Bài: Quy hoạch động

Xem bài học trên website để ủng hộ Kteam: Quy hoạch động

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage How Kteam nhé!

Dẫn nhập

Đến thời điểm này thì chúng ta đã cùng nhau đi một hành trình dài với khoá học. Trong bài học ngày hôm nay, chúng ta sẽ đi tìm hiểu về chủ đề cuối cùng trong khóa học này. Đó chính là **Quy hoạch động**.

Nội dung

Để có thể tiếp thu bài học này một cách tốt nhất, các bạn nên có những kiến thức cơ bản về:

- Các kiến thức cần thiết để theo dõi khóa học

Trong bài học ngày hôm nay, chúng ta sẽ cùng nhau tìm hiểu về:

- Tổng quan về Quy hoạch động

Tổng quan về Quy hoạch động

Khái niệm

Quy hoạch động (Dynamic Programming) là một phương pháp tối ưu trong đó bài toán lớn được phân chia thành các bài toán đơn giản hơn. Sau đó, từ kết quả của bài toán đơn giản hơn, ta sẽ tính được kết quả của bài toán ban đầu.

Một ví dụ về quy hoạch động mà các bạn có thể đã biết đó là tính dãy số Fibonacci.

Gọi F_n là số **Fibonacci** thứ *n* trong dãy. Giả sử ta cần tính F_6 thì ta sẽ tính F_5 và F_4, để tính F_4 ta lại tính F_3 và F_2 và **cứ lặp lại như vậy**. Đây chính là quy hoạch động.

Phương pháp chung

Để có thể giải được một bài toán quy hoạch động, chúng ta sẽ cần hai yếu tố cơ bản nhất đó là **công thức truy hồi** và **trường hợp cơ sở**.

Quay trở lại với bài toán về số Fibonacci, ta đã biết công thức truy hồi là $F_n = F_{n-1} + F_{n-2}$. Bản chất của công thức này là đệ quy. Vậy thì nó sẽ đệ quy đến khi nào? Ta cũng biết là $F_1 = F_2 = 1$ nên khi ***n=1*** hoặc ***n=2*** thì ta lấy luôn kết quả $F_n = 1$, đây chính là trường hợp cơ sở.

Như vậy, lời giải quy hoạch động của bài toán tìm số Fibonacci thứ n có thể biểu diễn như sau:

$$F_n = F_{n-1} + F_{n-2} \text{ với } n > 2 \text{ và } F_1 = F_2 = 1$$

Với những bài toán phức tạp thì công thức truy hồi sẽ không chỉ đơn giản là một công thức mà có thể là nhiều hàm, nhiều biến cùng được tính để cho ra kết quả.

Bài toán minh hoạ

Ta sẽ có một bài toán đơn giản về bài toán quy hoạch động như sau:

Cho một bảng hình chữ nhật có kích thước $n \times m$ với ô bắt đầu là ô góc trên bên trái có toạ độ *(1, 1)*, ô kết thúc là ô góc dưới bên phải có toạ độ *(n, m)*. Trên mỗi ô của bảng có ghi một số nguyên dương là số điểm được cộng thêm khi người chơi đi vào ô đó. Ban đầu, người chơi ở ô bắt đầu với số điểm là 0. Biết rằng, người chơi chỉ có thể di chuyển sang phải hoặc đi xuống dưới. Hỏi số điểm tối đa mà người chơi có thể đạt được khi di chuyển từ ô bắt đầu đến ô kết thúc là bao nhiêu?

Input:

- Dòng *1*: Hai số nguyên dương ***n, m*** thể hiện cho kích thước của bảng ***(n, m ≤ 10^3)***
- Dòng *2...n+1*: Mỗi dòng gồm ***m*** số nguyên dương thể hiện cho giá trị các ô trong bảng. Giá trị một ô không vượt quá ***10^9***.

Output:

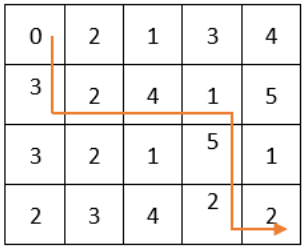
- Một số nguyên duy nhất là kết quả của bài toán

Ví dụ:

Input	Output
4 5 0 2 1 3 4 3 2 4 1 5 3 2 1 5 1 2 3 4 2 2	19

Giải thích ví dụ:

Dưới đây là bảng tương ứng với ví dụ ở trên và cách di chuyển để đạt được số điểm tối đa



Phân tích

Hãy cùng nhau phân tích bài toán ở trên một chút nhé. Ta thấy rằng chỉ có hai cách di chuyển là từ trên xuống và từ trái qua phải. Do đó, để đi đến một ô, ta chỉ có thể đi từ ô phía trên hoặc phía bên trái nó. Nói cách khác, ô ***(i, j)*** chỉ có thể được đi đến từ ô ***(i - 1, j)*** hoặc ô ***(i, j – 1)***.

Gọi ***dp[i][j]*** là tổng giá trị các ô đã đi qua khi đi đến ô ***(i, j)***. Xét ô kết thúc là ô ***(n, m)***, ta thấy để đi đến ô này thì chỉ có thể đi từ ô ***(n-1, m)*** hoặc ô ***(n, m-1)***.

Do đó, ***dp[n][m] = max(dp[n - 1][m], dp[n][m - 1]) + a[n][m]***. Lúc này, việc tính **dp[n][m]** quy về việc tính ***dp[n – 1][m]*** và ***dp[n][m - 1]***. Để tính hai giá trị trên, ta lại tiếp tục chia nhỏ như cách tính ***dp[n][m]***. Đây chính là công thức truy hồi.

Việc tiếp theo là tìm ra trường hợp cơ sở. Ta thấy, mọi cách đi đều bắt đầu tại ô (1,1) và giá trị của ô này luôn là 0. Do đó, ***dp[1][1] = 0*** chính là trường hợp cơ sở.

Tổng kết lại ta có: $dp[i][j] = dp[i-1][j], dp[i][j-1] + a[i][j]\ voi\ (i,j) \not\equiv (1,1)\ dp[1][1] = 0$

Theo các bạn, công thức trên đã ổn chưa? Hãy thử tạm dừng suy nghĩ hoặc code thử để xem vấn đề của công thức trên là gì nhé!

Các bạn đã nhận ra công thức trên thiếu sót gì chưa? Hãy cùng nhau tìm hiểu nhé!

Ta thấy, nếu như các ô được xét ở hàng trên cùng thì sẽ không tồn tại ô nào phía trên nó để có thể đi xuống. Ví dụ như sẽ không tồn tại ô ***(0, 3)*** để đi xuống ô ***(1, 3)***. Tương tự cũng sẽ không tồn tại ô ở phía bên trái các ô ở hàng ngoài cùng bên trái. Do đó, với các ô ở hàng trên cùng và hàng ngoài cùng bên trái, ta sẽ cần xét riêng.

Khi đó, công thức được sửa lại thành:

- ***dp[i][j] = dp[i-1][j], dp[i][j-1] +a[i][j]*** *với* ***i, j>1***
- ***dp[i][j] = dp[i][j-1]+a[i][j]*** *với* ***i=1, j≠1***
- ***dp[i][j] = dp[i-1][j]*** *với* ***j=1, i≠1***
- ***dp[i][j] = 0*** *với* ***(i,j)=(1,1)***

Đến lúc này công thức đã hoàn chỉnh. Ta có thể code bằng đệ quy.

Code

C++:

```
#include<bits/stdc++.h>
using namespace std;

typedef long long ll;

const int MaxN = 1 + 1e3;

int n, m, a[MaxN][MaxN];

ll Calc(int i, int j){
    if(i == 1 && j == 1) return 0;
    if(i == 1) return Calc(i, j - 1) + a[i][j];
    if(j == 1) return Calc(i - 1, j) + a[i][j];
    return max(Calc(i - 1, j), Calc(i, j - 1)) + a[i][j];
}

int main(){
    freopen("CTDL.inp","r",stdin);
    freopen("CTDL.out","w",stdout);
    cin >> n >> m;
    for(int i = 1 ; i <= n ; ++i)
    for(int j = 1 ; j <= m ; ++j) cin >> a[i][j];
    cout << Calc(n, m) << endl;

    return 0;
}
```

Cải tiến

Đoạn code trên nếu chạy với dữ liệu nhỏ thì khá ổn. Tuy nhiên, nếu dữ liệu lớn thì sẽ chạy chậm do có khá nhiều đoạn tính lặp lại. Ví dụ như khi tính ô ***(2, 3)*** thì ta đã yêu cầu tính ô ***(2, 2)*** một lần. Tuy nhiên, khi tính ô ***(3, 2)*** ta lại gọi lại việc tính ô ***(2, 2)*** này một lần nữa. Việc lặp lại này khiến chương trình chạy rất lâu.

Để giải quyết vấn đề trên, ta sẽ dùng một mảng để lưu lại giá trị các ô đã được tính. Nếu như ô đó đã được tính, ta sẽ dùng luôn kết quả lúc trước mà không cần tính lại. Kỹ thuật này được gọi là **Đệ quy có nhớ**.

C++:

```
#include<bits/stdc++.h>
using namespace std;

typedef long long ll;

const int MaxN = 1 + 1e3;

int n, m, a[MaxN][MaxN];
ll dp[MaxN][MaxN];

ll Calc(int i, int j){
    if(dp[i][j] != -1) return dp[i][j];
    if(i == 1) return dp[i][j] = Calc(i, j - 1) + a[i][j];
    if(j == 1) return dp[i][j] = Calc(i - 1, j) + a[i][j];
    return dp[i][j] = max(Calc(i - 1, j), Calc(i, j - 1)) + a[i][j];
}

int main(){
    freopen("CTDL.inp","r",stdin);
    freopen("CTDL.out","w",stdout);
    cin >> n >> m;
    for(int i = 1 ; i <= n ; ++i)
    for(int j = 1 ; j <= m ; ++j) cin >> a[i][j];
    memset(dp, 0xff, sizeof dp);
    dp[1][1] = 0;
    cout << Calc(n, m) << endl;

    return 0;
}
```

Tips:

Ở trong đoạn code trên, các bạn sẽ thấy mình dùng câu lệnh `memset(dp, 0xff, sizeof dp)`. Câu lệnh này là câu lệnh để gán tất cả các phần tử của mảng thành -1. Đây là một cách để đánh dấu một ô đã được tính hay chưa. Nếu như giá trị
dp[i][j] = -1 nghĩa là ô ***(i, j)*** chưa được tính do giá trị của một ô luôn không âm.

Với đoạn code cải tiến trên, độ phức tạp chương trình khi này là $O(n \times m)$ do mỗi ô sẽ chỉ được tính đúng 1 lần.

Khử đệ quy

Cách làm trên là một cách làm đúng. Tuy nhiên, việc dùng đệ quy không tối ưu do nếu gọi đệ quy nhiều thì sẽ tốn bộ nhớ và thời gian. Do đó, ta sẽ tìm cách loại bỏ đệ quy trong code.

Ta nhận thấy rằng, ta sẽ luôn phải tính các ô ở gần với ô bắt đầu trước. Do đó, ta sẽ sử dụng vòng lặp để tính toán giá trị các ô theo thứ tự từ các ô gần ô bắt đầu đến ô kết thúc.

C++:

```
#include<bits/stdc++.h>
using namespace std;

typedef long long ll;

const int MaxN = 1 + 1e3;

int n, m, a[MaxN][MaxN];
ll dp[MaxN][MaxN];

int main(){
    freopen("CTDL.inp","r",stdin);
    freopen("CTDL.out","w",stdout);
    cin >> n >> m;
    for(int i = 1 ; i <= n ; ++i)
    for(int j = 1 ; j <= m ; ++j) cin >> a[i][j];
    for(int i = 1 ; i <= n ; ++i)
    for(int j = 1 ; j <= m ; ++j) dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]) + a[i][j];
    cout << dp[n][m] << endl;

    return 0;
}
```

Mình có một chú ý với các bạn về đoạn code trên. Như ở trên ta đã phân tích, khi ô được xét là ở hàng trên cùng thì thực tế sẽ không có ô nào nằm trên nó cả. Tuy nhiên, trong đoạn code của mình, mình đã bỏ qua yếu tố này. Lí do là vì ta thấy ***dp[0][i]*** (kết quả ô nằm trên một ô ***(1, i)*** luôn là 0 nên sẽ không ảnh hưởng đến hàm max. Nếu như vì lí do nào đó mà ***dp[0][i]*** có ảnh hưởng đến kết quả bài toán thì các bạn sẽ cần xét riêng trường hợp này. Điều tương tự cũng xảy ra với ***dp[i][0]*** (kết quả ô bên trái một ô ***(i, 1)***.

Kết luận

Qua bài này chúng ta đã nắm về **Quy hoạch động**

Bài sau chúng ta sẽ tìm hiểu về **Một số bài toán điển hình về quy hoạch động**

Cảm ơn các bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thử thách – Không ngại khó**".