Phương pháp Quy hoạch động (Dynamic Programming)

I. Ý tưởng của phương pháp quy hoạch động

Qui hoạch động là một phương pháp giải các bài toán bằng kết hợp lời giải của các bài toán con theo nguyên tắc là giải các bài toán con đúng một lần, lưu lười giải vào một bảng và thử các phương án phối hợp các bài toán con để giải bài toán lơn một cách tốt nhất.

Về bản chất đó là phương pháp đệ quy từ dưới lên.

Phương pháp này thường được áp dụng cho các bài toán tìm lời giả có giá trị tối ưu.

II. Nhận dạng bài toán

Các bài toán có các đặc trưng sau thì có thể giải bằng quy hoạch động

1. Bài toán tìm lời giải tối ưu.
2. Bài toán có thể được phân rã thành nhiều bài toán con mà sự phối hợp lời giải của các bài toán con sẽ cho ta lời giải của bài toán ban đầu.
3. Quá trình từ các bài toán con đơn giản nhất để tìm ra lời giải của bài toán ban đầu có thể được thực hiện qua một số hữu hạn các bước có tính truy hồi.
4. Không gian bộ nhớ cần thiết cho các bài toán con là khả thi.

III. Các bước xây dựng thuật toán quy hoạch động

Quan trọng nhất: Công thức truy hồi tìm nghiệm bài toán lớn thông qua nghiệm của bài toán con đã giải quyết.

1. Bước 1: Xây dựng bảng phương án. Các giá trị lời giải của các bài toán con được lưu vào một bảng gọi là bảng phương án.
   - Bảng phương án kí hiệu F(i): nghiệm bài toán ở bước thứ i.
   - F(i) được lựa chọn dựa trên các giá trị F(j) mà j<i.
   - F – thường là bảng 1,2, hoặc 3 chiều.
2. Bước 2: Xác định nghiệm của bài toán. Truy vết để xác định các bài toán con tham gia vào nghiệm.

IV. Các ví dụ

1. Bài toán tìm đường đi ngắn nhất. Thuật toán Floyd:

Đặt bài toán:

Cho đồ thị vô hướng, có trọng số G=(V,E), N đỉnh và M cạnh. Bài toán đặt ra là xác định mọi d[u,v] là đường đi ngắn nhất từ đỉnh u đến đỉnh v.

Input: file floyd.inp. Dòng đầu tiên ghi số đỉnh và số cạnh. Các dòng tiếp theo là một cặp 3 giá trị lần lượt là đỉnh xuất phát, đỉnh đích và độ dài quãng đường từ đỉnh xuất phát đến đích.

Output:

1. *Người dùng nhập vào 2 đỉnh I, j, in trực tiếp ra màn hình phương án đi từ I đến j một cách ngắn nhất.*

2. *Ghi ra file floyd.out ma trận mới thể hiện đường đi ngắn nhất từ i đến j với mọi i, j.*

Phân tích bài toán:
Bài toán có những đặc trưng để có thể giải bằng phương pháp quy hoạch động.

1. Đây là một bài toán tối ưu, tìm đường đi ngắn nhất đối với mọi cặp đỉnh.
2. Bài toán có thể được phân rã thành nhiều bài toán con. Xét một lộ trình ngắn nhất từ i đến j, ký hiệu là p. Gọi k là một đỉnh trung gian được đánh số cao nhất. Ta có nhận xét sau
   - Nếu k không phải là một đỉnh trung gian của lộ trình p, thì tất cả các đỉnh trung gian của lộ trình p nằm trong tập hợp {1,2,.. k-1}. Như vậy, một lộ trình ngắn nhất từ đỉnh i đến đỉnh j với tất cả các đỉnh trung gian trong tập hợp {1,2,… , k-1} cũng là một lộ trình ngắn nhất từ i đến j với tất cả các đỉnh trung gian trong tập hợp {1,2,… ,k}.
   - Nếu k là một đỉnh trun ggian của lộ trình p, thì ta tách nhỏ p thành p1: i đến k và p2: k đến j. Thế thì p1 (hoặc p2) là một lộ trình ngắn nhất từ i đến k (hoặc từ k đến j) với tất cả các đỉnh trung gian trong tập hợp {1,2,… , k}.

Các bước để xây dựng thuật toán quy hoạch động:
Cho d[i, j] lưu giá trị của lộ trình ngắn nhất từ i đến j. Vào lúc đầu nó bằng trọng số của cạnh (i,j). Chúng ta liên tục cải thiện d[i,j] bằng cách kiểm tra xem liệu k có phải là một đỉnh thuộc lộ trình ngắn nhất từ i đến j hay không.
Công thức đệ quy từ dưới lên:

- D(k) [i,j] = w[i,j] nếu k=0 (không có đỉnh trung gian trong lộ trình ngắn nhất từ i đến j)
- D1 = D(k-1) [i,j] trong trường hợp lộ trình ngắn nhất từ i đến j không đi qua k

D2 = D(k-1) [i,k] + D(k-1) [k,j] trong trường hợp lộ trình ngắn nhất đi qua k.
D(k) [i,j]= min{D1,D2}.

Lược đồ thuật toán:
Floyd()

```
For k=1 to n
    For i=1 to n
        For j=1 to n
            If(d[i,j]>d[i,k]+d[k,j])
                d[i,j] = d[i,k] + d[k,j]
                t[i,j] = k.
```

Thủ tục tìm vết:

```
Trace(i,j)
    if (t[i,j]=0) print i→j;
    else
        Trace(i,t[i,j])
        print →t[i,j]
        Trace(t[i, j],j)
    end.
```

Thời gian thực hiện: $O(|V|^3)$.
Chương trình: file floyd.cpp; dữ liệu vào: floyd.inp; dữ liệu xuất: floyd.out. Chi tiết trong thư mục Floyd.

2. Mô hình bài toán "Xếp ba lô":

2.1. Bài toán Xếp ba lô:

Một balô có thể chứa M đơn vị trọng lượng. Có N vật trong đó vật thứ i có trọng lượng W[i] và có giá trị sử dụng là V[i]. Hỏi nên chọn những vật nào bỏ vào balô để tổng giá trị sử dụng là lớn nhất có thể.

Phân tích:

Nếu gọi B[i,j] là giá trị lớn nhất có thể có bằng cách chọn trong các vật {1, 2, …. i } với giới hạn trọng lượng j thì giá trị lớn nhất khi chọn được trong số N vật với giới hạn trọng lượng M chính là B[N, M].

i. Công thức truy hồi tính B[i, j]

Với giới hạn trọng lượng j việc chọn tối ưu trong số các vật {}để có giá trị lớn nhất sẽ có hai khả năng:

- Nếu không chọn vật thứ i thì B[i,j] là giá trị lớn nhất có thể bằng cách chọn trong số các vật {1, 2, … i-1} với giới hạn trọng lượng là j. Tức là B[i,j] = B[i-1, j]
- Nếu chọn vật thứ i (chỉ xét khi mà Wi ≤ j) thì B[i, j] bằng giá trị vật thứ i là Vi cộng với giá trị lớn nhất có thể có được bằng cách chọn trong số các vật {1, 2, … i-1} với giới hạn trọng lượng j-Wi. Tức là về mặt giá trị thu được B[i, j] = Vi + B[i-1, j-Wi]

Vì theo cách xây dựng B[i,j] là giá trị lớn nhất có thể có nên nó sẽ là max trong hai giá trị thu được ở trên.

ii. Cơ sở quy hoạch động

Dễ thấy B[0, j] = giá trị lớn nhất có thể bằng cách chọn trong số 0 gói = 0

iii. Bảng phương án

Bảng phương án B gồm N+1 dòng, M+1 cột trước tiên được điền theo cơ sở Quy hoạch động. Sử dụng công thức truy hồi, dùng dòng 0 tính dòng 1, dùng dòng 1 tính vòng 2….vv cho đến khi tính đến dòng N.

iv. Truy vết

Tính xong bảng phương án ta quan tâm đến B[N,M] - là giá trị lớn nhất thu được khi chọn trong N vật với giới hạn kích thước M. Nếu B[N,M] = B[N-1,M] thì tức là không chọn vật thứ N ta truy tiếp B[N-1,M]. Còn nếu B[N,M] ≠ B[N-1,M] thì ta thông báo có chọn vật thứ N và truy tiếp B[N-1,M-W[N]]. Cứ tiếp tục cho đến dòng thứ 0(dòng trên cùng) của bảng phương án.

Cài đặt chi tiết trong thư mục XepBaLo.

2.2. Bài toán Treo tranh:
Cho N bức tranh mã số từ 1 đến N không vượt quá 50. Người ta cần chọn ra một bức tranh để đặt ở cửa ra vào phòng tranh, số còn lại treo thẳng hàng trong phòng vào M vị trí, treo theo trật tự nghiêm ngặt sau đây:

- *Tranh có số hiệu nhỏ phải nằm bên trái tranh có số hiệu lớn.*
- *Tranh thứ i treo tại cửa sẽ đạt giá trị thẩm mỹ C[i].*
- *Tranh thứ i đặt tại vị trí j sẽ đạt giá trị thẩm mỹ V[i, j] (j = 1, 2, ... M ≥ N)*

Hãy xây dựng một phương án treo tranh sao để có giá trị thẩm mỹ là lớn nhất.

Dữ liệu vào cho bởi file TREOTRANH.INP trong đó dòng đầu ghi 2 số N và M lần lượt là tổng số bức tranh và tổng số vị trí sau vị trí cửa ra vào. Tiếp đó là N + 1 dòng, trong đó dòng thứ nhất (của N+1 dòng này) ghi các số C[i] (N số), N dòng sau ghi các số V[i, j] (mỗi dòng M số).
Dữ liệu ra ghi vào file TREOTRANH.OUT trong đó dòng đầu tiên ghi tổng giá trị thẩm mỹ lớn nhất thu được, sau đó là N dòng, mỗi dòng ghi 2 số một là số hiệu của ảnh và một là số hiệu của vị trí đặt ảnh. Với quy ước vị trí cửa ra vào điền số hiệu 0.

Ví dụ:

TREOTRANH.INP	TREOTRANH.OUT
4 6	496
100 22 345 12	1 1
123 23 12 56 29 19	2 2
213 44 124 12 200 32	3 6
12 56 67 87 128 86	4 7
34 300 31 90 87 55	

Phân tích:

Rõ ràng đây là một bài toán thuộc vào dạng tìm một lời giải tối ưu được thành lập từ N kết quả tìm được. Theo nguyên lý quy hoạch động Bellman ta có thể đi tìm được lời giải tối ưu từ việc tối ưu tất cả các kết quả hay các bài toán con của nó.

Nếu ta coi cửa ra vào thực tế cũng chỉ là một vị trí đặt tranh đơn thuần thì bài toán sẽ có dạng đơn giản như sau: Tìm phương án đặt N bức tranh vào M+1 vị trí sao cho tổng giá trị thẩm mỹ là lớn nhất mà cách đặt tranh phải thỏa mãn một số ràng buộc sau:

- Tranh phải được đặt theo thứ tự từ cửa vào đến vị trí M (bây giờ sẽ là vị trí thứ M+1, nhưng sẽ coi vị trí cửa ra vào là vị trí mang số hiệu 0), tranh có số hiệu nhỏ phải nằm bên trái tranh có số hiệu lớn.
- Tranh thứ i treo tại vị trí j sẽ đạt giá trị thẩm mỹ là A[i, j] trong đó

$$A[i,j] = \begin{cases} C[i] & \text{nếu } j = 0 \\ V[i,j] & \text{nếu } j > 0 \end{cases}$$

Dễ thấy nếu đặt F[i,j] là hàm mục tiêu cho bài toán thể hiện giá trị thẩm mỹ lớn nhất khi treo i bức tranh đầu tiên vào j vị trí đầu tiên tính từ cửa. Thế thì ta có công thức sau đây: (với $0 \le i \le N$, $0 \le j \le M+1$)

$$F[i,j] = \begin{cases} 0 & \text{nếu i > j hoặc i = 0 hoặc j = 0} \\ \sum_{k=0}^{i} A[k,k] & \text{nếu i = j} \\ \max\{F[i,j-1], F[i-1,j-1] + A[i,j]\} & \text{nếu i < j} \end{cases}$$

Trường hợp i = j thì hiển nhiên khi đó chỉ có thể có đúng một cách đặt đó là xếp đúng theo thứ tự (vì ràng buộc tranh có số hiệu nhỏ hơn phải nằm bên trái tranh có số hiệu lớn hơn).

Trong trường hợp i < j:

- Nếu ta không chọn bức tranh thứ i vào vị trí thứ j thì nó có thể đã được chọn đặt ở một ví trí trước đó và tổng giá trị thẩm mỹ lúc này chính bằng tổng giá trị thẩm mỹ khi đặt i bức tranh đầu tiên vào j – 1 vị trí đầu tiên.
- Nếu chọn bức tranh i vào vị trí thứ j thì giá trị thẩm mỹ sẽ là

F[i-1, j-1] + A[i, j]

→ Kết quả cuối cùng sẽ là F[N, M+1] !

Cài đặt:

```
void TreoTranh()
{
    int i, j, k;

    //Khởi tạo ma trận phương án
    for(i = 0; i <= N; i++)
        for(j = 0; j <= M+1; j++)
            F[i][j] = 0;

    printf("\nTap phuong an:\n");
    for(i = 1; i <= N; i++)
        for(j = 1; j <= M+1; j++)
        {
            if(i == j)
            {
                for(k = 1; k <= i; k++)
                    F[i][i] += A[k][k];
                printf("%d %d %d\n", i, i, F[i][i]);
            }
            else if(i < j)
            {
                F[i][j] = F[i][j-1] > F[i-1][j-1] + A[i][j] ?
 F[i][j-1] : F[i-1][j-1] + A[i][j];
                printf("%d %d %d\n", i, j, F[i][j]);
            }
        }
}
```

Đánh giá:

Thuật toán trên có độ phức tạp cỡ $O(N \times M)$ cả về không gian cũng như thời gian.

Vấn đề truy vết phương án tối ưu:

Ta cần tìm vị trí đặt của các bức tranh. Do đó dùng một hàm đệ quy để xác định xem khi nào xảy ra tình huống F[i][j] = F[i-1][j-1] + A[i][j] thì tức là bức tranh i được đặt vào vị trí j.

```
void TruyVet(int i, int j, FILE *f)
{
    if(i == 0 || j == 0) return;

    if(F[i][j] == F[i-1][j-1] + A[i][j])
    {
        TruyVet(i-1, j-1, f);
        printf("%d %d\n", i, j);
        fprintf(f, "%d %d\n", i, j);
    }
    else
    {
        TruyVet(i, j-1, f);
    }
}
```

Gọi đệ quy từ cuối: `TruyVet(N, M+1, f);`
Chương trình chi tiết được cài đặt trong thư mục TreoTranh.

2.3. Bài toán Xếp hàng mua vé:
N người xếp hàng mua vé. Ta đánh số cho họ từ 1 đến N theo thứ tự đứng trong hàng. Thời gian phục vụ bán vé cho người thứ i là Ti. Mỗi người cần mua một vé nhưng được quyền mua tối đa là 2 vé, vì thế một số người có thể nhờ người đứng ngay trước mình mua hộ. Người thứ i nhận mua vé cho người thứ i+1, thì thời gian mua vé cho hai người là Ri. Tìm phương án mua vé cho N người sao cho N người đều có vé với thời gian mua là ít nhất.
Dữ liệu vào cho bởi file MUAVE.INP trong đó dòng đầu tiên ghi số N thể hiện số người mua vé. Tiếp sau đó là N dòng mỗi dòng lần lượt ghi hai số Ti và Ri. Dòng cuối cùng chỉ ghi TN.
Dữ liệu ra ghi vào file MUAVE.OUT, dòng đầu tiên ghi tổng thời gian bán vé ít nhất. Các dòng sau ghi số hiệu của người mua. Hai người mua chung thì ghi chung trên một dòng.

Ví dụ:

MUAVE.INP	MUAVE.OUT
10	70
10 11	1
8 13	2 3
19 20	4 5
9 16	6 7
13 14	8
4 13	9 10
15 12	
2 9	
11 16	

14

Phân tích:

Nếu gọi F[i][j] là tổng lượng thời gian nhỏ nhất khi sắp xếp mua vé cho i người đầu tiên trong dãy N người đang đứng xếp hàng, trong đó nếu j = 0 thì người thứ i nhờ người đứng trước mình mua hộ, còn j = 1 thì người đứng thứ i sẽ tự mua (có thể sẽ mua cả cho người đứng sau). Dễ thấy ta có:

$$F[i][j] = \begin{cases} 0 & \text{nếu i = 0} \\ \infty & \text{nếu i = 1 \&\& j = 0} \\ T[1] & \text{nếu i = 1 \&\& j = 1} \\ \min\{F[i-2][0], F[i-2][1]\} + R[i-1] & \text{nếu i > 1 \&\& j = 0} \\ \min\{F[i-1][0], F[i-1][1]\} + T[i] & \text{nếu i > 1 \&\& j = 1} \end{cases}$$

Trong đó: với trường hợp i > 1

- Nếu người thứ i nhờ người đứng trước mình mua hộ thì bắt buộc người đứng trước phải tự mua vé và khi đó có hai tình huống xảy ra:
  - Nếu người đứng trước nữa không tự mua vé:
    F[i][0] = F[i-2][0] + R[i-1]
  - Nếu người đứng trước nữa tự mua ve:
    F[i][0] = F[i-2][1] + R[i-1]
- Nếu người thứ i tự mua vé cho mình thì có hai tình huống xảy ra:
  - Nếu người đứng trước không tự mua vé thì
    F[i][1] = F[i-1][0] + T[i]
  - Nếu người đứng tự mua vé thì
    F[i][1] = F[i-1][1] + T[i]

Kết quả cuối cùng của bài toán chính là $\min\{F[N][0], F[N][1]\}$.

Cài đặt:

```
void MuaVe()
{
    int i;

    //Khởi tạo ma trận phương án
    for(i = 0; i <= N; i++)
    {
        F[i][0] = 0;
        F[i][1] = 0;
    }

    F[1][0] = 10000;
    F[1][1] = T[1];
```

```
    printf("\nTap phuong an:\n");
    printf("%d %d %d\n", 1, F[1][0], F[1][1]);
    for(i = 2; i <= N; i++)
    {
        F[i][0] = ((F[i-2][0] < F[i-2][1]) ? F[i-2][0] : F[i-2][1])
+ R[i-1];
        F[i][1] = ((F[i-1][0] < F[i-1][1]) ? F[i-1][0] : F[i-1][1])
+ T[i];
        printf("%d %d %d\n", i, F[i][0], F[i][1]); //In phương án
    }
}
```

Đánh giá:

Thuật toán trên có độ phức tạp cỡ $O(N)$ cả về không gian cũng như thời gian.

Vấn đề truy vết phương án tối ưu:

Truy vết ngược từ người cuối cùng N. Tại bước truy vết việc mua của người i, nếu F[i][0] < F[i][1] chứng tỏ người i nhờ người đứng trước mình mua hộ, khi đó người đứng trước (i-1) bắt buộc phải là tự mua nên ta sẽ truy vết tiếp từ người i-2. Nếu ngược lại thì người i này chỉ mua cho riêng mình (nếu mua cho người đứng trước thì không cần truy vết cho người i này).

```
void TruyVet(int i, FILE *f)
{
    if(i == 0) return;

    if(F[i][0] < F[i][1])
    {
        TruyVet(i-2,f);
        fprintf(f, "%d %d", i-1, i);
        printf("%d mua cho %d", i-1, i);
    }
    else
    {
        TruyVet(i-1,f);
        fprintf(f, "%d", i);
        printf("%d tu mua", i);
    }
    fprintf(f,"\n");
    printf("\n");
}
```

Gọi thủ tục truy vết từ người cuối cùng trong hàm KetQua(): `TruyVet(N,f);`
Chương trình chi tiết được cài đặt trong thư mục MuaVe.

3. Mô hình bài toán "Dãy con đơn điệu tăng dài nhất":

Mô hình của bài toán:

Cho một dãy các yếu tố $U = \{U_1, U_2, ..., U_N\}$ *được sắp xếp theo một quan hệ thứ tự nào đó.*

Yêu cầu cần tìm một dãy các yêu tố con $\overset{*}{U} = \{U_{k_1}, U_{k_2}, ..., U_{k_l}\}$ *của dãy đã cho sao cho vẫn thỏa mãn quan hệ thứ tự trong dãy và đồng thời làm cho một tính chất nào đó tối ưu.*

Cách giải:

Giả sử dãy đã được sắp đúng theo quan hệ thứ tự. Khi đó:

Gọi L[i] là giá trị tối ưu của dãy con $\overline{U}$ được kết thúc bởi yếu tố U_i. Ta cần xây dựng lại toàn bộ dãy con $\overline{U}$ này và tính chính xác L[i].

Do dãy U đã được sắp nên để thực hiện công việc này ta chỉ cần xác định yếu tố U_j được chọn trước yếu tố U_i bằng cách duyệt toàn bộ các yếu tố U_k đứng trước yếu tố U_i kiểm tra xem yếu tố nào làm cho $L[k]+\Delta i > L[i]$ thì ta chọn nó ($U_j = U_k$) đứng trước U_i và cập nhật lại $L[i] = L[k] + \Delta i$. Trong đó Δi là lượng tăng tính tối ưu khi chọn U_i vào dãy $\overline{U}$.

Đương nhiên sau khi duyệt hết toàn bộ $L[i], i = \overline{1,N}$ các dãy con thu được sẽ thỏa mãn quan hệ thứ tự đã có (do dãy đã được sắp). Để xác định dãy nào là dãy tối ưu cần tìm ta chỉ cần đi xác định $L\max = \max\{L[i], i = \overline{1,N}\} = L[\text{Imax}]$. Sau đó truy vết các phần tử đứng trước $U_{\text{Im}ax}$ ta sẽ được dãy con tối ưu.

Lược đồ thuật toán:

```
void LuocDo()
{
    int i, k;

    for(i = 0; i < N; i++)
    {
        L[i] = Δ[i]; //Có thể dãy chỉ có đúng phần tử là U_i
        for(k = 0; k < i; k++)
        {
            if(Kiểm tra tính chất cần tối ưu && L[i] < L[k] + Δ[i])
                L[i] = L[k] + Δ[i];
        }
    }

    Lmax = 0;
    for(i = 0; i < N; i++)
    {
        if(L[i] >= Lmax)
        {
            Lmax = L[i];
            Imax = i;
        }
    }
}
```

Đánh giá:

- Thuật toán trên có độ phức tạp về không gian là O(N).
- Về thời gian là $O(N^2)$. Thật vậy, số phép so sánh của thuật toán là:

$$2.\sum_{i=0}^{N-1}(i+1) = N(N+1) = O(N^2)$$

Về vấn đề truy vết tìm dãy con tối ưu:

- Tham số i: thể hiện cần tìm yếu tố đứng trước yếu tố U_i trong dãy con tối ưu.
- Tham số Lc: thể hiện giá trị tối ưu hiện tại ứng với dãy con có yếu tố U_i làm yếu tố cuối.

```
void TruyVet(int i, int Lc)
{
    if(i == 0) return;

    if(L[i] == Lc)
    {
        TruyVet(i - 1, Lc - Δ[i], f);
        InKetQua(i);
    }
    else
    {
        TruyVet(i - 1, Lc, f);
    }
}
```

Gọi đệ quy: **TruyVet(Imax, Lmax).**

Lưu ý:
Có nhiều bài toán gặp phải tình huống quan hệ thứ tự đã được đặt ẩn đi. Do đó vấn đề hết sức thận trọng đó là cần tìm đúng quan hệ thứ tự của các yếu tố và xét xem chúng đã được xếp đúng theo quan hệ thứ tự đó hay chưa, nếu chưa thì hãy sắp lại chúng cho đúng thứ tự. Khi sắp lại thứ tự thì cần thêm một mảng Q đùng để đánh dấu vị trí ban đầu của các yếu tố trong dãy ban đầu khi dãy chưa bị đảo lộn thứ tự ban đầu.

3.1. Bài toán Dãy con đơn điệu tăng dài nhất:
Cho một dãy số nguyên A gồm N phần tử A_i (i = 1,2,... N). Hãy tìm một dãy con đơn điệu tăng của dãy A có nhiều phần tử nhất mà không đảo lộn thứ tự của chúng.
Dữ liệu vào cho bởi file DAYCONDONDIEU.INP gồm 2 dòng. Dòng đầu ghi số lượng phần tử có trong dãy. Dòng tiếp theo liệt kê các phần tử trong dãy.
Dữ liệu ra ghi vào file DAYCONDONDIEU.OUT cũng gồm 2 dòng. Dòng đầu ghi số lượng phần tử của dãy con đơn điệu tăng dài nhất, dòng thứ 2 liệt kê các phần tử thuộc dãy con đơn điệu tăng dài nhất đó.

Ví dụ:

DAYCONDONDIEU.INP	DAYCONDONDIEU.OUT
8	4
8 1 3 2 5 4 7 6	1 2 4 6

Phân tích:

Rõ ràng đây là một ví dụ cho mô hình bài toán tổng quát trên trong đó, quan hệ thứ tự ở đây chính là quan hệ thứ tự về chỉ số của các phần tử trong dãy gốc (do yêu cầu bài toán không được thay đổi thứ tự của chúng), tính chất tối ưu ở đây là tính đơn điệu tăng

và có số phần tử lớn nhất. Do vậy, nếu gọi L[i] là độ dài lớn nhất của dãy con đơn điệu tăng của dãy A kết thúc bởi phần tử A_i. Thế thì:

- Khởi tạo cho L[i] = 1.
- Lần lượt so sánh tất cả các phần tử A_k (đứng trước A_i) với A_i
    - Nếu thấy $A_k \le A_i$ đồng thời kiểm tra nếu thấy L[i] < L[k] + 1 (ở đây $\Delta i = 1, \forall i = \overline{1, N}$ do mỗi lần chọn một phần tử A_i thì dãy con tối ưu hiện tại chỉ tăng lượng phần tử lên là 1) thì
        - Ghép A_i vào dãy con có phần tử cuối là A_k hay:

$$L[i] = L[k] + 1$$

- Sau khi duyệt hết các phần tử đứng trước A_i ta sẽ được một dãy con đơn điệu tăng dài nhất tính đến A_i và với A_i là phần tử đứng cuối cùng.

Cuối cùng nghiệm của bài toán sẽ là

$$Lmax = \max_i \{L[i] + 1\} \text{ với } i = \overline{1, N}$$

Cài đặt:

```
int Lmax = 0;
int Imax;

void DayConDonDieu()
{
     int i;
     for(i = 0; i < N; i++)
     {
          L[i] = 1;
          for(k = 0; k < i; k++)
               if(a[k] <= a[i] && L[i] <= L[k] + 1)
                    L[i] = L[k] + 1;
     }

     for(i = 0; i < N; i++)
          if(L[i] >= Lmax)
          {
               Lmax = L[i];
               Imax = I;
          }
}
```

Vấn đề truy vết để xác định các phần tử của dãy con đơn điệu tăng lớn nhất:

```
void TruyVet(int i, int L)
{
     if(L == 0) return;

     if(L[i] == L)
     {
          TruyVet(i-1, L-1);
          printf("%d ", i);
     }
     else
     {
          TruyVet(i-1, L);
```

```
        }
    }

    TruyVet(Imax, Lmax);
```

Đánh giá:

Chi phí không gian cho bài toán này là $O(N)$, chi phí thời gian là $O(N^2)$.

Chương trình chi tiết được đặt trong thư mục DayConDonDieuTangLonNhat.

3.2. Bài toán Bố trí phòng họp:

Có N cuộc họp, cuộc họp thứ i bắt đầu vào thời điểm Ai và kết thúc vào thời điểm Bi. Do chỉ có một phòng hội thảo nên 2 cuộc họp bất kỳ sẽ cùng được bố trí phục vụ nếu khoảng thời gian làm việc của chúng chỉ giao nhau tại đầu mút. Hãy bố trí phòng họp để phục vụ được nhiều cuộc họp nhất.

Dữ liệu vào cho bởi file PHONGHOP.INP trong đó dòng đầu ghi N là số cuộc họp. N dòng tiếp theo mỗi dòng lần lượt ghi hai số Ai và Bi.

Dữ liệu ra đưa vào file PHONGHOP.OUT có 2 dòng. Dòng đầu tiên ghi tổng số cuộc họp nhiều nhất có thể bố trí. Dòng thứ hai lần lượt ghi số hiệu của các cuộc họp có thể bố trí theo trình tự cuộc họp nào diễn ra trước thì viết trước. Số hiệu của các phòng họp cách nhau bởi đúng 1 ký tự trắng.

Ví dụ:

PHONGHOP.INP	PHONGHOP.OUT
10	4
3 8	5 2 4 3
4 6	
8 9	
7 8	
3 4	
1 4	
1 8	
4 8	
2 4	
2 8	

Phân tích:

Rõ ràng nếu i và j với i < j là hai cuộc họp được bố trí thì B[i] ≤ A[j] ≤ B[j]. Đây chính là quan hệ thứ tự được bài toán ẩn đi. Do đó trước hết ta sẽ sắp xếp lại các cuộc họp theo thứ tự tăng dần về thời điểm kết thúc. Việc sắp xếp này tốt nhất ta sẽ dùng thuật toán QuickSort. Trong trường hợp các cuộc họp có cùng thời điểm kết thúc ta sẽ sắp theo thứ tự tăng dần của thời điểm bắt đầu.

Gọi L[i] là số lượng tối đa các cuộc họp có thể bố trí ứng với cuộc họp cuối cùng là cuộc họp mang số hiệu i (tất nhiên sau khi đã sắp xếp). Hoàn toàn tương tự như phương pháp đã nêu, ta có hướng giải như sau:

- Khởi tạo cho L[i] = 1.
- Lần lượt so sánh thời điểm kết thúc B[k] của các cuộc họp mang số hiệu k (đứng trước cuộc họp i) với thời điểm bắt đầu A[i] của cuộc họp i:

- Nếu thấy B[k] ≤ A[i] (cuộc họp k xảy ra trước cuộc họp i) đồng thời kiểm tra nếu thấy L[i] < L[k] + 1 (ở đây $\Delta i = 1, \forall i = \overline{1, N}$ do mỗi lần chọn một cuộc họp "dãy con" các cuộc họp tối ưu hiện tại chỉ tăng lên 1):
  - Bố trí cuộc họp i vào dãy các cuộc họp có cuộc họp k là cuộc họp cuối, hay:

$$L[i] = L[k] + 1$$

- Sau khi duyệt hết các cuộc họp đứng trước cuộc họp i ta sẽ được một cách bố trí nhiều cuộc họp nhất với i là cuộc họp cuối cùng.

Cuối cùng nghiệm của bài toán sẽ là

$$Lmax = \max_i \{L[i] + 1\} \text{ với } i = \overline{1, N}$$

Cài đặt:

```
void BoTriPhongHop()
{
     int i, k;

     for(i = 0; i < N; i++)
     {
          L[i] = 1;
          for(k = 0; k < i; k++)
          {
               if(B[k] <= A[i] && L[i] < L[k] + 1)
                    L[i] = L[k] + 1;
          }
     }

     Lmax = 0;
     for(i = 0; i < N; i++)
     {
          if(L[i] >= Lmax)
          {
               Lmax = L[i];
               Imax = i;
          }
     }
}
```

Vấn đề truy vết các cuộc họp:

Khác với bài toán Dãy con đơn điệu tăng dài nhất, việc sắp thứ tự trong bài này chỉ dựa trên một yếu tố thời điểm kết thúc các cuộc họp do đó nếu ta chỉ truy vết như trong ví dụ trước thì có thể dẫn tới tình huống khi tìm cuộc họp bố trí trước cuộc họp i ta tìm được một cuộc họp j nào đó đứng trước cuộc họp i trong dãy được sắp nhưng lại không thể bố trí được. Chẳng hạn với thủ tục truy vết của bài Dãy con đơn điệu tăng dài nhất ta có kết quả output như sau:

PHONGHOP.OUT

4

1 8 4 3

Ứng với số hiệu sau khi sắp là 7 8 9 10

Mặc dù L[7] = 1, L[8] = 2, L[9] = 3, L[10] = 4, tuy nhiên A[9] < B[8] và A[8] < B[7] nên phương án bố trí như vậy là không hề khả thi. Để tìm đúng phương án, ta đưa thêm vào

thủ tục truy vết một tham số (sau) cho biết số hiệu của cuộc họp vừa chọn vào dãy tối ưu (tức là cho biết cuộc họp sẽ cần tìm cuộc họp bố trí trước nó), đồng thời tại bước kiểm tra giá trị tối ưu L[i] = Lc ta kiểm tra thêm điều kiện B[i] ≤ A[sau], khi đó các cuộc họp tìm được mới có thể bố trí được.

```
void TruyVet(int i, int Lc, int sau, FILE *f)
{
    if(i == 0) return;

    if(L[i] == Lc && B[i] <= A[sau])
    {
        TruyVet(i - 1, Lc - 1, i, f);
        printf("%d ", Q[i] + 1);
        fprintf(f, "%d ", Q[i] + 1);
    }
    else
    {
        TruyVet(i - 1, Lc, sau, f);
    }

}
```

Gọi đệ quy: Tìm cuộc họp bố trí trước cuộc họp cuối cùng Imax.

```
TruyVet(Imax-1, Lmax-1, Imax, f);
```

Vấn đề sắp xếp:

Sử dụng phương pháp *sắp xếp nhanh* trong đó với hàm phân đoạn *(Partition)* ta đưa thêm một tham số đó là mảng cần sắp xếp. Ban đầu ta sắp xếp toàn bộ các cuộc họp tăng dần theo thời điểm kết thúc: PhanDoan(0, N-1, B), sau đó duyệt lại mảng B, nhặt ra những đoạn con có giá trị bằng nhau ta đem sắp lại các cuộc họp nằm trên đoạn con đó theo thứ tự tăng dần của thời điểm ban đầu (mảng A).

```
void QuickSort() //Theo kiểu sắp xếp nhanh quick sort
{
    int i, j;
    PhanDoan(0,N-1, B);

    i = 0;
    while(i < N)
    {
        if(B[i] == B[i+1])
        {
            j = i;
            while(B[j] == B[j+1]) j++;
            PhanDoan(i,j, A);
            i = j;
        }
        i++;
    }
}

void PhanDoan(int l, int r, int *X)
{
```

```
    int i, j, x;
    i = l;
    j = r;
    x = X[(l+r)/2];

    do
    {
        while(X[i] < x) i++;
        while(X[j] > x) j--;

        if(i <= j)
        {
            HoanVi(&B[i],  &B[j]);
            HoanVi(&A[i],  &A[j]);
            HoanVi(&Q[i],  &Q[j]);
            i++;
            j--;
        }
    }
    while(i<=j);

    if(l < j) PhanDoan(l,j,X);
    if(i < r) PhanDoan(i,r,X);
}
```

Đánh giá độ phức tạp của thuật toán:

- Thuật toán QuickSort có độ phức tạp về mặt thời gian thực hiện là $O(N \log N)$. Việc sắp xếp lại các cuộc họp cũng có độ phức tạp $O(N \log N)$.
- Thủ tục bố trí phòng họp có độ phức tạp cho việc lập bảng phương án là $O(N^2)$.
- Thủ tục truy vết có số lần so sánh lớn nhất là 3N, do đó chi phí tối đa là $O(N)$.

→ Tóm lại chi phí thời gian cho bài toán là cỡ $O(N^2)$.

Chương trình chi tiết được cài đặt trong thư mục BoTriPhongHop.

3.3. Bài toán Cho thuê máy:

Trung tâm tính toán hiệu năng cao nhận được đơn đặt hàng của N khách hàng. Khách hàng i muốn sử dụng máy trong khoảng thời gian từ A[i] đến B[i] và trả khoảng tiền là C[i]. Hãy bố trí lịch cho thuê máy để tổng số tiền thu được là lớn nhất mà thời gian sử dụng máy của hai người khách hàng bất kỳ được phục vụ là không giao nhau (do cả trung tâm chỉ có một máy cho thuê).

Dữ liệu vào cho bởi file CHOTHUEMAY.INP gồm N+1 dòng. Dòng thứ nhất khi N là tổng số khách hàng muốn thuê máy. N dòng tiếp theo mỗi dòng điền 3 số A[i], B[i], C[i]. Dữ liệu ra ghi vào file CHOTHUEMAY.OUT gồm 2 dòng. Dòng đầu ghi tổng số khách có thể bố trí cho thuê và tổng số tiền tối đa thu được. Dòng thứ 2 ghi thứ tự của các khách hàng, mỗi số cách nhau đúng một ký tự trắng.

Ví dụ:

CHOTHUEMAY.INP	CHOTHUEMAY.OUT
6	210
3 8 100	5 2 3 4
3 4 30	

5 7 120
8 9 30
2 3 30
1 3 10

Phân tích:

Sắp xếp lại thứ tự khách hàng theo chiều tăng dần của thời điểm kết thúc. Nếu hai người khách cùng có thời điểm kết thúc sử dụng máy thì ưu tiên người khách nào sẽ sử dụng trước.

Duyệt từ đầu danh sách, điều kiện cần để người thứ i được chọn là nếu thời gian bắt đầu sử dụng của họ là sau thời điểm kết thúc của người chọn ngay trước.

Gọi L[i] là tổng số tiền tối đa khi chọn khách hàng thứ i làm người cuối cùng sử dụng máy. Khởi tạo L[i] = C[i] (có thể xảy ra trường hợp chỉ có thể chọn duy nhất người thứ i sử dụng máy). Tiếp đó ta cần xác định những người được sử dụng máy trước người thứ i này bằng cách duyệt mảng L với các phần tử đứng trước L[i], nếu thấy L[k] + C[i] > L[i] thì cập nhật lại L[i] = L[k] + C[i] với nghĩa người thứ k sẽ được chọn ngay trước người i. Khi đó:

$$L[i] = \max_{k}\{L[k] + C[i]\},\ k=\overline{1,i-1},\ B[k] \le A[i]$$

Cuối cùng, tổng số tiền lớn nhất thu được chính bằng $L\max = \max_{i} L[i],\ i = \overline{1,N}$.

Cài đặt:

```
void SapXep(); //Dùng QuickSort hoặc HeapSort

void ChoThueMay()
{
    int i, k;

    for(i = 0; i < N; i++)
    {
        L[i] = C[i];
        for(k = 0; k < i; k++)
        {
            if(B[k] <= A[i] && L[i] < L[k] + C[i])
                L[i] = L[k] + C[i];
        }
    }

    //Tìm Lmax và chỉ số của người khách cuối được chọn
    Lmax = 0;
    for(i = 0; i < N; i++)
    {
        if(L[i] >= Lmax)
        {
            Lmax = L[i];
            Imax = i;
        }
    }
}

void TruyVet(int i, int Lc, FILE *f) //Tìm những người được chọn
{
    if(i == 0) return;
```

```
        if(L[i] == Lc)
        {
            TruyVet(i - 1, Lc - C[i], f);
            printf("%d ", Q[i] + 1);
            fprintf(f, "%d ", Q[i] + 1);
        }
        else
        {
            TruyVet(i - 1, Lc, f);
        }

    }

    void KetQua()
    {
        FILE *f;
        f= fopen("CHOTHUEMAY.OUT", "wt");

        printf("Tong so tien toi da thu duoc: %d\n", Lmax);
        fprintf(f, "%d\n", Lmax);
        printf("Bo tri lan luot cho cac khach hang sau thue may:\n");
        TruyVet(Imax, Lmax, f);
        fclose(f);
    }
```

Thuật toán có độ phức tạp về thời gian là $O(N^2)$.
Chương trình chi tiết được đặt trong thư mục ChoThueMay.

3.4. Bài toán Dãy tam giác bao nhau:
Cho N tam giác trên mặt phẳng. Tam giác i được gọi là bao tam giác j nếu 3 đỉnh của tam giác j đều nằm trong tam giác i (có thể nằm trên cạnh). Hãy tìm dãy tam giác bao nhau có nhiều tam giác nhất.
Dữ liệu vào cho bởi file TAMGIAC.INP gồm N+1 dòng. Dòng thứ nhất ghi N là số lượng tam giác ta có. N dòng tiếp theo, mỗi dòng ghi 6 số nguyên lần lượt là hoành độ và tung độ của 3 đỉnh của tam giác. Các số trên cùng 1 dòng cách nhau bởi đúng một ký tự trắng.
Dữ liệu ra ghi vào file TAMGIAC.OUT gồm 2 dòng. Dòng thứ nhất ghi tổng số tam giác bao nhau tối đa tìm được. Dòng thứ 2 ghi số hiệu của các tam giác bao nhau, theo thứ tự tam giác nằm trong ghi trước. Tương tự các số hiệu cách nhau bởi đúng một ký tự trắng.

Ví dụ:

Phân tích:

Nhận xét: Dễ thấy nếu tam giác i có thể bao được tam giác k thì khi đó diện tích của tam giác k sẽ phải nhỏ hơn diện tích của tam giác i. Do đó trước hết ta sắp xếp lại thứ tự các tam giác theo hướng tăng dần về diện tích. Khi đó, tam giác k sẽ nằm trong tam giác i nếu k < i và cả 3 cạnh của tam giác k đều nằm trong tam giác i. Và như vậy bài toán có thể thực hiện theo mô hình bài toán dãy con đơn điệu tăng dài nhất. Việc sắp xếp lại các tam giác theo hướng tăng dần của diện tích sẽ đảm bảo ta không gặp những trường hợp kiểm tra tam giác lớn có nằm trong tam giác nhỏ hay không ! Đương nhiên là không.

Gọi L[i] là tổng số tam giác bao nhau tối đa ứng với tình huống tam giác i là tam giác lớn nhất bao tất cả các tam giác trong dãy tam giác bao nhau này.

- Khởi tạo L[i] = 1.
- Lần lượt kiểm tra trong tất cả các tam giác k có diện tích nhỏ hơn tam giác i (k<i) xem có tam giác nào có cả 3 đỉnh nằm trong tam giác i mà L[i] < L[k] + 1 hay không. Nếu có thì cập nhật L[i] = L[k] + 1 (đây là thao tác đưa tam giác i vào dãy tam giác bao nhau có tam giác lớn nhất là k, bao tam giác k bởi tam giác i).
- Sau khi kiểm tra hết các tam giác có diện tích nhỏ hơn tam giác i, ta sẽ thu được một dãy các tam giác bao nhau được bao bởi tam giác i. Số lượng tam giác thuộc dãy là L[i].

Cuối cùng để biết được dãy nào có nhiều tam giác nhất ta tìm

$$L\max = \{L[i], i = \overline{1, N}\} = L[Imax]$$

và truy vết để tìm các tam giác thuộc dãy.

Lưu ý:

- Việc kiểm tra một điểm M có nằm trong tam giác ABC hay không ta dựa vào công thức đặc điểm: Điểm M nằm trong tam giác ABC nếu

$$S_{\Delta ABC} = S_{\Delta ABM} + S_{\Delta ACM} + S_{\Delta BCM}$$

- Để tính diện tích tam giác ta sử dụng công thức Hê-rông:

$$S = \sqrt{(p-a)\times(p-b)\times(p-c)\times p}$$

với p là nửa chu vi tam giác $p = \frac{a+b+c}{2}$.