m]$$

Chứng minh

Thật vậy, với $\forall k: 1 < k \le m$, theo định nghĩa thì $e[k]$ là giá trị phần tử cuối của một dãy con tăng độ dài k. Dãy này nếu xét từ đầu tới phần tử sát cuối là một dãy con tăng độ dài $k-1$. Phần tử sát cuối này nhỏ hơn $e[k]$ do dãy đang xét là dãy tăng, mặt khác, phần tử

sát cuối này chắc chắn $\geq e[k-1]$ (theo định nghĩa của $e[k-1]$). Vậy ta có $e[k-1] < e[k]$ $(\forall k)$.

Các giá trị $f[i]$ vẫn được tính lần lượt theo $i = 0,1, \dots, n-1$. Thông tin của dãy $e[1 \dots m]$ được dùng để tính $f[i]$ và giá trị $f[i]$ sau khi tính xong lại được kết hợp với $a[i]$ để cập nhật dãy $e[1 \dots m]$ mới theo sơ đồ sau:

```
m = 0;
for (int i = 0; i < n; ++i)
{
    «Tính f[i] (dựa vào dãy e bước trước để lại)»;
    «Cực đại hóa m theo f[i]»;
    «Chỉnh lại dãy e theo a[i]»;
}
```

Để tính $f[i]$, ta tìm cách nối a_i vào đuôi một dãy con tăng dài nhất đứng trước a_i và có phần tử cuối $< a_i$. Từ Bổ đề 1-1 về tính tăng ngặt của dãy e, ta có thể dùng thuật toán tìm kiếm nhị phân tìm k nhỏ nhất sao cho $e[k] \geq a_i$:

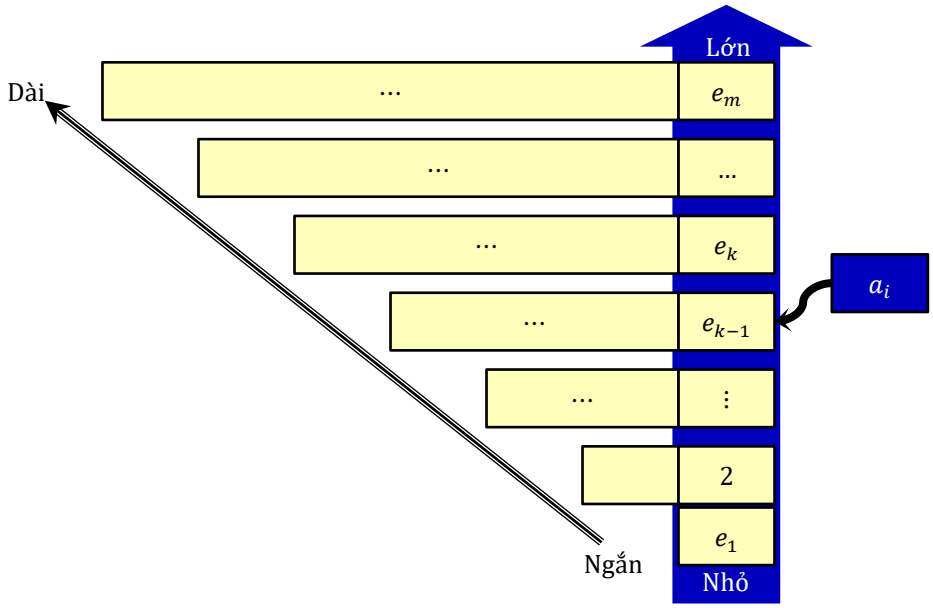
$$e[1] < \cdots < e[k-1] < \boxed{a_i} \leq e[k] < \cdots < e[m]$$

(Nếu không có $e_k \geq a_i$ tức là mọi phần tử $e[.]$ đều nhỏ hơn a_i, ta coi như $k = m + 1$)

Xét dãy con tăng độ dài $k-1$ có phần tử cuối là $e[k-1]$, nếu nối thêm a_i vào cuối dãy này ta sẽ được một dãy tăng độ dài k và vì vậy $f[i] = k$. Thao tác tính $f[i]$ mất thời gian $O(\log m)$ bằng thuật toán tìm kiếm nhị phân.

Việc cực đại hóa m theo $f[i]$ vừa tính được: $m_{\text{mới}} = \max\{m_{\text{cũ}}, f[i]\}$ mất thời gian $O(1)$

Việc cập nhật lại dãy e cũng dựa vào tính tăng dần của dãy đã chỉ ra trong Bổ đề 1-1. Rõ ràng $e[1 \dots k-1]$ không thể thay bằng a_i vì hiện tại chúng đang $< a_i$. LIS kết thúc tại a_i có độ dài k, tức là không có dãy con tăng kết thúc tại a_i có độ dài lớn hơn k, điều này cho thấy $e[k+1 \dots m]$ cũng không thể thay bằng a_i. Chỉ duy nhất $e[k]$ phải thay bằng a_i vì $a_i \leq e[k]$, lý do là vì a_i bây giờ cũng là giá trị cuối của một dãy con tăng độ dài k và giá trị a_i "tốt hơn hoặc bằng" $e[k]$ theo nghĩa: bất kỳ giá trị nào có thể nối thêm vào đuôi dãy con tăng kết thúc tại $e[k]$ đều có thể nối thêm vào đuôi dãy con tăng kết thúc tại a_i (để được dãy con tăng độ dài $k+1$) (Hình 1-4).



Hình 1-4: Cách thức nối thêm phần tử vào đuôi một dãy con tăng

Như vậy việc cập nhật dãy e mất thời gian $O(1)$.

Toàn bộ mảng $f[0 \dots n)$ có thể tính trong thời gian $O(n \log m)$, việc chỉ ra LIS dựa vào mảng f mất thời gian $O(n)$ nên thời gian thực hiện giải thuật này là $O(n \log m)$.

LIS2_DP.cpp Tìm dãy con đơn điệu tăng dài nhất (cải tiến)

```cpp
#include <iostream>
#include <algorithm>
using namespace std;
const int maxN = 1e6;
const int infty = 1e6 + 1;

int n, m, a[maxN], f[maxN], e[maxN + 1];

void ReadInput() //Nhập dữ liệu
{
    cin >> n;
    for (int i = 0; i < n; ++i) cin >> a[i];
}

void SolveRecurrence() //Quy hoạch động
{
    m = 0;
    for (int i = 0; i < n; ++i) //Tính f[i]
    {
        //e[1] < e[2] < ... < e[m]; f[i] = k nhỏ nhất: a[i] ≤ e[k]
        f[i] = lower_bound(e + 1, e + m + 1, a[i]) - e;
        if (f[i] > m) ++m; //Cực đại hóa m theo f[i]
        e[f[i]] = a[i]; //Cập nhật e[k] theo a[i]
    }
}

```

```
27  void Print() //In kết quả
28  {
29      cout << "Length = " << m << '\n';
30      int b = infty;
31      for (int i = n - 1; i >= 0; --i)
32          if (a[i] < b && f[i] == m) //Tìm thấy i: a[i] < b và f[i] = m
33          {
34              cout << "a[" << i << "] = " << a[i] << '\n';
35              --m;       // Quy về tìm dãy con tăng của a[0…i - 1] độ dài --m
36              b = a[i]; // mà phần tử cuối < a[i]
37          }
38  }
39
40  int main()
41  {
42      ReadInput();
43      SolveRecurrence();
44      Print();
45  }
```

Tương tự như cách cài đặt trước, việc đưa ra chỉ số các phần tử được chọn vào LIS bị ngược. Để in ra theo thứ tự xuôi, ngoài cách lưu trữ lại danh sách các phần tử được chọn để in ra danh sách theo thứ tự ngược lại, ta có thể đặt lại hàm mục tiêu: $f[i]$ là độ dài LIS dài nhất bắt đầu tại a_i và quy hoạch động từ cuối mảng f về đầu mảng f. Tất nhiên khi đó ta sẽ phải tính mảng f dựa vào mảng $s[1 \dots m]$ trong đó $s[k]$ là giá trị đầu lớn nhất của một dãy con tăng độ dài k.

1.3.2. Dãy con chung dài nhất

Bài toán *dãy con chung dài nhất* (*Longest Common Subsequence - LCS*) phát biểu như sau: Cho hai xâu ký tự $A = a[0 \dots m)$ và $B = b[0 \dots n)$. Hãy tìm xâu C với độ dài lớn nhất là dãy con của cả A và B. (Xâu X được gọi là dãy con của xâu Y nếu như có thể xóa đi một số ký tự trong Y để được X).*

Input

- Dòng 1 chứa xâu A (độ dài không quá 1000)
- Dòng 2 chứa xâu B (độ dài không quá 1000)

Output

Dãy con chung dài nhất của cả A và B

Sample Input	Sample Output
CCTAAG ACGGTAG	CTAG

❋ Thuật toán

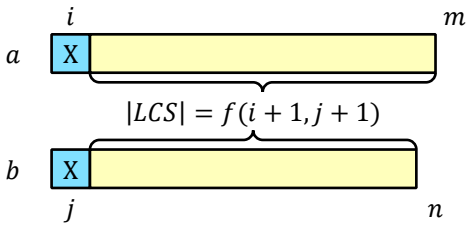
Gọi $f(i, j)$ là độ dài LCS của đoạn cuối xâu A tính từ vị trí i: $a[i \dots m)$ và đoạn cuối xâu B tính từ vị trí j: $b[j \dots n)$. Ta quan tâm tới $f(0,0)$: độ dài LCS của hai xâu A và B.

---

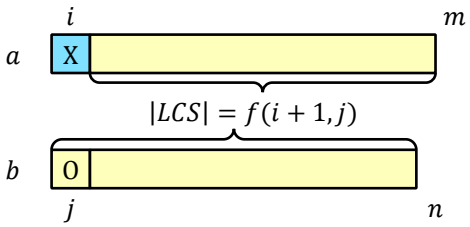
* Như ta đã trình bày, cần phân biệt khái niệm *dãy con (subsequence)* và *xâu con (substring)*. Xâu con của một xâu là một dãy ký tự liên tiếp lấy từ xâu ban đầu. Bài toán *xâu con chung dài nhất* (*Longest Common Substring*) cũng là một bài toán nổi tiếng khác liên quan tới các thuật toán xử lý xâu.

Để tính $f(i,j)$, cần phân tích LCS của hai xâu $a[i \dots m)$ và $b[j \dots n)$ được xây dựng như thế nào... Để minh bạch các ký hiệu sau này, ta coi như a_i là tham chiếu tới phần tử thứ i trong xâu A và b_j là tham chiếu tới phần tử thứ j trong xâu B. Về giá trị thì a_i và b_j là những ký tự nhưng khi nói chúng được chọn/không được chọn tức là nói đến phần tử tương ứng chứ không phải nói về giá trị của phần tử đó.

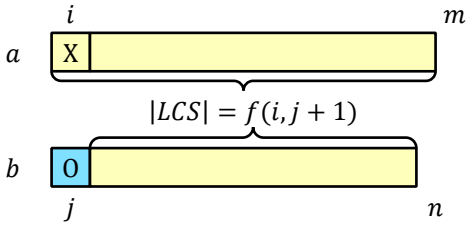
- Nếu $a_i = b_j$, LCS của $a[i \dots m)$ và $b[j \dots n)$ sẽ được tạo thành bằng cách lấy ký tự a_i (cũng là b_j) nối vào đầu LCS của $a[i+1 \dots m)$ và $b[j+1 \dots n)$. Trong trường hợp này $f(i,j) = f(i+1, j+1) + 1$.



- Nếu $a_i \neq b_j$, xét LCS của $a[i \dots m)$ và $b[j \dots n)$, ký tự đầu tiên của LCS (nếu có) hoặc là khác với a_i hoặc là khác với b_i:
  - LCS không bắt đầu bởi ký tự bằng a_i, khi đó ta có thể bỏ đi phần tử a_i, vấn đề quy về tìm LCS của $a[i+1 \dots m)$ và $b[j \dots n)$, tức là $f(i,j) = f(i+1, j)$.



  - LCS không bắt đầu bởi ký tự bằng b_i, khi đó ta có thể bỏ đi phần tử b_j, vấn đề quy về tìm LCS của $a[i \dots m)$ và $b[j+1 \dots n)$, tức là $f(i,j) = f(i, j+1)$.



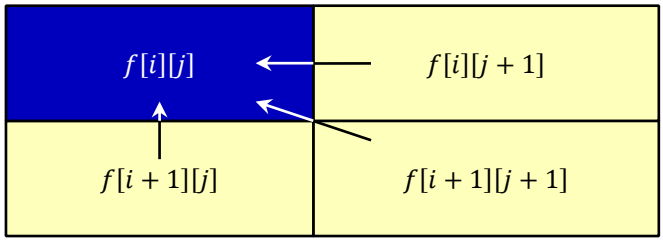
  (Trường hợp ký tự đầu tiên của LCS không bằng cả a_i cũng như b_j, ta coi như một trong hai trường hợp trên). Vậy $f(i,j) = \max\{f(i+1,j), f(i,j+1)\}$ vì ta muốn cực đại hóa $f(i,j)$ theo mọi trường hợp có thể.

Tổng hợp lại, ta có công thức truy hồi:

$$f(i,j) = \begin{cases} f(i+1,j+1) + 1, \text{nếu } a_i = b_j \\ \max\{f(i+1,j), f(i,j+1)\}, \text{nếu } a_i \neq b_j \end{cases} \quad (1.5)$$

Đồng nhất hàm mục tiêu f với bảng hai chiều f: $f(i,j) \equiv f[i][j]$. Công thức truy hồi (1.5) cho thấy mỗi phần tử $f[i][j]$ được tính dựa vào phần tử kề góc phải dưới ($f[i+1][j+1]$),

phần tử ở hàng dưới, cùng cột ($f[i+1][j]$) và phần tử cùng hàng, kề bên phải ($f[i][j+1]$) (Hình 1-5).



Hình 1-5. Sơ đồ tính công thức truy hồi

Thứ tự hợp lý để tính bảng f là tính theo từng hàng từ dưới lên và trên mỗi hàng thì tính từ phải qua trái (cũng có thể tính theo từng cột từ phải qua trái và trên mỗi cột thì tính từ dưới lên). Thứ tự đó cũng cho thấy để khởi động quá trình tính toán, ta phải biết được giá trị các phần tử nằm ở cột cuối và hàng cuối của bảng. Đây là cơ sở quy hoạch động.

Ta sẽ sử dụng bảng f với $m+1$ hàng đánh số từ 0 tới m và $n+1$ cột đánh số từ 0 tới n. Hàng cuối bảng f gồm các phần tử dạng $f[m][j]$ theo định nghĩa là độ dài LCS giữa $a[m \dots m)$ (xâu rỗng) và $b[j \dots n)$ nên $f[m][j] = 0$ $(\forall j: 0 \le j \le n)$. Cột cuối bảng f gồm các phần tử dạng $f[i][n]$ theo định nghĩa là độ dài LCS giữa $a[i \dots m)$ và $b[n \dots n)$ (xâu rỗng) nên $f[i][n] = 0$ $(\forall i: 0 \le i \le m)$.

			A	C	G	G	T	A	G	
	f	0	1	2	3	4	5	6	7
C	0	4	4	3	3	3	2	1	0
C	1	4	4	3	3	3	2	1	0
T	2	3	3	3	3	3	2	1	0
A	3	3	2	2	2	2	2	1	0
A	4	2	2	2	2	2	2	1	0
G	5	1	1	1	1	1	1	1	0
	6	0	0	0	0	0	0	0	0

Hình 1-6. Giải công thức truy hồi và truy vết tìm LCS

Sau khi tính xong bảng f, phép truy vết lần ngược lại quá trình tính toán để tìm LCS. Bắt đầu với $i = j = 0$, ta truy từ $f[i][j]$ để tìm LCS của $a[i \dots m)$ và $b[j \dots n)$:

- Nếu $a_i = b_j$, ta chọn luôn ký tự $a_i = b_j$ và truy tiếp $f[i+1][j+1]$ để tìm LCS của $a[i+1 \dots m)$ và $b[j+1 \dots n)$
- Nếu $a_i \ne b_j$:
  - Nếu $f[i][j] = f[i+1][j]$, chắc chắn a_i không được chọn vào LCS, ta truy tiếp $f[i+1][j]$ để tìm LCS của $a[i+1 \dots m)$ và $b[j \dots n)$
  - Nếu không, chắc chắn b_j không được chọn vào LCS, ta truy tiếp $f[i][j+1]$ để tìm LCS của $a[i \dots m)$ và $b[j+1 \dots n)$.

Phép truy vết kết thúc khi ta truy đến cơ sở quy hoạch động ($i = m$ hoặc $j = n$), ta tìm được LCS của hai xâu A và B (Hình 1-6).

❋ Cài đặt

LCS_DP.cpp Tìm dãy con chung dài nhất

```cpp
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;
const int maxMN = 1000;
string a, b;
int m, n, f[maxMN + 1][maxMN + 1];

void ReadInput() //Nhập dữ liệu
{
    getline(cin, a);
    getline(cin, b);
    m = a.length();
    n = b.length();
}

void Init() //Khởi tạo cơ sở quy hoạch động: hàng m và cột n bảng f
{
    for (int i = 0; i <= m; ++i) f[i][n] = 0;
    for (int j = 0; j <= n; ++j) f[m][j] = 0;
}

void SolveRecurrence() //Giải công thức truy hồi
{
    for (int i = m - 1; i >= 0; --i) //Tính từ hàng dưới lên hàng trên
        for (int j = n - 1; j >= 0; --j) //Trên mỗi hàng tính từ phải qua trái
            if (a[i] == b[j])
                f[i][j] = f[i + 1][j + 1] + 1;
            else
                f[i][j] = max(f[i + 1][j], f[i][j + 1]);
}

void Print() //Truy vết tìm LCS
{
    int i = 0, j = 0;
    while (i < m && j < n) //Chưa truy đến cơ sở
        if (a[i] == b[j])
        {
            cout << a[i]; //Chọn luôn a[i]
            ++i; ++j; //Truy tiếp f[i + 1][j + 1]
        }
        else //truy tiếp f[i + 1][j] hoặc f[i][j + 1] tùy theo giá trị nào lớn hơn
            if (f[i][j] == f[i + 1][j]) ++i;
            else ++j;
}

int main()
{
    ReadInput();
    Init();
    SolveRecurrence();
    Print();
}
```

Bài toán tìm dãy ngắn nhất nhận cả hai xâu A, B làm dãy con (*Shortest Common Supersequence – SCS*) có thể giải bằng cách đặt lại công thức truy hồi và hàm mục tiêu: $f(i, j)$ là độ dài SCS của $a[i \dots m)$ và $b[j \dots n)$.

Tuy nhiên cũng có thể giải bài toán SCS của A và B bằng cách tìm LCS của A và B rồi chèn thêm các ký tự trong A cũng như trong B mà không được chọn vào LCS. Những vấn đề này ta coi như bài tập.

1.3.3. Bài toán xếp ba lô

Cho n sản phẩm đánh số từ 0 tới $n-1$, sản phẩm thứ i có trọng lượng là w_i và giá trị là v_i ($w_i, v_i \in \mathbb{Z}_{>0}$). Cho một balô có giới hạn trọng lượng là m, hãy chọn ra một số sản phẩm cho vào balô sao cho tổng trọng lượng của chúng không vượt quá m và tổng giá trị của chúng là lớn nhất có thể.

Input

- Dòng 1 chứa hai số nguyên dương n, m ($n \leq 1000; m \leq 10000$)
- n dòng tiếp theo, mỗi dòng chứa hai số nguyên dương lần lượt là trọng lượng và giá trị một sản phẩm.

Output

Phương án chọn các sản phẩm có tổng trọng lượng $\leq m$ và tổng giá trị lớn nhất có thể.

Sample Input	Sample Output
5 11 3 30 4 40 5 40 5 60 4 40	Selected Objects: Object 4: Weight = 4; Value = 40 Object 1: Weight = 4; Value = 40 Object 0: Weight = 3; Value = 30 Total Weight: 11 Total Value: 110

Thuật toán

Bài toán Knapsack tuy chưa có thuật toán giải hiệu quả trong trường hợp tổng quát nhưng với ràng buộc dữ liệu cụ thể như ở bài này thì có thuật toán quy hoạch động khá hiệu quả để giải, dưới đây chúng ta sẽ trình bày cách làm đó.

Gọi $f(i,j)$ là giá trị lớn nhất có thể có bằng cách chọn trong các sản phẩm $\{0,1,\dots,i\}$ với giới hạn trọng lượng j. Mục đích của chúng ta là đi tìm $f(n-1,m)$: Giá trị lớn nhất có thể có bằng cách chọn trong n sản phẩm đã cho với giới hạn trọng lượng m.

Với giới hạn trọng lượng j, cách chọn trong số các sản phẩm $\{0,1,\dots,i\}$ để có giá trị lớn nhất sẽ rơi vào một trong hai khả năng:

- Nếu không chọn sản phẩm thứ i, thì $f(i,j)$ là giá trị lớn nhất có thể có bằng cách chọn trong số các sản phẩm $\{1,2,\dots,i-1\}$ với giới hạn trọng lượng bằng j. Tức là $f(i,j) = f(i-1,j)$
- Nếu có chọn sản phẩm thứ i (tất nhiên chỉ xét tới trường hợp này khi mà $w_i \leq j$), thì $f(i,j)$ bằng giá trị sản phẩm thứ i cộng với giá trị lớn nhất có thể có được bằng cách chọn trong số các sản phẩm $\{1,2,\dots,i-1\}$ với giới hạn trọng lượng $j-w_i$. Tức là về mặt giá trị thu được, $f(i,j) = f(i-1,j-w_i) + v_i$

Theo cách xây dựng $f(i,j)$, ta suy ra công thức truy hồi:

$$f(i,j) = \begin{cases} f(i-1,j), \text{nếu } j < w_i \\ \max\{f(i-1,j), f(i-1,j-w_i)+v_i\}, \text{nếu } j \geq w_i \end{cases} \quad (1.6)$$

Công thức truy hồi (1.6) cho thấy các phần tử $f(i,.)$ được tính nhờ các phần tử $f(i-1,.)$. Vì vậy, ta sẽ đặt cơ sở quy hoạch động ở các giá trị $f(-1,.)$ để từ đó giải công thức truy hồi tính $f(n-1,m)$. Từ định nghĩa, $f(-1,$ j) giá trị lớn nhất chọn trong các sản phẩm từ 0 tới -1 (tập rỗng), nên hiển nhiên $f(-1,j)=0$ bất kể j bằng bao nhiêu.

Tính xong bảng phương án thì ta quan tâm đến $f(n-1,m)$, đó chính là giá trị lớn nhất thu được khi chọn trong cả n sản phẩm với giới hạn trọng lượng m. Việc còn lại là chỉ ra phương án chọn các sản phẩm. Bắt đầu với $i=n-1$ và $j=m$:

- Nếu $f(i,j)=f(i-1,j)$ thì phương án tối ưu không chọn sản phẩm i, ta truy tiếp $f(i-1,j)$.
- Còn nếu $f(i,j)\neq f(i-1,j)$ thì ta thông báo rằng phép chọn tối ưu có chọn sản phẩm i và truy tiếp $f(i-1,j-w_i)$.

Cứ tiếp tục cho tới khi truy tới $f(-1,.)$ là cơ sở quy hoạch động

Cài đặt

Ta sẽ giải công thức truy hồi theo kiểu từ trên xuống sử dụng một hàm đệ quy f kết hợp với một bảng ghi nhớ $g[0\ldots n)[0\ldots m]$. Hàm $f(i,j)$ sẽ trả về giá trị $gf[i][j]$ nếu giá trị này đã được tính, ngược lại hàm sẽ tính $f(i,j)$ bằng công thức truy hồi, lưu lại kết quả vào $g[i][j]$ rồi trả về giá trị g$[i][j]$.

KNAPSACK_DP.cpp Bài toán xếp ba lô

```cpp
#include <iostream>
#include <algorithm>
using namespace std;
const int maxN = 1e3;
const int maxM = 1e4;

int n, m, w[maxN], v[maxN];
int g[maxN][maxM + 1]; //Bảng ghi nhớ

void ReadInput() //Nhập dữ liệu
{
    cin >> n >> m;
    for (int i = 0; i < n; ++i)
        cin >> w[i] >> v[i];
    fill(g[0], g[n], -1); //Các phần tử bảng g đều chưa biết (-1)
}

int f(int i, int j) //Giải công thức truy hồi từ trên xuống
{
    if (i < 0) return 0; //Cơ sở quy hoạch động được đặt ở f(-1, .) = 0
    if (g[i][j] == -1) //g[i][j] chưa được tính, tính g[i][j] bằng đệ quy
        g[i][j] = j < w[i] ?
                  f(i - 1, j) : max(f(i - 1, j), f(i - 1, j - w[i]) + v[i]);
    return g[i][j]; //Trả về kết quả hàm là g[i][j]
}

```

```
void Print() //In kết quả
{
    int TotalW = 0;
    cout << "Selected Objects: \n";
    int j = m;
    for (int i = n - 1; i >= 0; --i)
        if (f(i, j) != f(i - 1, j)) //Có chọn sản phẩm i
        {
            cout << "Object " << i << ": "
                 << "Weight = " << w[i] << "; "
                 << "Value = " << v[i] << "\n";
            TotalW += w[i];
            j -= w[i];
        }
    cout << "Total Weight: " << TotalW << "\n"
         << "Total Value: " << f(n - 1, m);
}

int main()
{
    ReadInput();
    Print();
}
```

Thuật toán có độ phức tạp O(nm). Công thức truy hồi trong bài này hoàn toàn có thể giải từ dưới lên, tuy vậy phép giải công thức truy hồi từ trên xuống tỏ ra hiệu quả hơn trên thực tế. Ta sẽ phân tích qua một trường hợp cụ thể chính là ví dụ của bài, với dữ liệu như vậy, bảng g sau khi tính $f(4,11)$ sẽ là:

$$n = 5; m = 11;$$
$$w[0 \dots 4] = (3,4,5,5,4)$$
$$v[0 \dots 4] = (30,40,40,60,40)$$

	g	0	1	2	3	4	5	6	7	8	9	10	11
✓	0	-1	0	0	30	-1	-1	30	30	-1	-1	-1	30
✓	1	-1	0	0	-1	-1	-1	40	70	-1	-1	-1	70
	2	-1	-1	0	-1	-1	-1	40	70	-1	-1	-1	80
	3	-1	-1	-1	-1	-1	-1	-1	70	-1	-1	-1	100
✓	4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	110

Ta thấy còn rất nhiều giá trị -1 trong bảng, nghĩa là có rất nhiều phần tử trong bảng g không tham gia vào quá trình tính $f(4,11)$. Phép giải công thức truy hồi từ trên xuống hiệu quả hơn phép giải từ dưới lên trong dữ liệu cụ thể này do không phải tính toàn bộ bảng g.

Việc giải công thức truy hồi từ trên xuống cũng thuận lợi cho ta đặt cơ sở quy hoạch động tại $f(-1,.)$. Điều này giúp cho chương trình cài đặt đơn giản hơn, khi mà trong C++ mảng được đánh số từ 0, muốn thay đổi thành mảng đánh số từ -1 sẽ cần thêm một phép ánh xạ chỉ số.

1.3.4. Đối xứng hóa

Một xâu ký tự được gọi là xâu đối xứng (*Palindrome*) nếu nó không đổi khi ta viết các ký tự của nó theo thứ tự ngược lại. Ví dụ như các xâu "civic", "level", "racecar" là các xâu đối xứng.

Cho trước một xâu ký tự S, hãy chèn thêm một số ít nhất các ký tự vào trong xâu S để được một xâu T là xâu đối xứng.

Input

Gồm một dòng chứa xâu S dài không quá 1000 ký tự.

Output

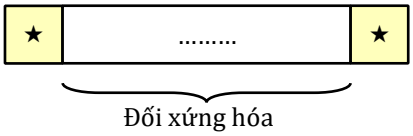
Ghi ra xâu T tìm được

Sample Input	Sample Output
AMREROM	AMOREROMA

❋ **Phân tích**

Để chèn các ký tự nhằm đối xứng hóa xâu S, ta cần tìm xâu T đối xứng và nhận xâu S làm dãy con. Nhận xét:

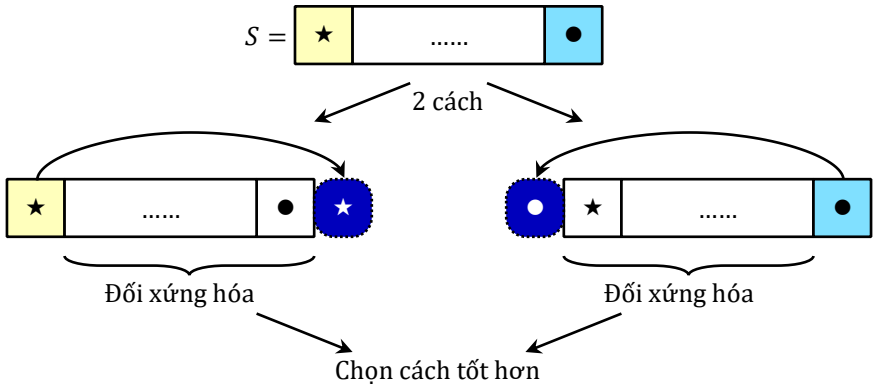
Nếu ký tự đầu và ký tự cuối của xâu S bằng nhau, ta chỉ cần chèn các ký tự để đối xứng hóa đoạn giữa trong xâu S từ vị trí sau ký tự đầu đến trước ký tự cuối:



Nếu ký tự đầu và ký tự cuối của xâu S khác nhau, vì xâu T cần tìm phải có ký tự đầu và ký tự cuối giống nhau, giá trị ký tự này hoặc phải bằng ký tự đầu của S hoặc phải bằng ký tự cuối của S (nếu không thì ta có thể loại bỏ cặp ký tự đầu và cuối của xâu T mà vẫn thu được xâu đối xứng ngắn hơn nhận S làm dãy con). Từ đó suy ra hai cách thức có thể dùng để đối xứng hóa xâu S:

- Hoặc chèn ký tự đầu của S xuống cuối xâu
- Hoặc chèn ký tự cuối của S lên đầu xâu

Sau đó thì ký tự đầu và ký tự cuối của xâu S sẽ bằng nhau, ta chỉ cần đối xứng hóa đoạn giữa trong xâu S từ vị trí sau ký tự đầu đến trước ký tự cuối. Trong trường hợp này có hai cách thức để đối xứng hóa xâu S, cách nào tốt hơn sẽ được lựa chọn:



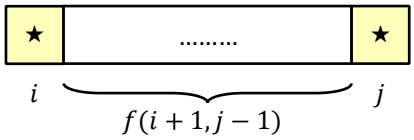
Việc đối xứng hóa một xâu dài quy về việc đối xứng hóa một xâu con ngắn hơn của nó. Ta dựa vào các quan sát này xây dựng hàm mục tiêu và công thức truy hồi.

❋ **Thuật toán**

Đánh số các ký tự trong xâu S từ 0 trở đi: $S = s[0 \dots n)$.

Xét xâu con $s[i \dots j]$. Gọi $f(i,j)$ là số ký tự ít nhất cần chèn vào để biến xâu con $s[i \dots j]$ thành xâu đối xứng, khi đó việc đối xứng hóa $s[i \dots j]$ được thực hiện như sau:

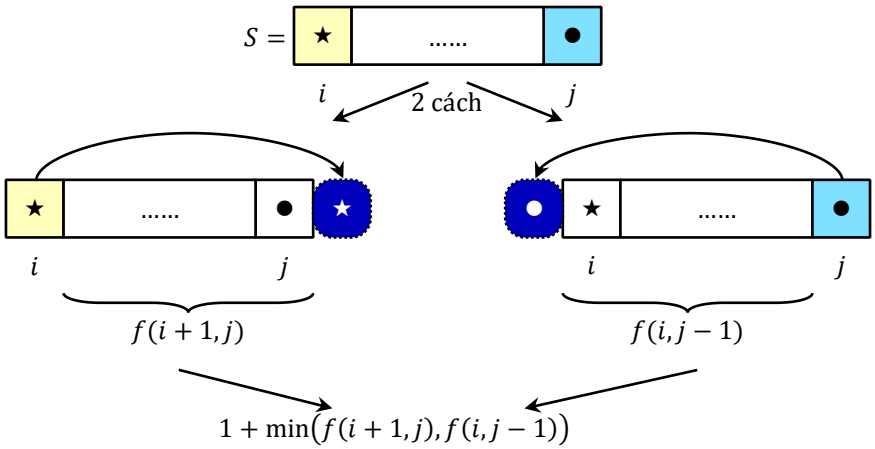
Nếu ký tự đầu và ký tự cuối bằng nhau: $s[i] = s[j]$, ta chỉ cần đối xứng hóa đoạn giữa: $s[i+1 \dots j-1]$. Tức là $f(i,j) = f(i+1, j-1)$:



Nếu ký tự đầu và ký tự cuối khác nhau: $s[i] \neq s[j]$, ta chọn cách làm tốt hơn trong hai cách:

- Hoặc chèn ký tự $s[i]$ xuống cuối rồi đối xứng hóa đoạn $s[i+1 \dots j]$: Tổng số ký tự cần chèn bằng $1 + f(i+1, j)$
- Hoặc chèn ký tự $s[j]$ lên đầu rồi đối xứng hóa đoạn $s[i \dots j-1]$: Tổng số ký tự cần chèn bằng $1 + f(i, j-1)$

Vậy trong trường hợp này, $f(i,j) = 1 + \min\big(f(i+1,j), f(i,j-1)\big)$:



Tóm lại:

$$f(i,j) = \begin{cases} f(i+1, j-1), \text{nếu } s_i = s_j \\ 1 + \min\big(f(i+1,j), f(i,j-1)\big), \text{nếu } s_i \neq s_j \end{cases}$$

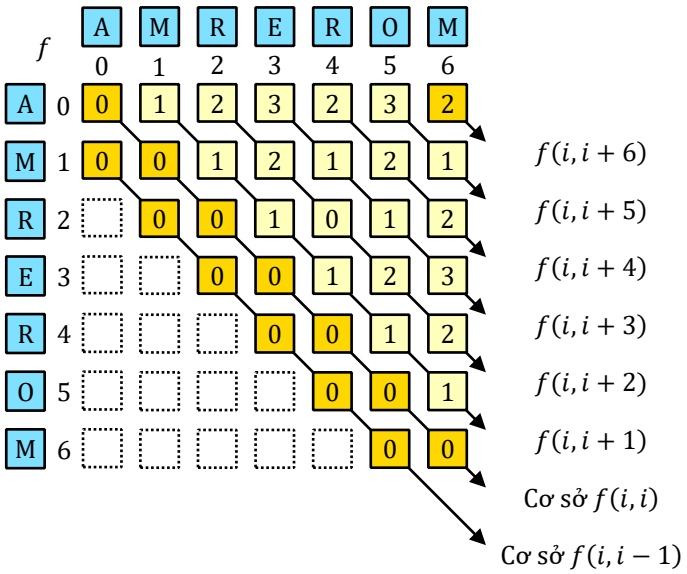
Công thức truy hồi đưa việc đối xứng hóa một xâu con dài về việc đối xứng hóa một xâu con ngắn hơn (ngắn hơn 1 hoặc 2 ký tự). Cơ sở quy hoạch động sẽ được đặt ở những xâu con có độ dài 0 hoặc độ dài 1, những xâu này hiển nhiên là đối xứng và không cần chèn thêm bất kỳ ký tự nào để đối xứng hóa chúng, cụ thể là với $\forall i$:

- $f(i, i-1) = 0$ vì xâu $s[i \dots i-1]$ là xâu rỗng
- $f(i,i) = 0$ vì xâu $s[i \dots i]$ chỉ có một ký tự.

Phép giải công thức truy hồi có thể thực hiện theo thứ tự từ các xâu con ngắn tới các xâu con dài:

```
for (len = 1; len < n; ++len) //Tính hàm f trên các xâu con độ dài len + 1
    for (i = 0; i + len < n; ++i) //Xét mọi vị trí đầu xâu con
    {
        j = i + len; //j là vị trí đứng cuối xâu con độ dài len + 1
        «Tính f[i][j]»;
    }
```

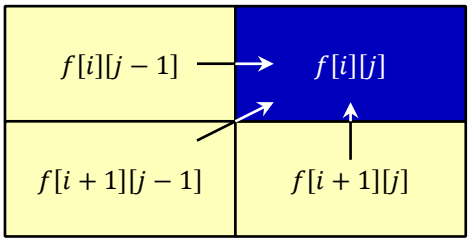
Sơ đồ tính toán trên được chỉ ra trong Hình 1-7



Hình 1-7. Giải công thức truy hồi tìm cách đối xứng hóa xâu

Phép giải công thức truy hồi theo thứ tự từ xâu con ngắn tới xâu con dài khá trực quan, trong khá nhiều bài toán mà phương án tối ưu trên một xâu được quy về phương án tối ưu trên các xâu con, phương pháp cài đặt này còn cho phép chứng minh thuật toán dễ dàng, đồng thời có thể đưa vào vài kỹ thuật tối ưu nhằm giảm độ phức tạp tính toán.

Tuy vậy đối với bài toán cụ thể trong mục này, chúng ta có thứ tự tính đơn giản hơn dựa vào sơ đồ tính công thức truy hồi. Mỗi phần tử $f[i][j]$ phải được tính sau phần tử ở hàng dưới, cột bên trái: $(f[i+1][j-1])$, phần tử hàng dưới, cùng cột: $(f[i+1][j])$ và phần tử cùng hàng, cột bên trái: $(f[i][j-1])$ (Hình 1-8):



Hình 1-8. Sơ đồ tính công thức truy hồi

Như vậy ta có thể tính bảng f theo thứ tự: Tính theo từng hàng từ dưới lên và trên mỗi hàng thì tính từ trái qua phải. Cách tính này đơn giản hơn, có tốc độ nhanh hơn nhờ việc truy cập các phần tử nằm sát nhau trong bộ nhớ, tăng hiệu quả của bộ nhớ cache.

```
for (i = n - 1; i >= 0; --i)
    for (j = i + 1; j < n; ++j)
        «Tính f[i][j] »;
```

Việc truy vết tìm xâu đối xứng hóa của S được thực hiện bằng cách dò ngược lại quá trình xác định $f[0][n-1]$ (đây là số ký tự ít nhất cần chèn để biến xâu S thành đối xứng). Tuy nhiên trong trường hợp này thuật toán lặp tỏ ra khá cồng kềnh, ta sử dụng thuật toán đệ quy để in kết quả: Viết một hàm `Print(i, j)` để in ra xâu đối xứng hóa của $s[i \dots j]$:

- Nếu $i > j$: Xâu $s[i \dots j]$ là xâu rỗng, ta không cần in gì cả. Thoát
- Nếu $i = j$: Xâu $s[i \dots j]$ chỉ gồm 1 ký tự, ta in ra ký tự duy nhất đó và thoát.
- Nếu $i < j$: Xâu $s[i \dots j]$ có ít nhất 2 ký tự:
  - Nếu $s_i = s_j$, sơ đồ in kết quả sẽ là:

    s[i] `Print(i + 1, j - 1)` s[j]
  - Nếu $s_i \neq s_j$, ta đi theo phương án lựa chọn tốt hơn theo hai bài toán con:
    - Nếu $f[i+1][j] < f[i][j-1]$, sơ đồ in kết quả sẽ là:

      s[i] `Print(i + 1, j)` s[i]
    - Còn nếu $f[i+1][j] \geq f[i][j-1]$, sơ đồ in kết quả sẽ là:

      s[j] `Print(i, j - 1)` s[j]

PALINDROME_DP.cpp Đối xứng hóa một xâu

```cpp
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;
const int maxN = 1000;
string s;
int n;
int f[maxN][maxN];

void ReadInput() //Nhập dữ liệu
{
    getline(cin, s);
    n = s.length();
}

void Init() //Khởi tạo cơ sở quy hoạch động
{
    f[0][0] = 0;
    for (int i = 1; i < n; ++i)
        f[i][i - 1] = f[i][i] = 0;
}

void SolveRecurrence() //Giải công thức truy hồi
{
    for (int i = n - 1; i >= 0; --i) //Tính theo từng hàng từ dưới lên
        for (int j = i; j < n; ++j) //Trên mỗi hàng tính từ trái qua phải
            f[i][j] = s[i] == s[j] ?
                      f[i + 1][j - 1] : 1 + min(f[i + 1][j], f[i][j - 1]);
}

```

```
void Print(int i, int j) //In ra xâu đối xứng của xâu con s[i...j]
{
    if (i > j) return; //Xâu s[i...j] rỗng, thoát
    if (i == j) //Xâu s[i...j] chỉ có 1 ký tự
    {
        cout << s[i]; //In ra ký tự duy nhất
        return; //Thoát
    }
    if (s[i] == s[j]) //Ký tự đầu s[i] bằng ký tự cuối s[j]
    {
        cout << s[i]; Print(i + 1, j - 1); cout << s[j];
        return; //Thoát
    }
    if (f[i + 1][j] < f[i][j - 1])
    {   //Chèn s[i] vào cuối xâu con và đối xứng hóa đoạn giữa
        cout << s[i]; Print(i + 1, j); cout << s[i];
    }
    else
    {   //Chèn s[j] vào đầu xâu con và đối xứng hóa đoạn giữa
        cout << s[j]; Print(i, j - 1); cout << s[j];
    }
 }

int main()
{
    ReadInput();
    Init();
    SolveRecurrence();
    Print(0, n - 1);
}
```

Nhìn vào sơ đồ tính công thức truy hồi, bài toán đối xứng hóa một xâu khá giống với bài toán LCS (mục 1.3.2). Trên thực tế, bài toán đối xứng hóa một xâu có thể quy về bài toán tìm LCS của một xâu với xâu đảo của nó (viết các ký tự theo thứ tự ngược lại). Vấn đề này chúng ta coi như bài tập.

1.3.5. Phân hoạch tam giác

Trên mặt phẳng với hệ tọa độ trực chuẩn Oxy, cho một đa giác lồi n đỉnh với các đỉnh đánh số từ 0 tới $n-1$ theo đúng thứ tự tạo thành đa giác. Một bộ $n-3$ đường chéo đôi một không cắt nhau sẽ chia đa giác đã cho thành $n-2$ tam giác, ta gọi bộ $n-3$ đường chéo đó là một phép tam giác phân của đa giác lồi ban đầu.

Tổng độ dài các đường chéo trong một phép tam giác phân gọi là trọng số của phép tam giác phân đó. Bài toán đặt ra là hãy tìm một phép tam giác phân có trọng số nhỏ nhất.

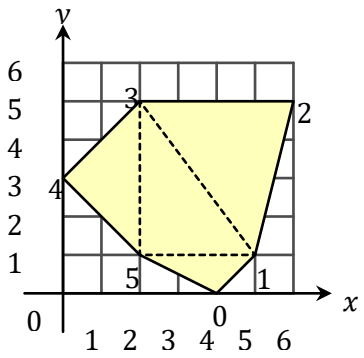
Input

- Dòng 1 chứa số n là số đỉnh của đa giác ($3 \leq n \leq 200$)
- n dòng tiếp theo, dòng thứ chứa toạ độ một đỉnh theo đúng thứ tự tạo thành đa giác. Tọa độ đỉnh gồm hai số thực là hoành độ và tung độ của đỉnh đó.

Output

Phép tam giác phân nhỏ nhất.

Sample Input	Sample Output
6 4 0 5 1 6 5 2 5 0 3 2 1	Minimum triangulation: 12 1-5: 3 1-3: 5 3-5: 4



❋ **Thuật toán**

Đồng nhất mỗi đỉnh với chỉ số của nó. Gọi (x_i, y_i) là tọa độ của điểm i. Ký hiệu $d(i,j)$ là độ dài đường chéo nối điểm i và điểm j $(i < j)$:

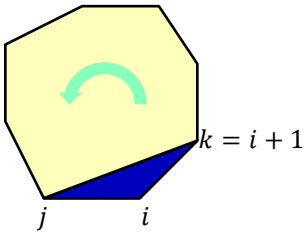
$$d(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Với $j - i \geq 3$, gọi $f(i,j)$ là trọng số phép tam giác phân nhỏ nhất để chia đa giác $i \dots j$ (đa giác với ≥ 4 đỉnh xác định bởi các đỉnh từ i tới j), khi đó $f(0, n-1)$ chính là trọng số phép tam giác phân nhỏ nhất để chia đa giác ban đầu.

Bằng quan sát hình học của một phép tam giác phân trên đa giác $i \dots j$, ta nhận thấy rằng trong bất kỳ phép tam giác phân nào, luôn tồn tại một và chỉ một tam giác chứa cạnh (i,j). Tam giác chứa cạnh (i,j) sẽ có một đỉnh là i, một đỉnh là j và một đỉnh k nào đó $(i < k < j)$. Trong phép tam giác phân đa giác $i \dots j$ có trọng số nhỏ nhất, ta quan tâm tới tam giác chứa cạnh (i,j) gọi là Δijk, có ba khả năng cho việc lựa chọn đỉnh k:

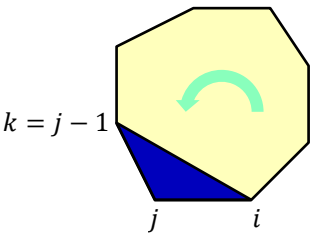
Khả năng thứ nhất: đỉnh $k = i + 1$, khi đó đường chéo $(i+1, j)$ sẽ thuộc phép tam giác phân đang xét, những đường chéo còn lại sẽ phải tạo thành một phép tam giác phân nhỏ nhất để chia đa giác lồi $i + 1 \dots j$. Tức là trong trường hợp này:

$$f(i,j) = d(i+1,j) + f(i+1,j)$$



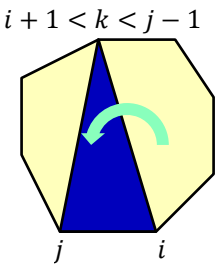
Khả năng thứ hai: đỉnh $k = j - 1$, khi đó đường chéo $(i, j-1)$ sẽ thuộc phép tam giác phân đang xét, những đường chéo còn lại sẽ phải tạo thành một phép tam giác phân nhỏ nhất để chia đa giác lồi $i \dots j - 1$. Tức là trong trường hợp này:

$$f(i,j) = d(i,j-1) + f(i,j-1)$$



Khả năng thứ ba: đỉnh k không trùng với $i+1$ và cũng không trùng với $j-1$, khi đó cả hai đường chéo (i,k) và (k,j) đều thuộc phép tam giác phân đang xét, những đường chéo còn lại sẽ tạo thành hai phép tam giác phân nhỏ nhất để chia từng đa giác $i \dots k$ và $k \dots j$. Tức là trong trường hợp này:

$$f(i,j) = d(i,k) + d(k,j) + f(i,k) + f(k,j)$$



Vì ta muốn cực tiểu hóa $f(i,j)$ nên điểm k sẽ được chọn $\in [i+1; j-1]$ để có được $f(i,j)$ nhỏ nhất theo các khả năng trên:

$$f(i,j) = \min \begin{cases} d(i+1,j) + f(i+1,j) \\ d(i,j-1) + f(i,j-1) \\ \min\limits_{\substack{\forall k: \\ i+1<k<j-1}} \{d(i,k) + d(k,j) + f(i,k) + f(k,j)\} \end{cases} \quad (1.7)$$

Công thức (1.7) có thể rút gọn nếu ta đưa vào hai quy ước: $\forall i < j$:

- Nếu (i,j) không phải là đường chéo đa giác, tức là (i,j) là cạnh đa giác ($j = i+1$), ta coi như $d(i,j) = 0$
- Nếu $j - i \le 2$, ta coi như $f[i][j] = 0$, thực ra chỉ có trường hợp $j - i = 1$ phải quy ước, trường hợp $j - i = 2$ thì đa giác $i \dots j$ là một tam giác và theo đúng định nghĩa hàm mục tiêu, $f(i, i+2) = 0$ do ta không cần tam giác phân một tam giác.

Những quy ước này cho phép ta viết gọn lại công thức truy hồi:

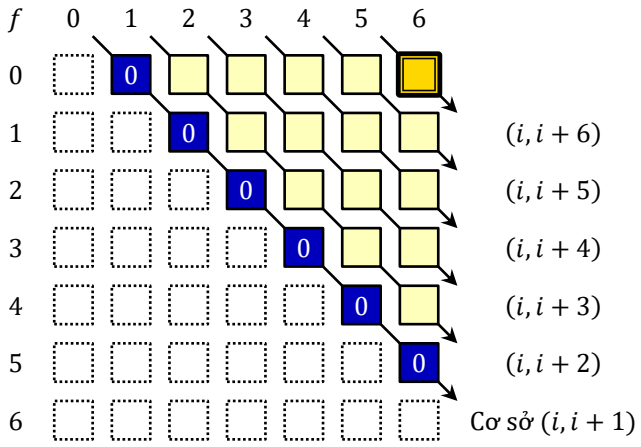
$$f(i,j) = \min_{\substack{\forall k \\ i<k<j}} \{d(i,k) + d(k,j) + f(i,k) + f(k,j)\} \quad (1.8)$$

Thêm một nhận xét nữa là mặc dù công thức truy hồi (1.8) được xây dựng tính $f(i,j)$ với $j - i \ge 3$ nhưng với quy ước ở trên, công thức này cũng đúng luôn cả với trường hợp $j - i = 2$ (đa giác $i \dots j$ là một tam giác)

Vì $f(i,j)$ được tính qua các giá trị $f(i,k)$ và $f(k,j)$ với $i < k < j$, suy ra hiệu số $j - i$ quyết định thứ tự nhỏ hơn/lớn hơn của các bài toán con. Khi đồng nhất hàm mục tiêu f với mảng hai chiều f, các giá trị $f[i][j]$ với $j - i$ lớn hơn sẽ được tính sau các giá trị $f[i'][j']$ có hiệu số $j' - i'$ nhỏ hơn. Thứ tự này cho phép xác định cơ sở quy hoạch động: Các giá trị $f[i][i+1]$ được đặt bằng 0 chính là cơ sở quy hoạch động

Cài đặt

Công thức truy hồi trong bài toán này có thể giải dễ dàng theo kiểu tính từ trên xuống bằng đệ quy kết hợp với một bảng ghi nhớ. Tuy vậy ta muốn thực hiện một lời giải ngắn hơn dùng giải thuật lặp: Đầu tiên bảng phương án f được điền cơ sở quy hoạch động: Các phần tử trên đường chéo $f[i][i+1]$ được đặt bằng 0, dùng công thức truy hồi tính tiếp các phần tử trên đường chéo $f[i][i+2]$, rồi tính tiếp đường chéo $f[i][i+3]$... Cứ như vậy cho tới khi tính được phần tử $f[0][n-1]$ (Hình 1-9).



Hình 1-9. Quy trình tính bảng phương án với $n = 7$

Việc chỉ ra phép tam giác phân cũng có thể thực hiện chỉ dựa vào bảng f. Tuy vậy trong bài toán này, ta sẽ sử dụng một cơ chế lưu lại vết để có thể dễ dàng truy vết tìm lại phương án tối ưu. Cơ chế lưu vết thực hiện cùng quá trình tính bảng f: Tại mỗi bước tính

$$f[i][j] = \min_{\forall k: i<k<j} \{d(i,k) + d(k,j) + f[i][k] + f[k][j]\}$$

ta lưu lại $trace[i][j]$ là chỉ số k mà tại đó $d(i,k) + d(k,j) + f[i][k] + f[k][j]$ đạt cực tiểu và được gán cho $f[i][j]$:

$$trace[i][j] = \arg \min_{\forall k: i<k<j} \{d(i,k) + d(k,j) + f[i][k] + f[k][j]\}$$

Sau khi quá trình tính toán kết thúc, để in ra phép tam giác phân nhỏ nhất đối với đa giác $i \dots j$, ta sẽ dựa vào phần tử $k = trace[i][j]$ để đưa vấn đề về việc in ra phép tam giác phân đối với hai đa giác $i \dots k$ và $k \dots j$. Điều này có thể được thực hiện đơn giản bằng một hàm đệ quy.

TRIANGULATION.cpp Phân hoạch tam giác

```
#include <iostream>
#include <cmath>
using namespace std;
const int maxN = 200;
int n;
double x[maxN], y[maxN];
double f[maxN][maxN];
int trace[maxN][maxN];

void ReadInput() //Nhập dữ liệu về đa giác
{
    cin >> n;
    for (int i = 0; i < n; ++i)
        cin >> x[i] >> y[i];
}

double d(int i, int j) //Độ dài đường chéo i - j, = 0 nếu i - j không phải đường chéo
{
    if (i + 1 == j) return 0;
    double dx = x[i] - x[j], dy = y[i] - y[j];
    return sqrt(dx * dx + dy * dy);
}

```

```
void SolveRecurrence() //Quy hoạch động
{
    //Khởi tạo cơ sở quy hoạch động f[i][i + 1] bằng 0, ∀i
    for (int i = 0; i + 1 < n; ++i)
        f[i][i + 1] = 0;
    //Giải công thức truy hồi từ dưới lên
    for (int t = 2; t < n; ++t) //Tính các f[i][i + t] theo t tăng dần
        for (int i = 0; i + t < n; ++i)
        {
            int j = i + t;
            f[i][j] = -1; //f[i][j] < 0 tức là chưa tính.
            for (int k = i + 1; k < j; ++k) //Xét ∀k = i + 1 … j - 1
            {
                //q = trọng số phép tam giác phân i…j nếu có chọn tam giác (i, j, k)
                double q = d(i, k) + d(k, j) + f[i][k] + f[k][j];
                if (f[i][j] < 0 || f[i][j] > q) //f[i][j] chưa tính hoặc lớn hơn q
                {
                    f[i][j] = q; //Cực tiểu hóa f[i][j] theo q
                    trace[i][j] = k; //Lưu vết
                }
            }
        }
}

//Hàm này in ra đường chéo i - j nếu (i, j) thực sự là đường chéo
void PrintDiag(int i, int j)
{
    if (j - i >= 2)
        cout << i << '-' << j << ": " << d(i, j) << '\n';
}

//Truy vết tìm phép tam giác phân tối ưu trên đa giác (i, i + 1, …, j)
void DoTrace(int i, int j)
{
    if (j - i <= 2)
        return; //Đa giác có ≤ 3 đỉnh thì không cần tam giác phân
    //In ra những đường chéo được sử dụng
    PrintDiag(i, trace[i][j]);
    PrintDiag(trace[i][j], j);
    //Truy tiếp các đa giác nhỏ hơn mới tạo ra bằng đệ quy
    DoTrace(i, trace[i][j]);
    DoTrace(trace[i][j], j);
}

int main()
{
    ReadInput();
    SolveRecurrence();
    cout << "Minimum triangulation: " << f[0][n - 1] << '\n';
    DoTrace(0, n - 1); //In ra phép tam giác phân đa giác đã cho
}
```

Mặc dù cơ sở quy hoạch động được đặt tại các $f[i][i+1]$ nhưng khi truy vết ta sẽ dừng ngay khi $j - i \leq 2$ mà có thể không cần truy tiếp về đến cơ sở. Điều này hợp lý khi đa giác có 3 đỉnh thì không cần tam giác phân.

Để tính mỗi giá trị $f[i][j]$ ta cần thời gian $O(n)$ nên việc giải công thức truy hồi mất thời gian $O(n^3)$. Phép truy vết mất thời gian $O(n)$. Vậy thời gian thực hiện giải thuật là $O(n^3)$.

Nếu như trong bài toán Knapsack (mục 1.3.3), ta giải công thức truy hồi theo kiểu từ trên xuống bằng đệ quy rồi truy vết bằng thuật toán lặp thì ở bài toán phân hoạch tam giác, ta giải công thức truy hồi theo kiểu từ dưới lên bằng thuật toán lặp rồi truy vết bằng đệ quy.

Thực ra chỉ có một quy tắc bắt buộc là phép truy vết phải lần ngược lại **chính xác** quá trình giải công thức truy hồi, còn việc sử dụng thuật toán lặp hay đệ quy tùy theo bài toán mà ta lựa chọn cách cài đặt ngắn gọn và hiệu quả nhất.

1.3.6. Lịch thực hiện tác vụ

Một máy tính cần thực hiện n tác vụ đánh số từ 0 tới $n-1$. Tác vụ thứ i thực hiện mất thời gian t_i trên một bộ vi xử lý của máy. Có m ràng buộc, mỗi ràng buộc cho bởi cặp chỉ số (x, y) cho biết tác vụ x phải được hoàn thành trước khi thực hiện tác vụ y. Hai tác vụ độc lập với nhau (không có ràng buộc nào giữa chúng) có thể thực hiện song song trên hai bộ xử lý khác nhau của máy (giả thiết số bộ vi xử lý của máy là không hạn chế). Hãy tìm một lịch thực hiện tác vụ, tức là cho biết thời điểm bắt đầu và kết thúc mỗi tác vụ, sao cho máy có thể hoàn thành tất cả n tác vụ trong thời gian nhanh nhất. Giả thiết thời điểm máy có thể bắt đầu thực hiện các tác vụ là thời điểm 0 và luôn tồn tại cách hoàn thành n tác vụ, tức là không có hai tác vụ x, y mà để thực hiện tác vụ y, cần hoàn thành tác vụ x trước và ngược lại, để thực hiện tác vụ x, cần hoàn thành tác vụ y trước.

Input

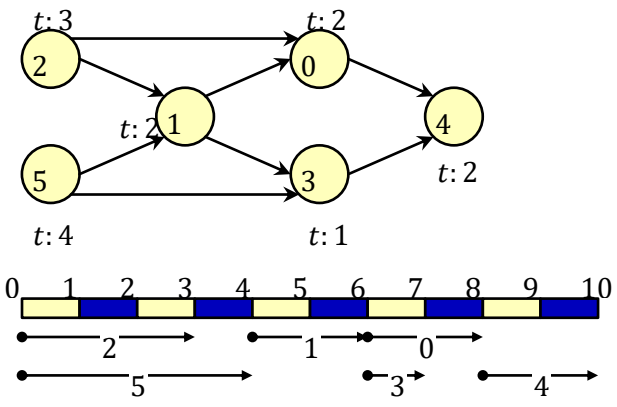
- Dòng 1 chứa hai số nguyên dương $n \le 10^5; m \le 10^6$
- Dòng 2 chứa n số nguyên dương $t_0, t_1, \dots, t_{n-1}$ $(t_i \le 10^4, \forall i)$
- m dòng tiếp theo, mỗi dòng chứa một ràng buộc: gồm hai số nguyên x, y cho biết tác vụ x phải hoàn thành trước khi thực hiện tác vụ y.

Output

Với mỗi tác vụ, cho biết nó có thể thực hiện vào thời gian nào trong phương án tối ưu

Ví dụ

Sample Input	Sample Output
6 8 2 2 3 1 2 4 0 4 1 0 1 3 2 0 2 1 3 4 5 1 5 3	Task 0: [6; 8] Task 1: [4; 6] Task 2: [0; 3] Task 3: [6; 7] Task 4: [8; 10] Task 5: [0; 4] Finish at: 10



❋ Thuật toán

Ký hiệu $B[y]$ là tập các tác vụ phải hoàn thành trước khi thực hiện tác vụ y theo các ràng buộc đã cho. Gọi $f(y)$ là thời điểm sớm nhất mà máy có thể hoàn thành tác vụ y theo lịch trình thỏa mãn yêu cầu đặt ra. Ta sẽ lập công thức truy hồi tính $f(y)$.

- Nếu $B[y] = \emptyset$, tác vụ y không bị phụ thuộc vào tác vụ nào khác, khi đó tác vụ y có thể thực hiện ngay từ thời điểm 0 và hoàn thành tại thời điểm t_y. Trong trường hợp này $f(y) = t_y$.
- Nếu $B[y] \neq \emptyset$, tác vụ y chỉ được bắt đầu sau khi đã hoàn thành mọi tác vụ $\in B[y]$. Ta sẽ cố gắng hoàn thành các tác vụ $\in B[y]$ sớm nhất và ngay khi chúng hoàn thành, tác vụ y sẽ được bắt đầu ngay lập tức. Theo cách làm đó, thời điểm mà mọi tác vụ $\in B[y]$ hoàn thành là $\max\limits_{\forall x \in B[y]}\{f(x)\}$ và như vậy $f[y] = \max\limits_{\forall x \in B[y]}\{f(x)\} + t_y$

Ta có công thức truy hồi:

$$f(y) = \begin{cases} t_y, \text{nếu } B[y] = \emptyset \\ \max\limits_{\forall x \in B[y]}\{f(x)\} + t_y, \text{nếu } B[y] \neq \emptyset \end{cases}$$

Vì $f(y)$ được tính dựa vào các $f(x)$ với $\forall x \in B[y]$ nên công thức truy hồi có thể giải theo kiểu từ trên xuống bằng một hàm đệ quy kết hợp với mảng ghi nhớ: $g[y]$ lưu lại những giá trị $f(y)$ đã được tính. Đây là một ví dụ mà thứ tự nhỏ/lớn của các bài toán con không được xác định từ đầu mà được xác định trong quá trình giải.

Không khó khăn để chứng minh phương án mà mọi mọi tác vụ y được hoàn thành vào thời điểm $f(y)$ chính là phương án tối ưu cần tìm.

❋ Cài đặt

TASKSCHEDULE.cpp.cpp Lập lịch thực hiện tác vụ

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
const int maxN = 1e5;
const int maxT = 1e4;
int n, t[maxN], g[maxN];
vector<int> B[maxN];

void Enter() //Nhập dữ liệu
{
    int m;
    cin >> n >> m;
    for (int i = 0; i < n; ++i)
        cin >> t[i];
    for (; m > 0; --m)
    {
        int x, y;
        cin >> x >> y;
        B[y].push_back(x);
    }
}

void Maximize(int& Target, int Value) //Cực đại hóa Target theo Value
{
    if (Value > Target) Target = Value;
}

```

```cpp
int f(int y) //Tính f(y), dùng g[y] lưu lại kết quả tránh tính lại
{
    if (g[y] == -1) //g[y] chưa được tính
    {   //Tính g[y] bằng đệ quy
        g[y] = 0;
        for (int x: B[y])
            Maximize(g[y], f(x));
        g[y] += t[y];
    }
    return g[y]; //Trả về giá trị g[y] trong kết quả hàm f(y)
}

void Solve()
{
    int time = 0;
    fill(g, g + n, -1); //Tất cả các phần tử g[0 … n) chưa biết (= -1)
    for (int i = 0; i < n; ++i)
    {
        cout << "Task " << i << ": ["
             << f(i) - t[i] << "; "
             << f(i) << "]\n";
        if (f(i) > time) time = f(i);
    }
    cout << "Finish at: " << time << '\n'; //time = max{f(i)}
}

int main()
{
    Enter();
    Solve();
}
```

Những ví dụ đã nêu trong bài chỉ là những bài toán động điển hình, lời giải không quá phức tạp nhằm mục đích làm rõ các bước giải. Một bài toán quy hoạch động có thể có nhiều cách tiếp cận khác nhau, nhưng quan trọng nhất là phải nhìn ra được bản chất đệ quy của bài toán và tìm ra công thức truy hồi để giải. Việc này không có phương pháp chung nào cả mà hoàn toàn dựa vào sự khéo léo và kinh nghiệm - những kỹ năng chỉ có được nhờ luyện tập.

Bài tập 1-1

Cho một thanh gỗ độ dài L, người ta có thể cắt thanh gỗ này ra thành các khúc để bán, biết giá bán một khúc gỗ độ dài x là $c(x)$ ($\forall 1 \leq x \leq L$), hãy tìm thuật toán $O(L^2)$ xác định cách cắt thanh gỗ ban đầu để bán được nhiều tiền nhất.

Bài tập 1-2

Cho hai xâu A, B. Gọi LCS là xâu dài nhất mà là dãy con của cả A và B, gọi SCS là xâu ngắn nhất mà nhận cả A và B là dãy con. Chứng minh rằng:

$$|SCS| = |A| + |B| - |LCS|$$

Ở đây $|.|$ là ký hiệu độ dài xâu.

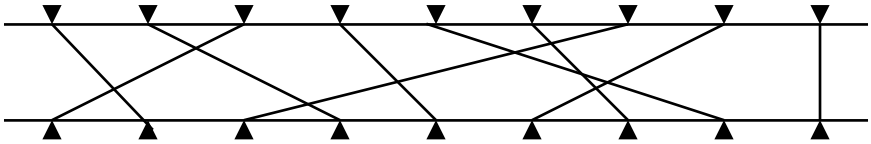
Bài tập 1-3

Cho ba số nguyên dương n, k và p. Tìm thuật toán $O(n.k)$ cho biết có bao nhiêu số tự nhiên có không quá n chữ số mà tổng các chữ số đúng bằng k. Tìm thuật toán $O(n)$ cho biết nếu mang tất cả các số đó sắp xếp theo thứ tự tăng dần thì số đứng thứ p là số nào.

Gợi ý: Tìm công thức truy hồi tính $f(i,j)$ là số các số có $\leq i$ chữ số mà tổng các chữ số đúng bằng j.

Bài tập 1-4

Cho hai đường thẳng song song a và b. Xét n đoạn thẳng, mỗi đoạn thẳng có một đầu nằm trên đường thẳng a và một đầu nằm trên đường thẳng b. Biết rằng không có hai đoạn thẳng nào có chung đầu mút.



Tìm thuật toán $O(n \log n)$ tìm một tập nhiều nhất các đoạn thẳng đôi một không cắt nhau

Tìm thuật toán $O(n \log n)$ tìm một tập nhiều nhất các đoạn thẳng đôi một cắt nhau

Bài tập 1-5

Cho dãy số nguyên dương $A = a[0 \dots n)$ và một số m. Tìm thuật toán $O(m.n)$ để chọn ra một số phần tử trong dãy A mà các phần tử được chọn có tổng đúng bằng m.

Gợi ý: Tìm công thức truy hồi tính $f(t)$ là chỉ số nhỏ nhất thoả mãn: tồn tại cách chọn trong dãy $a[0 \dots f(t)]$ ra một số phần tử có tổng đúng bằng t.

Bài tập 1-6

Cho dãy số tự nhiên $A = a[0 \dots n)$ Ban đầu các phần tử của A được đặt liên tiếp theo đúng thứ tự cách nhau bởi dấu "?":

$$a[0]?\, a[1]?\, \dots ?\, a[n-1]$$

Yêu cầu: Cho trước số nguyên m, hãy tìm thuật toán $O(m.n)$ để thay các dấu "?" bằng dấu cộng hay dấu trừ để được một biểu thức số học cho giá trị là m.

Ví dụ: Ban đầu dãy A là 1? 2? 3? 4, với $m = 0$ sẽ có phương án $1 - 2 - 3 + 4$.

Bài tập 1-7

Cho một lưới ô vuông kích thước $m \times n$, các hàng của lưới được đánh số từ 1 tới m và các cột của lưới được đánh số từ 1 tới n, trên mỗi ô của lưới ghi một số. Người ta muốn tìm một cách đi từ cột 1 tới cột n của lưới theo quy tắc: Từ một ô (i,j) chỉ được phép đi sang một trong các ô ở cột bên phải có đỉnh chung với ô (i,j). Tìm thuật toán $O(m \times n)$ chỉ ra cách đi mà tổng các số ghi trên các ô đi qua là lớn nhất.

7	2	1	2	6
1	2	5	4	5
1	5	3	5	2
5	2	3	1	1

Bài tập 1-8

Lập trình giải bài toán xếp ba lô với kích thước dữ liệu: $n \leq 10000, m \leq 10000$ và giới hạn bộ nhớ 10MB.

Gợi ý: Vẫn sử dụng công thức truy hồi như ví dụ trong bài, nhưng thay đổi cơ chế lưu trữ. Giải công thức truy hồi lần 1, không lưu trữ toàn bộ bảng phương án mà cứ cách 100 hàng mới lưu lại 1 hàng. Sau đó với hai hàng được lưu trữ liên tiếp thì lấy hàng trên làm cơ sở,

giải công thức truy hồi lần 2 tính qua 100 hàng đến hàng dưới, nhưng lần này lưu trữ toàn bộ những hàng tính được để truy vết.

Bài tập 1-9 (Khoảng cách soạn thảo)

Với xâu ký tự $X = x[0 \dots m)$, xét 3 phép biến đổi:

- $Insert(i, c)$: Chèn ký tự c vào trước vị trí i của xâu X. Quy ước khi $i =$ chiều dài xâu X thì ký tự c được thêm vào cuối xâu X
- $Delete(i)$: Xoá ký tự tại vị trí i của xâu X.
- $Replace(\ i, c)$: Thay ký tự tại vị trí i của xâu X bởi ký tự c.

Yêu cầu: Cho hai xâu ký tự $X = X[0 \dots m)$ và $Y = y[0 \dots n)$. Tìm thuật toán $O(m.n)$ để chỉ ra một dãy gồm ít phép biến đổi nhất biến xâu X thành xâu Y.

Gợi ý: Tìm công thức truy hồi tính hàm mục tiêu $f[i][j]$ là số phép biến đổi ít nhất cần sử dụng để biến xâu $x[i \dots m)$ thành xâu $y[j \dots n)$

Bài tập 1-10

Cho hai xâu ký tự A và B. Hãy tìm xâu C với độ dài lớn nhất là dãy con của cả A và B, nếu có nhiều dãy con chung với độ dài lớn nhất, chỉ ra xâu C có thứ tự từ điển lớn nhất trong số đó. Yêu cầu thuật toán với độ phức tạp $O(|A||B|)$.

Bài tập 1-11

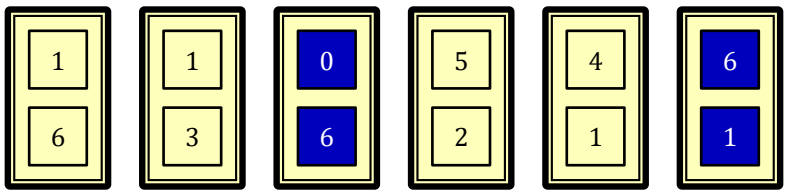
Cho dãy A, tìm thuật toán $O(|A|^2)$ xác định dãy đối xứng dài nhất là dãy con của A và dãy đối xứng ngắn nhất chứa dãy A.

Bài tập 1-12

Có n loại tiền giấy đánh số từ 0 tới $n-1$, tờ giấy bạc loại i có mệnh giá là là một số nguyên dương v_i. Hỏi muốn mua một món hàng giá là m thì có bao nhiêu cách trả số tiền đó bằng những loại giấy bạc đã cho, nếu tồn tại cách trả, cho biết cách trả phải dùng ít tờ tiền nhất. Yêu cầu tìm thuật toán $O(mn)$.

Bài tập 1-13

Cho n quân cờ dominoes xếp dựng đứng theo hàng ngang và được đánh số từ 0 đến $n-1$. Quân cờ dominos thứ i có số ghi ở ô trên là a_i và số ghi ở ô dưới là b_i (a_i, b_i là các số nguyên trong phạm vi từ 0 tới 6). Xem hình vẽ:



Cho phép lật ngược các quân dominoes. Khi quân domino thứ i bị lật, nó sẽ có số ghi ở ô trên là b_i và số ghi ở ô dưới là a_i. Vấn đề đặt ra là hãy tìm cách lật các quân dominos sao cho chênh lệch giữa tổng các số ghi ở hàng trên và tổng các số ghi ở hàng dưới là tối thiểu. Nếu có nhiều phương án lật tốt như nhau, thì chỉ ra phương án phải lật ít quân nhất.

Như ví dụ trên thì sẽ lật hai quân dominos thứ 2 và thứ 5. Khi đó:

Tổng các số ở hàng trên: $1 + 1 + 6 + 5 + 4 + 1 = 18$

Tổng các số ở hàng dưới: $6 + 3 + 0 + 2 + 1 + 6 = 18$

Bài tập 1-14

Xét bảng $H = \{h_{ij}\}$ kích thước 4×4, các hàng và các cột được đánh chỉ số A, B, C, D. Trên 16 ô của bảng, mỗi ô ghi 1 ký tự A hoặc B hoặc C hoặc D.

H	A	B	C	D
A	A	A	B	B
B	C	D	A	B
C	B	C	B	A
D	B	D	D	D

Cho xâu $S = s_0 s_1 \dots s_{n-1}$ chỉ gồm các chữ cái {A, B, C, D}. Xét phép co $R(i)$: thay ký tự s_i và s_{i+1} bởi ký tự $h[s_i][s_{i+1}]$. Ví dụ: $S =$ 'ABCD', áp dụng liên tiếp 3 lần $R(1)$ sẽ được: ABCD → ACD → BD → B

Yêu cầu: Cho trước một ký tự $x \in$ {A, B, C, D}, tìm thuật toán $O(n^2)$ chỉ ra thứ tự thực hiện $n - 1$ phép co để ký tự còn lại cuối cùng trong S là x.

Bài tập 1-15

Cho dãy số nguyên $A = (a_1, a_2, \dots, a_n)$ và một số nguyên dương k, tìm thuật toán $O(nk)$ chọn ra một dãy con gồm nhiều phần tử nhất của A có tổng chia hết cho k.

Bài tập 1-16. Tối ưu phép nhân dãy ma trận

Với ma trận $A = \{a_{ij}\}$ kích thước $m \times n$ và ma trận $B = \{b_{jk}\}$ kích thước $n \times p$. Người ta có phép nhân hai ma trận đó để được ma trận $C = \{c_{ik}\}$ kích thước $m \times p$. Nếu các hàng và các cột được đánh số từ 0 trở đi thì mỗi phần tử của ma trận C được tính theo công thức:

$$c_{ik} = \sum_{j=0}^{n-1} a_{ij}.b_{jk}$$

Ví dụ với A là ma trận kích thước 3×4, B là ma trận kích thước 4×5 thì C sẽ là ma trận kích thước 3×5.

1	2	3	4	
5	6	7	8	
9	10	11	12	

$\times$

1	0	2	4	0
0	1	0	5	1
3	0	1	6	1
1	1	1	1	1

$=$

14	6	9	36	9
34	14	25	100	21
54	22	41	164	33

Xét thuật toán để nhân hai ma trận $A(m \times n)$ và $B(n \times p)$:

```
for (i = 0; i < m; ++i)
    for (k = 0; k < p; ++k)
    {
        c[i][k] = 0;
        for (j = 0; j < n; ++j)
            c[i][k] += a[i][j] * b[j][k];
    }
```

Để nhân hai ma trận $A(m \times n)$ và $B(n \times p)$ chúng ta cần $m \times n \times p$ phép nhân*.

Phép nhân ma trận không có tính chất giao hoán nhưng có tính chất kết hợp:

$$(AB)C = A(BC)$$

Ví dụ nếu A là ma trận cấp 3×4, B là ma trận cấp 4×6 và C là ma trận cấp 6×8 thì:

- Để tính $(AB)C$, phép tính (AB) cho ma trận 3×6 sau $3.4.6 = 72$ phép nhân số học, sau đó nhân tiếp với C được ma trận 3×8 sau $3.6.8 = 144$ phép nhân số học. Vậy tổng số phép nhân số học phải thực hiện sẽ là 216.
- Để tính $A(BC)$, phép tính (BC) cho ma trận 4×8 sau $4.6.8 = 192$ phép nhân số học, lấy A nhân với ma trận này được ma trận 3×8 sau $3 \times 4 \times 8 = 96$ phép nhân số học. Vậy tổng số phép nhân số học phải thực hiện sẽ là 288.

Ví dụ này cho chúng ta thấy rằng trình tự thực hiện có ảnh hưởng lớn tới số phép nhân số học cần thực hiện. Yêu cầu đặt ra là tìm trình tự nhân dãy ma trận $M_0 M_1 \dots M_{q-1}$ để số phép nhân số học phải thực hiện là ít nhất.

Gợi ý: Tìm công thức truy hồi tính hàm mục tiêu $f(i, j)$ là số phép nhân số học tối thiểu cần thực hiện để nhân đoạn ma trận liên tiếp từ M_i tới M_j.

Bài tập 1-17. Tập độc lập cực đại trên cây

Cho một cây gồm n nút, mỗi nút được gán trọng số. Yêu cầu tìm thuật toán $O(n)$ để chọn ra một tập các nút trên cây thỏa mãn: Hai nút bất kỳ được chọn không có quan hệ cha – con và tổng trọng số những nút được chọn là lớn nhất.

Bài tập 1-18

Tìm thuật toán $O(m^2 \log n)$ tìm dãy con chung dài nhất của hai dãy số $a[0 \dots m)$ $b[0 \dots n)$ trong đó m khá nhỏ và n rất lớn.

Gợi ý: Tìm công thức truy hồi tính hàm mục tiêu $f(i, k)$ là chỉ số j nhỏ nhất thỏa mãn dãy con chung dài nhất của $a[0 \dots i]$ và $b[0 \dots j]$ có độ dài k.

Bài tập 1-19. Dãy con tăng chung dài nhất

Cho hai dãy số nguyên $A = a[0 \dots m)$ và $B = b[0 \dots n)$. Tìm thuật toán $O(m.n)$ tìm một dãy số nguyên $C = c[0 \dots p)$ thỏa mãn:

- C vừa là dãy con của A vừa là dãy con của B
- C tăng dần
- Độ dài C lớn nhất có thể

Gợi ý: Tìm công thức truy hồi tính hàm mục tiêu $f(i, j)$ là độ dài dãy con tăng chung dài nhất của $a[0 \dots i]$ và $b[0 \dots j-1]$ sao cho nếu nối thêm $b[j]$ vào cuối dãy con đó ta thu được dãy tăng

Bài tập 1-20. Du lịch Đông ↔ Tây (IOI 1993)

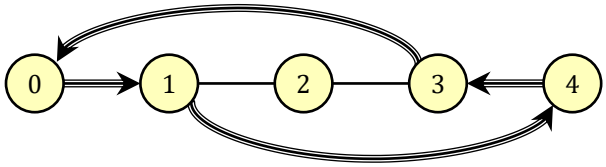
Bạn là người thắng cuộc trong một cuộc thi do một hãng hàng không tài trợ và phần thưởng là một chuyến du lịch do bạn tuỳ chọn. Có n thành phố và chúng được đánh số từ

---

* Để nhân hai ma trận, có một số thuật toán tốt hơn, chẳng hạn thuật toán Strassen $O(n^{\lg 7})$ hay thuật toán Coppersmith-Winograd $O(n^{2,376})$. Ở đây ta dùng thuật toán nhân ma trận đơn giản nhất.

0 tới $n-1$ theo vị trí từ Tây sang Đông (không có hai thành phố nào ở cùng kinh độ), có m tuyến bay hai chiều do hãng quản lý, mỗi tuyến bay nối giữa hai thành phố trong số n thành phố đã cho. Chuyến du lịch của bạn phải xuất phát từ thành phố 0, bay theo các tuyến bay của hãng tới thành phố $n-1$ và chỉ được bay từ Tây sang Đông, sau đó lại bay theo các tuyến bay của hãng về thành phố 0 và chỉ được bay từ Đông sang Tây. Hành trình không được thăm bất kỳ thành phố nào quá một lần, ngoại trừ thành phố 0 là nơi bắt đầu và kết thúc hành trình.

Yêu cầu: Tìm thuật toán $O(n^3)$ xác định hành trình du lịch qua nhiều thành phố nhất.

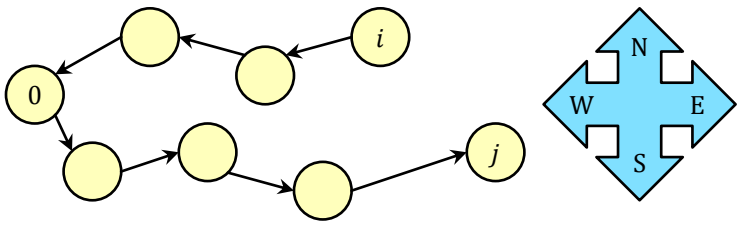


Hướng dẫn giải: Hành trình đi qua nhiều thành phố nhất cũng là hành trình đi bằng nhiều tuyến bay nhất mà không có thành phố nào thăm qua hai lần ngoại trừ thành phố 0 là nơi bắt đầu và kết thúc hành trình. Với hai thành phố i và j trong đó $i < j$, ta xét các đường bay xuất phát từ thành phố i, bay theo hướng Tây tới thành phố 0 rồi bay theo hướng Đông tới thành phố j sao cho không có thành phố nào bị thăm hai lần.

Nếu tồn tại đường bay như vậy, ta xét đường bay qua nhiều tuyến bay nhất, ký hiệu $(i \rightsquigarrow 0 \rightsquigarrow j)$ và gọi $f(i,j)$ là số tuyến bay dọc theo đường bay này.

Nếu không tồn tại đường bay thoả mãn điều kiện đặt ra, ta coi $f(i,j) = -\infty$

Chú ý rằng chúng ta chỉ quan tâm tới các $f(i,j)$ với $i < j$ mà thôi.



Nếu tính được các $f(i, n-1)$ với $\forall i$ thì tua du lịch cần tìm sẽ gồm các tuyến bay trên một đường bay $(i \rightsquigarrow 0 \rightsquigarrow n-1)$ nào đó ghép thêm tuyến bay $(n-1, i)$, tức là số tuyến bay (nhiều nhất) trên tua du lịch tối ưu cần tìm sẽ là $\max_i \{f[i][n-1]\} + 1$. Công thức truy hồi tính các $f[i][j]$ có thể xây dựng được khi xét mọi khả năng của thành phố k liền trước thành phố j trên đường bay $i \rightsquigarrow 0 \rightsquigarrow j$.

Bài tập 1-21. Thuật toán Viterbi

Thuật toán Viterbi được đề xuất bởi Andrew Viterbi như một phương pháp hiệu chỉnh lỗi trên đường truyền tín hiệu số. Nó được sử dụng để giải mã chập sử dụng trong điện thoại di động kỹ thuật số (CDMA/GSM), đường truyền vệ tinh, mạng không dây, v.v... Trong ngành khoa học máy tính, thuật toán Viterbi được sử dụng rộng rãi trong Tin-Sinh học, xử lý ngôn ngữ tự nhiên, nhận dạng tiếng nói. Khi nói tới thuật toán Viterbi, không thể không kể đến một ứng dụng quan trọng của nó trong mô hình Markov ẩn (Hidden Markov Models - HMMs) để tìm dãy trạng thái tối ưu đối với một dãy tín hiệu quan sát được.

Việc trình bày cụ thể một mô hình thực tế có ứng dụng của thuật toán Viterbi là rất dài dòng, chúng ta sẽ tìm hiểu thuật toán này qua một bài toán đơn giản hơn để qua đó hình dung được cách thức thuật toán Viterbi tìm đường đi tối ưu trên lưới như thế nào.

Một dây chuyền lắp ráp ô tô có một robot và n dụng cụ đánh số từ 0 tới $n-1$. Có tất cả m loại bộ phận trong một chiếc ô tô đánh số từ 0 tới $m-1$. Mỗi chiếc ô tô phải được lắp ráp từ t bộ phận $C = c[0 \dots t)$ theo đúng thứ tự này ($\forall i: 0 \le c_i < m$). Biết được những thông tin sau:

- Tại mỗi thời điểm, robot chỉ có thể cầm được 1 dụng cụ.
- Tại thời điểm bắt đầu, robot không cầm dụng cụ gì cả và phải chọn một trong số n dụng cụ đã cho, thời gian chọn không đáng kể.
- Khi đã có dụng cụ, robot sẽ sử dụng nó để lắp một bộ phận trong dãy C, biết thời gian để Robot lắp bộ phận loại k bằng dụng cụ thứ i là b_{ik} ($0 \le i < n; 0 \le k < m$)
- Sau khi lắp xong mỗi bộ phận, robot được phép đổi dụng cụ khác để lắp bộ phận tiếp theo, biết thời gian đổi từ dụng cụ i sang dụng cụ j là a_{ij}. (a_{ij} có thể khác a_{ji} và a_{ii} luôn bằng 0).
- Hãy tìm ra quy trình lắp ráp ô tô một cách nhanh nhất.

Gợi ý: Gọi $f(k, i)$ là thời gian ít nhất để lắp ráp dãy bộ phận $c[k \dots t)$ mà dụng cụ được robot dùng để lắp bộ phận đầu tiên (c_k) là dụng cụ i. Nếu dụng cụ dùng để lắp bộ phận tiếp theo (c_{k+1}) là dụng cụ j thì thời gian ít nhất để lắp lần lượt các bộ phận $c[k \dots t)$: $f[k, i]$ sẽ được tính bằng:

Thời gian lắp bộ phận c_k: $b[i][c_k]$

Cộng với thời gian đổi từ dụng cụ i sang dụng cụ j: $a[i][j]$

Cộng với thời gian ít nhất để lắp ráp lần lượt các bộ phận $c_{k+1}, c_{k+2}, \dots, c_{t-1}$, trong đó bộ phận c_{k+1} được lắp bằng dụng cụ j: $f(k+1, i)$

Vì chúng ta muốn cực tiểu hóa $f(k, i)$, có thể thử mọi khả năng chọn dụng cụ j để lắp c_{k+1} và chọn phương án mà theo đó $f(k, i)$ nhỏ nhất. Từ đó suy ra công thức truy hồi:

$$f(k, i) = \min_{\forall j: 0 \le j < n} \{b[i][c_k] + a[i][j] + f(k+1, j)\}$$