TẠP CHÍ KHOA HỌC TRƯỜNG ĐẠI HỌC HOA LƯ
ISSN 2615-9538
Website: http://hluv.edu.vn/vi/tckh

PHƯƠNG PHÁP QUY HOẠCH ĐỘNG GIẢI MỘT SỐ BÀI TOÁN CÓ TÍNH CHẤT QUY HỒI BẰNG NGÔN NGỮ LẬP TRÌNH C++

Phùng Thị Thao[1]

Ngày nhận bài: 21/9/2023 *Ngày chấp nhận đăng: 21/12/2023*

Tóm tắt: *Phương pháp quy hoạch động là một kỹ thuật hiệu quả để tối ưu hóa và giảm thiểu sự lặp lại việc tính toán, được sử dụng để giải quyết các bài toán có tính chất quy hồi.*

Trong bài báo này, tác giả giới thiệu phương pháp quy hoạch động và nhận diện một số đặc trưng cơ bản của các bài toán có thể giải bằng phương pháp quy hoạch động, các bước cài đặt để giải một bài toán bằng phương pháp quy hoạch động. Bài báo cũng phân tích một bài toán đặc trưng được giải theo phương pháp quy hoạch động, so sánh với phương pháp khác để chỉ ra ưu, nhược điểm của các phương pháp quy hoạch động được sử dụng. Đồng thời, bài báo đưa ra lời giải cụ thể cho một số bài toán, cài đặt bằng ngôn ngữ lập trình C++, cung cấp hàm sinh các bộ dữ liệu kiểm thử để kiểm chứng tính tối ưu của giải thuật.

Từ khóa: *Quy hoạch động, Bài toán quy hoạch động điển hình, Phương pháp quy hoạch động, Bài toán tối ưu*

DYNAMIC PLANNING METHOD AND INSTALLATION OF SOME PROBLEMS USING C++ PROGRAMMING LANGUAGE

Abstract: *The dynamic programming method is an effective technique for optimizing and minimizing computational repetition, used to solve regression problems.*

In this article, the author introduces the dynamic programming method and identifies some basic characteristics of the problem that can be solved using the dynamic programming method and the installation steps to solve the problem using the dynamic programming method. The article also analyzes some optimization problems solved by dynamic programming methods, comparing them with other methods to point out the advantages and disadvantages of the methods used. At the same time, the article provides specific solutions to problems implemented in the C++ programming language and provides functions to generate test data sets to verify the optimality of the algorithm.

Keywords: *Dynamic programming, Last data method, Typical dynamic programming problem*

1. ĐẶT VẤN ĐỀ

Dynamic programming (Quy hoạch động) được phát minh bởi nhà toán học nổi tiếng người Mỹ, Richard Bellman, vào những năm 1950 như một phương pháp chung để tối ưu hóa quá trình ra quyết định nhiều giai đoạn. Từ "programming" trong tên của kỹ thuật này là viết tắt của "lập kế hoạch" và không ám chỉ đến lập trình máy tính. [4].

[1] Trường PTTHSP Tràng An, Trường Đại học Hoa Lư; Email: ptthao@hluv.edu.vn

Phương pháp quy hoạch động là một kỹ thuật quan trọng trong lập trình máy tính được sử dụng để giải quyết các bài toán có tính chất quy hồi. Quy hoạch động giúp giải quyết các bài toán bằng cách chia chúng thành các bài toán con nhỏ hơn, lưu trữ và sử dụng lại kết quả của các bài toán con để tính toán kết quả cuối cùng. Đây là một phương pháp hiệu quả để tối ưu hóa và giảm thiểu sự lặp lại tính toán, đặc biệt đối với các bài toán có cấu trúc quy hồi.

Đã có nhiều nghiên cứu về phương pháp quy hoạch động trong nhiều lĩnh vực khác nhau như: "*Một số ứng dụng của quy hoạch động trong giải bài toán TSP*" - Trần Văn Thư, Phan Xuân Thịnh. (Kỷ yếu Hội thảo Tự nhiên khoa học và Công nghệ toàn quốc, 2009) giới thiệu việc sử dụng phương pháp quy hoạch động trong giải bài toán Con đường ngắn nhất (Traveling Salesman Problem - TSP). "*Quy hoạch động ứng dụng vào giải bài toán tìm dãy con chung dài nhất*" - Trần Tuấn Dũng, Đỗ Thị Thanh Minh, Vũ Thanh Lâm. (Kỷ yếu Hội thảo quốc gia về công nghệ thông tin và truyền thông, 2016) giới thiệu việc sử dụng phương pháp quy hoạch động để giải quyết bài toán tìm dãy con chung dài nhất (Longest Common Subsequence). "*Sử dụng phương pháp quy hoạch động trong giải bài toán lập lịch công việc*" - Trương Quốc Thắng, Nguyễn Thị Hà, Trần Thị Hương. (Hội nghị khoa học công nghệ trẻ lần thứ XII, 2013), …

Trong bài báo này, tác giả đã tổng hợp lý thuyết về phương pháp quy hoạch động từ nhiều nguồn khác nhau giúp tạo ra cách tiếp cận hợp nhất với quy hoạch động, phân tích bài toán Fibonaci đặc trưng theo 2 cách giải. Phần 3 trình bày cụ thể cài đặt trên ngôn ngữ lập trình C++ một số bài toán giải bằng phương pháp quy hoạch động, đồng thời cung cấp thêm các hàm sinh bộ dữ liệu kiểm thử để kiểm tra tính tối ưu của mỗi bài toán này.

2. NỘI DUNG

2.1. Phương pháp giải bài toán quy hoạch động

*** Đặc trưng của các bài toán giải bằng phương pháp quy hoạch động** [3]

Các bài toán giải bằng phương pháp quy hoạch động thường có các đặc trưng chung là chúng có cấu trúc quy hồi và tổng hợp kết quả từ các bài toán con thành kết quả của bài toán gốc. Phương pháp quy hoạch động thường được sử dụng để giải các bài toán có dạng:

- Bài toán có các bài toán con gối nhau.
- Bài toán có cấu trúc con tối ưu.

Cấu trúc con tối ưu: Các bài toán quy hoạch động thể hiện tính chất của cấu trúc con tối ưu, có nghĩa là lời giải tối ưu cho bài toán có thể được xây dựng từ lời giải tối ưu của các bài toán con của nó. Nói cách khác, việc chia bài toán thành các bài toán con nhỏ hơn sẽ dẫn đến giải pháp tối ưu cho toàn bộ bài toán.

Ví dụ: Trong bài toán tìm đường đi ngắn nhất trên đồ thị, nếu một node x nằm trên đường đi ngắn nhất giữa hai node u, v thì đường đi ngắn nhất từ u đến v sẽ là tổng hợp của đường đi ngắn nhất từ u đến x và đường đi ngắn nhất từ x đến v. Một số thuật toán tìm đường trên đồ thị (ví dụ thuật toán Dijkstra) đều dựa trên tính chất này, và nó được áp dụng trong quy hoạch động.

Tính chất cấu trúc con tối ưu rất quan trọng, nó cho phép chúng ta giải bài toán lớn dựa vào các bài toán con đã giải được. Nếu không có tính chất này, chúng ta không thể áp dụng quy hoạch động được.

Các bài toán con gối nhau: Nhiều bài toán quy hoạch động có các bài toán con gối nhau, nghĩa là các bài toán con giống nhau được giải nhiều lần. Để tránh tính toán dư thừa, các phương pháp lập trình động lưu trữ và sử dụng lại kết quả của các bài toán con, thường sử dụng các cấu trúc dữ liệu như mảng hoặc bảng để lưu kết quả. Một ví dụ điển hình của bài toán con gối nhau là bài toán tính số Fibonacci thứ n.

Quy hoạch động sẽ không thể áp dụng được (hoặc nói đúng hơn là áp dụng cũng không có tác dụng gì) khi các bài toán con không gối nhau. Ví dụ với thuật toán tìm kiếm nhị phân, quy hoạch động cũng không thể tối ưu được gì cả, bởi vì mỗi khi chia nhỏ bài toán lớn thành các bài toán con, mỗi bài toán cũng chỉ cần giải một lần mà không bao giờ được gọi lại.

*** Phương pháp giải các bài toán quy hoạch động:** Có 2 cách tiếp cận là Top-Down và Bottom-Up [3]

Top-Down (Từ trên xuống - Ghi nhớ): Trong cách tiếp cận từ trên xuống, còn được gọi là ghi nhớ, ta bắt đầu với bài toán ban đầu và chia nó thành các bài toán con nhỏ hơn theo cách đệ quy. Tuy nhiên, ta phải lưu trữ kết quả của các bài toán con trong cấu trúc dữ liệu (thường là mảng hoặc bảng băm) để tránh các phép tính dư thừa. Khi gặp một bài toán con đã được giải, ta lấy kết quả của nó từ cấu trúc dữ liệu thay vì tính toán lại nó.

Bottom-Up (Từ dưới lên - Lập bảng): Trong cách tiếp cận từ dưới lên, còn được gọi là lập bảng, ta phải giải các bài toán con lặp đi lặp lại, bắt đầu từ các bài toán con nhỏ nhất và dần dần phát triển đến bài toán ban đầu. Ta sử dụng một bảng hoặc mảng để lưu trữ kết quả của các bài toán con và điền vào bảng một cách có trình tự để đảm bảo rằng mỗi bài toán con đều được giải trước khi sử dụng kết quả của nó để giải các bài toán con lớn hơn.

*** Quy trình cài đặt của phương pháp quy hoạch động có thể được mô tả như sau:**

1. **Xác định cấu trúc quy hồi:** Điều này bao gồm việc xác định các bước hoặc các phần con trong bài toán mà có thể được giải quyết độc lập và có cấu trúc quy hồi.
2. **Xác định hàm quy hoạch động:** Sau khi xác định cấu trúc quy hồi, ta sẽ tạo một hàm quy hoạch động để tính toán và lưu trữ kết quả của các bài toán con. Hàm quy hoạch động này thường sử dụng một mảng hoặc bảng để lưu trữ kết quả tại mỗi bước.
3. **Thiết lập điểm khởi đầu**: Đặt giá trị cho các bài toán con đơn giản nhất hoặc giải các bài toán cơ sở. Điều này thường được thực hiện bằng cách gán giá trị trực tiếp vào mảng hoặc bảng lưu trữ kết quả.
4. **Duyệt qua các bài toán con**: Bắt đầu từ các bài toán cơ sở, ta sẽ duyệt qua từng bài toán con và tính toán kết quả của nó. Kết quả này sau đó được lưu trữ trong mảng hoặc bảng.
5. **Xây dựng giải pháp cuối cùng**: Sau khi đã tính toán kết quả của tất cả các bài toán con, ta sẽ xây dựng giải pháp cuối cùng cho bài toán lớn bằng cách kết hợp kết quả từ các bài toán con. Kết quả này thường nằm ở vị trí cuối cùng trong mảng hoặc bảng lưu trữ.
6. **Truy vấn giải pháp**: truy vấn giải pháp tối ưu tại vị trí cụ thể trong mảng hoặc bảng, nơi chứa kết quả của bài toán ban đầu.

Để biết được bài toán có thể giải bằng quy hoạch động hay không, ta có thể tự đặt câu hỏi: "***Một nghiệm tối ưu của bài toán lớn có phải là sự phối hợp các nghiệm tối ưu của các bài toán con hay không?"*** và ***"Liệu có thể nào lưu trữ được nghiệm các bài toán con dưới một hình thức nào đó để phối hợp tìm được nghiệm bài toán lớn?"***. Nếu câu trả lời là có thì bài toán đó hoàn toàn có thể giải bằng phương pháp quy hoạch động [5].

2.2. Phân tích và cài đặt bài toán Fibonaci thứ n bằng 2 phương pháp

Bài toán: Dãy Fibonacci là dãy số nguyên dương được định nghĩa như sau:

$F_1 = F_2 = 1$;

$F_i = F_{i-1} + F_{i-2}$ với mọi $i \geq 3$

Hãy tính F_n

Phân tích bài toán [4]

Nếu chúng ta cố gắng sử dụng trực tiếp phép truy hồi F (n) = F (n-1) + F (n-2) để tính số Fibonacci thứ n F (n), chúng ta sẽ phải tính lại các giá trị giống nhau của hàm này nhiều lần. Lưu ý rằng bài toán tính F (n) được thể hiện dưới dạng các bài toán con nhỏ hơn và gối nhau của nó về tính F (n − 1) và F (n − 2). Vì vậy, ta có thể chỉ cần điền các phần tử của mảng một chiều với n + 1 giá trị liên tiếp của F (n) bằng cách bắt đầu, theo điều kiện ban đầu bằng 0 và 1, và sử dụng phương trình F (n) = F (n-1) + F (n-2) làm quy luật để tìm ra tất cả các phần tử khác. Hiển nhiên, phần tử cuối cùng của mảng này sẽ chứa F(n)

Xét hai cách cài đặt chương trình:

Cách 1 - Sử dụng đệ quy:

int F(int i)

```
{
  int res=0;
  if (i < 3) res= 1;
    else res = F(i - 1) + F(i - 2);
    return res;
}
```

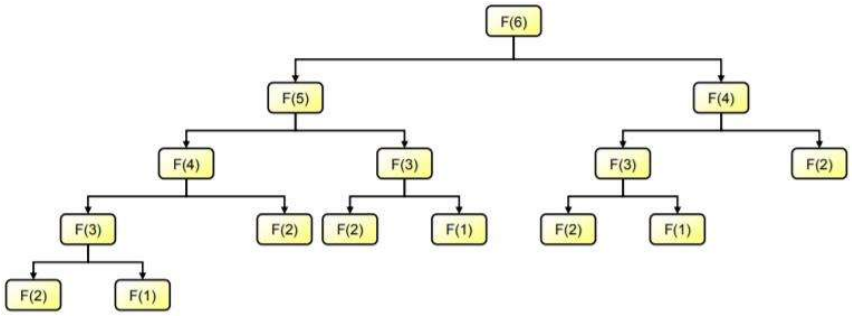
Trong phương pháp đệ quy, ta sử dụng định nghĩa trực tiếp của dãy Fibonacci để tính Fibonacci(n).

Ta kiểm tra nếu n bằng 0 hoặc 1, thì trả về n.

Nếu n lớn hơn 1, ta gọi đệ quy để tính Fibonacci(n-1) và Fibonacci(n-2), sau đó cộng chúng lại và trả về kết quả.

Hàm đệ quy F(i) để tính số Fibonacci thứ i. Chương trình chính gọi F(6), nó sẽ gọi tiếp F(5) và F(4) để tính … Quá trình tính toán có thể vẽ như cây dưới đây. Ta nhận thấy để tính F(6) nó phải tính 1 lần F(5), hai lần F(4), ba lần F(3), năm lần F(2), ba lần F(1) [5].

Tuy cách giải này đơn giản, nhưng nó có độ phức tạp thời gian là O(2^n) do việc tính toán trùng lặp nhiều lần. Điều này làm cho thuật toán trở nên không hiệu quả khi n lớn.



Hình 2.1. Hàm đệ quy tính số Fibonacci

Cách 2 - Sử dụng phương pháp quy hoạch động:

Đặc trưng của bài toán là các bài toán con gối nhau, muốn tìm f_n ta phải tìm f_{n-1} và f_{n-2}, muốn tìm f_{n-1} ta phải tìm f_{n-2} và f_{n-3} cứ như vậy đến f_0

Ta xác định được:

- Bài toán cơ sở là $f_0 = f_1 = 1$;
- Công thức truy hồi: $f_i = f_{i-1} + f_{i-2}$
- Bảng phương án ta dùng một mảng F[] để lưu các giá trị
- Nghiệm của bài toán chính là F[n]

Ta sử dụng phương pháp quy hoạch động theo các bước như sau:

1. Sử dụng một bảng (hoặc mảng) để lưu trữ các giá trị Fibonacci đã tính để tránh tính toán trùng lặp.
2. Khởi tạo một mảng (thường là mảng 1 chiều) với ít nhất n+1 phần tử (để có đủ chỗ lưu Fibonacci(n)).
3. Sử dụng một vòng lặp để tính toán lần lượt các giá trị Fibonacci từ 0 đến n bằng cách sử dụng các giá trị đã tính trước đó.
4. Trả về Fibonacci(n) sau khi tính xong

```
Int fb2(int n)
{
   f[1] = f[2] =1;
   for(int i=3; i<=n; i++) f[i] = f[i-1] +f[i-2];
 }
```

Phương pháp quy hoạch động này có độ phức tạp thời gian là O(n) do chỉ tính toán mỗi giá trị Fibonacci một lần và lưu trữ kết quả trong mảng.

**** Ưu điểm của phương pháp Quy hoạch động trong giải bài toán Fibonaci:***

Hiệu suất tối ưu: Phương pháp quy hoạch động giảm thiểu tính toán lặp lại, bằng cách lưu trữ các giá trị Fibonacci đã tính toán trước đó trong một mảng hoặc bảng. Khi cần tính toán một số Fibonacci, ta có thể truy cập trực tiếp giá trị đã tính mà không cần tính lại từ đầu.

Độ phức tạp thời gian thấp: bài toán Fibonaci nếu giải theo phương pháp đệ quy (cách 1) thì độ phức tạp tính toán khá cao (có thể lên đến 2^{n-1}). Với độ lớn của n từ hơn 100 thì việc tính toán mất nhiều thời gian mới ra kết quả và không thể đáp ứng yêu cầu về mặt tốc độ tính toán, bộ nhớ lưu trữ. Phương pháp quy hoạch động có độ phức tạp thời gian O(n), trong đó n là số Fibonacci cần muốn tính. Điều này có nghĩa là việc tính toán số Fibonacci thứ n chỉ cần một lần duyệt qua mảng chứa các giá trị đã tính. Đối với các số Fibonacci lớn, phương pháp này cực kỳ hiệu quả. Tuy nhiên, cũng cần lưu ý rằng phương pháp quy hoạch động sẽ tiêu tốn một lượng bộ nhớ để lưu trữ kết quả của các bài toán con.

2.3. Cài đặt một số bài toán mới theo phương pháp quy hoạch động bằng ngôn ngữ lập trình C++

2.3.1. Cơ sở toán học

A/ Phương pháp quy nạp

"*Nếu một mệnh đề toán học P(n) phụ thuộc biến số tự nhiên n là đúng với giá trị cơ sở n = a, ngoài ra khi P(k) đúng kéo theo P(k+1) đúng thì mệnh đề P(n) đúng với mọi số tự nhiên n không nhỏ hơn a".*

Điều quan trọng nhất được áp dụng ở đây không phải dùng phương pháp quy nạp chứng minh bài toán mà chính là cách chứng minh P(k+1) đúng. Để làm điều này, trong phương pháp quy nạp người ta đã vận dụng thật sáng tạo cơ sở đã có đó là P(n) đúng với các n không quá k. Việc này hoàn toàn tương tự khi chúng ta phân tích xây dựng x_k theo các giá trị đã có trước đó trong cấu hình đang xây dựng của bài toán [1].

B/ Công thức truy hồi

Có hai dạng công thức truy hồi chúng ta sẽ sử dụng:

1/ Một dãy số hoàn toàn xác định khi biết k giá trị đầu tiên và công thức xác định một số hạng theo k số hạng liền trước nó:

{Fn} xác định nếu biết $F_1, F_2,...,F_k$ và công thức $F_n = F(F_{n-1},F_{n-2},...,F_{n-k})$.

2/ Một dãy số hoàn toàn xác định khi biết giá trị đầu tiên và công thức xác định một số hạng theo các số hạng trước nó:

{Fn} xác định nếu biết F_1 và công thức $F_n = F(F_{n-1},F_{n-2},...,F_1)$.

Việc tính các giá trị ban đầu $F_1, F_2, ..., F_k$ gọi là bước cơ sở, việc tìm công thức cho phép tính một số hạng của dãy theo các số hạng trước nó gọi là bước quy nạp [1].

2.3.2. Các bài toán ứng dụng phương pháp quy hoạch động

Sau đây là một số bài toán được chia theo các dạng *Đếm các xâu, các cấu hình, các dãy số thỏa mãn điều kiện nào đó* để thấy được tác dụng phong phú của việc giải bài toán tin học bằng phương pháp quy hoạch động.

Bài toán 1: ĐẾM XÂU NHỊ PHÂN

Cho số nguyên dương n. Tính số xâu nhị phân độ dài n không có 2 chữ số 1 liền nhau.

- Dữ liệu: vào từ file binary.inp gồm một số nguyên dương n
- Kết quả: ghi ra file binary.out một số duy nhất là kết quả bài toán, vì kết quả có thể là một số rất lớn nên ta lấy kết quả là phần dư cho $10^9 + 7$

Ví dụ

Binary.inp	Binary.out
4	8

Xây dựng giải thuật:

Từ nhận xét: Một xâu độ dài n thỏa mãn yêu cầu bài toán (không có 2 ký tự 1 liên tiếp) thì xâu con n-1 ký tự đầu tiên của nó cũng thỏa mãn điều này, thế có nghĩa là để có xâu độ dài n ta chỉ việc thêm 0 hoặc 1 vào sau xâu độ dài n-1 như vậy mỗi đơn vị dữ liệu là một ký tự và *đơn vị dữ liệu cuối* ở bước thứ k là ký tự thứ k của xâu. Từ đó bằng phương pháp phân tích quy nạp, ta có thể xây dựng công thức quy hoạch động cho phép tính số xâu độ dài n theo số xâu có độ dài nhỏ hơn cùng thỏa mãn yêu cầu bài toán.

Gọi F[k] là số xâu nhị phân độ dài k không có 2 chữ số 1 liền nhau.

Bước cơ sở: n = 1: có 2 xâu thoả mãn là '0' và '1' nên F[1]:=2.

n = 2: có 3 xâu thoả mãn là '00';'01'và '10' nên F[2]:=3.

Bước quy nạp: Giả sử đã tìm được các F[i] với i:=1,2…,k-1. Ta tính F[k] – số xâu độ dài k thỏa mãn bài toán có được do:

+ Thêm 0 vào sau 1 xâu độ dài k – 1, số xâu loại này bằng số xâu độ dài k-1 không có 2 chữ số 1 liền nhau và bằng F[k-1].

+ Thêm 1 vào sau 1 xâu độ dài k – 1, để tạo xâu độ dài k thoả mãn yêu cầu thì ký tự cuối cùng xâu độ dài k-1 phải là 0, do đó số xâu loại này bằng số xâu độ dài k-2 không có 2 chữ số 1 liền nhau và bằng F[k-2].

Vậy F[k] = F[k – 1] + F[k – 2] (1)

Do đã biết F[1], F[2] nên từ công thức này ta có thể tính F[n] với n tùy ý.

Độ phức tạp: do ta mất một vòng for để tính F[i] nên độ phức tạp là O(n).

Cài đặt

```
#include <bits/stdc++.h>
#define mod 1000000007
using namespace std;
long long f[100005]; int n;
int main()
{
   //sinh();
   freopen("binary.inp","r",stdin);
   freopen("binary.out","w",stdout);
   cin>>n;
   f[1]=2;
   f[2]=3;
   for(int i=3; i<=n; i++) f[i] = (f[i-1]+f[i-2])%mod;
   cout<<f[n];
   return 0;
}
```

Hàm sinh test:

```
void sinh(){
   srand(time(0));
   freopen("binary.inp","w",stdout);
   n =rand()%100000+2;
   cout<<n;
}
```

Bài toán 2: ĐẾM XÂU NHỊ PHÂN MỞ RỘNG

Tương tự như trên, ta xây dựng công thức tính số xâu nhị phân độ dài n không có k chữ số 1 liền nhau (k>=2).

Ở đây ta cũng gọi F[n] để giữ giá trị số xâu nhị phân không có k chữ số 1 liền nhau và có độ dài n.

Bước cơ sở: Ta thấy F[0]:=1; F[i]:=2^i với i:=1,2,..k-1.

Với i:=k, F[k] = 2^k – 1 (trừ xâu có k chữ số 1)

Bước quy nạp:

+ Thêm 0 vào sau 1 xâu độ dài n – 1, số xâu loại này bằng số xâu độ dài n-1 không có 2 chữ số 1 liền nhau và bằng F[n-1].

+ Thêm 1 vào sau 1 xâu độ dài n – 1 cũng tạo thành F[n-1] xâu nhị phân mới có độ dài n nhưng trong đó có một số xâu không thoả mãn vì k chữ số liên tiếp cuối cùng bằng 1. Số xâu sau khi thêm 1 có k chữ số cuối bằng 1 bằng số xâu độ dài n-1 thỏa mãn bài toán và có k-1 chữ số cuối cùng là 1 : $x_1x_2..x_{n-k-1}x_{n-k}11\ldots1$, khi đó x_{n-k} = 0 và vì vậy số xâu loại này bằng số xâu $x_1x_2..x_{n-k-1}$ thỏa mãn bài toán và bằng F[n-k-1], từ đó số xâu độ dài n tận cùng bằng 1 thỏa mãn bài toán là: F[n-1] – F[n – k – 1] .

Vậy Ta có công thức quy hoạch động: F[n]:=2*F[n-1]-F[n-k-1].

Độ phức tạp: O(n) do có n vòng for.

Cài đặt chương trình

```
#include <bits/stdc++.h>
#define mod 1000000007
using namespace std;
long long f[100005]; int n,k;
int main()
{
   //sinh();
   freopen("binaryex.inp","r",stdin);
   freopen("binaryex.out","w",stdout);
   cin>>n>>k;
   f[0]=1;
   for(int i=1; i<k; i++) f[i]=f[i-1]*2;
   f[k]=f[k-1]*2 -1;
   for(int i=k+1; i<=n; i++) f[i] = (2*f[i-1]%mod-f[i-k-1])%mod;
   cout<<f[n];
   return 0;
}
```

Hàm sinh test

```
void sinh(){
   srand(time(0));
   freopen("binaryex.inp","w",stdout);
   n =rand()%100000+2;
   k =rand()%100000+2;
   if(n<k) swap(n,k);
   cout<<n<<" "<<k;
}
```

Bài toán 3: ĐẾM SỐ DÃY CON TĂNG DÀI NHẤT

(Mở rộng của bài toán Tìm dãy con đơn điệu tăng dài nhất)

Cho dãy số nguyên A = a_1, a_2, ..., a_n. (n ≤ 1000, -10000 ≤ a_i ≤ 10000). Một dãy con của A là một cách chọn ra trong A một số phần tử giữ nguyên thứ tự. Như vậy A có 2^n dãy con.

Yêu cầu: Tìm số lượng dãy con đơn điệu tăng của A có độ dài lớn nhất.

Dữ liệu:

- Dòng đầu là số n
- Dòng thứ 2 chứa n số nguyên

Kết quả:

- Dòng đầu là độ dài dãy con tăng dài nhất
- Dòng thứ 2 là số lượng dãy con tăng dài nhất

Ví dụ: A = (1, 2, 3, 4, 9, 10,11, 13, 5, 6, 7, 8). Dãy con đơn điệu tăng dài nhất là: (1, 2, 3, 4, 5, 6, 7, 8) hoặc (1, 2, 3, 4, 9, 10, 11, 13).

Ví dụ:

Cis.inp	Cis.out
12 1 2 3 4 9 10 11 13 5 6 7 8	8 2

Giải quyết bài toán mở rộng này có ý nghĩa sâu sắc trong việc rèn luyện tư duy quy hoạch động cho học sinh. Ở đây ta có nhận xét quan trọng rằng: a_{i1}, a_{i2}, ..., a_{ik} là dãy con tăng dài nhất thì mỗi đoạn đầu liên tiếp của nó: a_{i1}, a_{i2}, ..., a_{ip} (p<=k) sẽ là dãy con tăng dài nhất có số hạng cuối là a_{ip} như vậy có nghĩa p là độ dài dãy con tăng này nên nó phải bằng F[ip] (mảng F nói trên). Từ đó ta sẽ tính số dãy con tăng T[i] có độ dài F[i] và số hạng cuối cùng là a_i. Rõ ràng a_i lúc này chỉ được thêm vào sau số hạng a_j bé hơn nó mà F[j] +1 =F[i]. Khi ấy tập các dãy con tăng độ dài F[i] kết thúc tại a[i] được bổ xung thêm T[j] phần tử. Vậy ở đây ta có công thức quy hoạch động như sau:

T[i] = Tổng các T[j] với j thỏa mãn: j < i; a[j] < a[i]; F[j] + 1 = F[i].

điều này có nghĩa là để tính T[i] ta phải tính trước các F[j], tất nhiên việc này sẽ kết hợp tính song song trong cùng một vòng lặp.

Kết quả cuối cùng là tổng các T[i] mà F[i] = max{F[j] : j = 1,2,...,n}.

Độ phức tạp: $O(n^2)$

Cài đặt chương trình

```
#include <bits/stdc++.h>
#define maxn 1000
using namespace std;
int a[maxn+5], f[maxn+5], t[maxn+5], n, res=0,c=0;
int main()
{
   //sinh();
   freopen("cis.inp","r",stdin);   freopen("cis.out","w",stdout);
   cin>>n;
   for(int i=1; i<=n; i++){
      cin>>a[i];      f[i]=1;      t[i]=1;
   }
   for(int i=1; i<=n; i++){
      for(int j=i-1; j>=1; j--){
         if(a[j]<a[i]){
            if(f[j]+1>f[i]){f[i]=f[j]+1;t[i]=t[j];}
            else if(f[j]+1==f[i]) t[i]+=t[j];
         }
      }
      res=max(res,f[i]);
   }
```

```
    for(int i=1; i<=n; i++) if(f[i]==res) c+=t[i];
    cout<<res<<endl<<c;
    return 0;
}
```

Hàm sinh test

```
void sinh(){
    srand(time(0));
    freopen("cis.inp","w",stdout);
    n = 1000;
    cout<<n<<endl;
    for(int i=1; i<=n; i++) cout<<rand()%100000+1<<" ";
}
```

Với cách cài đặt như trên ta có một thuật toán với độ phức tạp $O(n^2)$.

*** Mở rộng:** Bài toán "***Đếm số dãy con tăng dài nhất***" (hoặc bài toán "***Tìm dãy con đơn điệu tăng dài nhất***") có thể sử dụng thêm tìm kiếm nhị phân hoặc sử dụng cấu trúc dữ liệu Cây chỉ số nhị phân (**BIT**) để tính F[i] với độ phức tạp O(n*log n) – Ví dụ khi sử dụng cây **BIT**, để tìm vị trí j thỏa mãn $j<i$, $a_j<a_i$ mà f[j] đạt max, ta chỉ mất độ phức tạp O(log n).

Cài đặt chương trình

```
#include <bits/stdc++.h>
#define maxn 100005
using namespace std;
typedef long long lint;
int n, a[maxn], b[maxn];
lint f[maxn], t[maxn], bit[maxn];
lint get(int u) {
   lint kq=bit[u];
   while (u) {
      kq=max(kq,bit[u]);
      u&=(u-1);
   }
   return kq;
}
void update(int u,lint val) {
   while (u<=100000) {
      bit[u]=max(bit[u],val);
      u+=u & (-u);
   }
}
int main() {
   freopen("cis.inp","r",stdin);
   freopen("cis.out","w",stdout);
   scanf("%d",&n);
   for(int i=1;i<=n;i++) scanf("%d",&a[i]);
   for(int i=1;i<=n;i++) {
      f[i]=get(a[i]-1)+1;
      update(a[i],f[i]);
   }
```

```
    long long maxf=f[1];
    for(int i=1;i<=n;i++) maxf=max(maxf,f[i]);
        for(int i=1; i<=n; i++)
        for(int j=1; j<i; j++){
            if(a[j]<a[i]&&f[j]+1==f[i])
            {
                if(t[i]==0) t[i]=1;
                else t[i]+=t[j];
            }
        }
    long long res=0;
    for(int i=1;i<=n;i++)
       if(f[i]==maxf) res+=t[i];
    cout<<maxf<<endl<<res;
}
```

Việc sử dụng cấu trúc dữ liệu Cây chỉ số nhị phân (**BIT**) sẽ cải tiến được tốc độ của bài toán Tìm dãy con đơn điệu tăng dài nhất (còn O (n*log n)).

Tuy nhiên trong ***Bài toán 3 (Đếm số lượng dãy còn tăng dài nhất)***, để tính số lượng dãy con tăng dài nhất T[i] thì ta vẫn mất thêm hai vòng lặp nữa để tính nên độ phức tạp vẫn là $O(n^2)$ giống như khi sử dụng phương pháp quy hoạch động đơn thuần.

3. KẾT LUẬN

Bài báo đã trình bày được các kiến thức tổng quan về phương pháp quy hoạch động, đồng thời thử nghiệm cài đặt một số bài toán bằng ngôn ngữ lập trình C++, viết hàm sinh bộ dữ liệu kiểm thử để kiểm tra tính chính xác và tối ưu của giải thuật. Bài báo giúp tạo ra một tiếp cận hợp nhất đối với quy hoạch động, góp phần tạo ra nguồn tư liệu tham khảo cho giáo viên, học sinh, sinh viên, và những người quan tâm đến quy hoạch động. Đồng thời mở ra hướng nghiên cứu sâu hơn về việc kết hợp quy hoạch động với các kĩ thuật và cấu trúc dữ liệu mới để cải thiện tốc độ tính toán khi giải quyết bài toán trong tin học.

TÀI LIỆU THAM KHẢO

[1] Hồ Sĩ Đàm, *Tài liệu giáo khoa chuyên tin quyển 1*, NXB Giáo dục Việt Nam, 2009.

[2] Lê Minh Hoàng, Ebook *Giải thuật và lập trình*, Đại học sư phạm Hà Nội, 2002.

[3] Trần Ngọc Anh, https://topdev.vn/blog/thuat-toan-quy-hoach-dong/

[4] Anany Levitin (2002), Ebook "*Introduction to the Design and Analysis of Algorithms*"

[5] https://v1study.com/giai-thuat-va-lap-trinh-phuong-phap-quy-hoach-dong.html