1. Tổng quan về quy hoạch động

Quy hoạch động là một kỹ thuật để giải quyết các vấn đề với các bài toán con chồng chéo. Thông thường, các bài toán con này được phát sinh từ phép lặp trong lời giải của bài toán đã cho. Thay vì giải quyết lặp đi lặp lại các bài toán con chồng chéo, quy hoạch động gợi ý chỉ giải quyết từng bài toán con nhỏ hơn một lần và ghi lại kết quả vào một bảng để từ đó có thể thu được lời giải cho bài toán ban đầu. Kỹ thuật này có thể được minh họa qua bài toán dãy số Fibonacci.

Các số Fibonacci là các phần tử của chuỗi

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...,

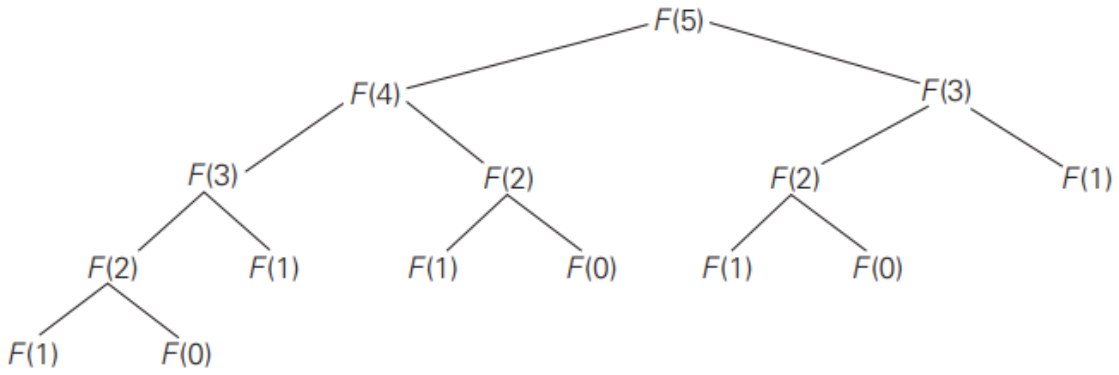
có thể được định nghĩa bởi công thức đệ quy đơn giản

$$F(n) = F(n-1) + F(n-2) \text{ với } n > 1 \qquad (\mathbf{1})$$

và hai điều kiện ban đầu

$$F(0) = 0, F(1) = 1. \qquad (\mathbf{2})$$

Nếu cố gắng sử dụng quan hệ truy hồi (**1**) trực tiếp để tính toán số Fibonacci thứ n, ta sẽ phải tính lại các giá trị của hàm này nhiều lần.



Hình 1. Cây đệ quy khi thực hiện tính toán $F(5)$ bằng công thức (**1**).

Quan sát rằng vấn đề khi tính toán $F(n)$ có thể được biểu diễn dưới dạng các bài toán con nhỏ hơn và chồng chéo của nó qua công thức $F(n-1)$ và $F(n-2)$. Vì vậy, ta có thể đơn giản điền vào một mảng một chiều $n+1$ giá trị liên tiếp của $F(n)$, với điều kiện ban đầu (**2**) và sử dụng công thức (**1**) là quy tắc để tạo ra tất cả các phần tử khác. Rõ ràng, phần tử cuối cùng của mảng này sẽ chứa $F(n)$.

THUẬT TOÁN *Fib* (n)

// Tính toán số Fibonacci thứ n bằng phương pháp đệ quy thông qua định nghĩa

// Input: Một số nguyên không âm n

// Output: Số Fibonacci thứ n

if $n \leq 1$ **return** n

else return $Fib\ (n-1) + Fib\ (n-2)$

Thực tế, chúng ta có thể tránh sử dụng mảng phụ bằng cách chỉ ghi lại hai giá trị cuối cùng của chuỗi Fibonacci. Khi đó độ phức tạp không gian của thuật toán $Fib\ (n)$ chỉ còn $\Theta(1)$. Do đó, mặc dù có thể xem ứng dụng trực tiếp của quy hoạch động như một loại đánh đổi không gian lấy thời gian, một thuật toán quy hoạch động đôi khi có thể được tinh chỉnh để tránh sử dụng không gian bổ sung.

Một số thuật toán tính số Fibonacci thứ n mà không cần tính toàn bộ các phần tử trước trong dãy. Những thuật toán này dựa trên phương pháp bottom-up cổ điển, thường phải giải quyết *tất cả* các bài toán con nhỏ hơn của một bài toán cụ thể. Một biến thể của phương pháp quy hoạch động khai thác các hàm bộ nhớ, được gọi là phương pháp top-down, nhằm tránh giải quyết các bài toán con không cần thiết.

Cho dù sử dụng phiên bản bottom-up cổ điển của quy hoạch động hay biến thể top-down, bước quan trọng trong thiết kế một thuật toán như vậy vẫn giống nhau: suy ra một quan hệ truy hồi giải quyết bài toán với các giải pháp cho các bài toán con nhỏ hơn của nó. Sự sẵn có ngay của công thức (**1**) để tính toán số Fibonacci thứ n là một trong số ít các ngoại lệ của quy tắc này.

2. Các phương pháp chính để triển khai quy hoạch động:

Có hai phương pháp chính để triển khai quy hoạch động là bottom-up và top-down.

Phương pháp Bottom-up:

Phương pháp bottom-up bắt đầu giải quyết các bài toán con đơn giản hơn trước, sau đó lưu trữ giá trị của chúng và sử dụng để giải quyết các bài toán con phức tạp hơn cho đến khi giải quyết được bài toán chính. Kỹ thuật này được gọi là "bottom-up" bởi vì nó bắt đầu từ các bài toán con ở đáy, và tính toán dần lên đến bài toán lớn hơn ở đỉnh.

Cụ thể, để triển khai phương pháp bottom-up, ta thực hiện các bước sau:

1. Xác định bài toán con cơ bản dùng để giải các bài toán lớn hơn.
2. Định nghĩa các trường hợp cơ bản có thể giải mà không cần phân tích thêm.
3. Tạo một bảng để lưu trữ các giá trị đã tính toán.
4. Giải quyết các bài toán con đơn giản trước và lưu giữ kết quả của chúng trong bảng.
5. Sử dụng các giá trị đã lưu trữ trong bảng để giải quyết các bài toán con phức tạp hơn và lưu trữ kết quả vào bảng.
6. Lấy giá trị cuối cùng của bảng để trả về kết quả của bài toán chính.

Ưu điểm:

- Độ phức tạp tính toán thường thấp hơn phương pháp top-down vì không cần gọi đệ quy.
- Tiết kiệm bộ nhớ do không cần lưu trữ kết quả các bài toán con.
- Không gặp vấn đề stack overflow.

Nhược điểm:

- Cần phải đưa ra quan hệ truy hồi hoặc dãy chuyển tiếp rõ ràng.
- Không thể sử dụng cho các bài toán có cấu trúc đệ quy phức tạp.

Phương pháp Top-down:

Phương pháp top-down được gọi là phương pháp đệ quy với việc sử dụng memoization. Nó bắt đầu từ bài toán lớn và giải quyết nó bằng cách sử dụng kết quả của các bài toán con (tương tự như bottom-up). Tuy nhiên, top-down sử dụng hàm đệ quy để giải quyết các bài toán con và lưu trữ kết quả của chúng trong một bảng gọi là bảng memoization để tránh việc tính toán lại các bài toán con đã được giải quyết.

Để triển khai phương pháp top-down, ta thực hiện các bước sau:

1. Xác định bài toán con cơ bản dùng để giải các bài toán lớn hơn.
2. Định nghĩa các trường hợp cơ bản có thể giải mà không cần phân tích thêm.
3. Tạo một bảng để lưu trữ các giá trị đã tính toán.
4. Viết hàm đệ quy để giải quyết bài toán chính. Hàm đệ quy này sẽ gọi đến chính nó để giải quyết các bài toán con, và trả về giá trị của bài toán chính.
5. Trước khi giải quyết một bài toán con, ta kiểm tra xem giá trị của nó đã được tính toán trước đó chưa. Nếu đã tính toán rồi, ta sử dụng giá trị đã tính toán đó để tránh tính toán lại. Nếu chưa chưa có trong bảng, hàm đệ quy tính toán và lưu trữ kết quả vào bản.
6. Hàm đệ quy trả về giá trị của bài toán chính.

Ưu điểm:

- Có thể giải quyết các bài toán có cấu trúc đệ quy phức tạp.
- Dễ hiểu và dễ triển khai.

Nhược điểm:

- Tốn bộ nhớ do phải lưu trữ kết quả của các bài toán con.
- Có thể gặp vấn đề stack overflow nếu số lượng bài toán con quá lớn hoặc độ sâu của đệ quy quá sâu.

Nhìn chung, phương pháp bottom-up có xu hướng nhanh hơn và sử dụng ít bộ nhớ hơn phương pháp top-down bởi vì nó tránh được chi phí hoạt động của đệ quy và ghi nhớ lệnh gọi hàm. Tuy nhiên, phương pháp top-down có thể thuận tiện hơn và dễ thực hiện hơn đối với một số bài toán nhất định, đặc biệt là những bài toán có cấu trúc phức tạp và độ sâu đệ quy không quá lớn.

3. Ba ví dụ cơ bản của quy hoạch động:

Mục tiêu của phần này là giới thiệu quy hoạch động thông qua ba ví dụ điển hình.

VÍ DỤ 1: *Bài toán dãy tiền xu*

Cho một hàng gồm n đồng xu với giá trị tương ứng là $c_1, c_2, \ldots, c_n$ (không nhất thiết phải khác nhau). Mục tiêu là chọn được số tiền lớn nhất có thể, với điều kiện không chọn hai đồng xu liền kề.

Gọi $F(n)$ là số tiền lớn nhất có thể chọn được từ n đồng xu. Để suy ra một quan hệ truy hồi cho $F(n)$, ta chia tất cả các cách chọn tiền hợp lệ thành hai nhóm: nhóm chứa đồng xu cuối cùng và nhóm không chứa đồng xu cuối cùng. Ta có thể lấy được số tiền lớn nhất từ nhóm thứ nhất bằng cách chọn đồng xu cuối cùng và lấy số tiền lớn nhất có thể từ $n - 2$ đồng xu đầu tiên, tức là $c_n + F(n - 2)$. Số tiền lớn nhất có thể lấy được từ nhóm thứ hai bằng $F(n - 1)$, có được thông qua định nghĩa của $F(n)$. Do đó, ta có quan hệ truy hồi sau đây với điều kiện ban đầu:

$$F(n) = \max \{c_n + F(n - 2), F(n - 1)\} \text{ với } n > 1,$$

$$F(0) = 0, F(1) = c_1. \qquad (\mathbf{3})$$

Ta có thể tính F (n) bằng cách điền vào bảng một hàng từ trái sang phải, tương tự như cách tính số Fibonacci thứ n bằng thuật toán $Fib(n)$ trong phần trước.

THUẬT TOÁN *CoinRow*(C [1..n])

// Áp dụng công thức (**3**) theo phương pháp bottom-up để tìm số tiền lớn nhất có

// thể chọn được từ dãy tiền xu mà không lấy hai đồng xu liền kề

// Input: Mảng C [1..n] gồm các số nguyên dương thể hiện giá trị của các đồng xu

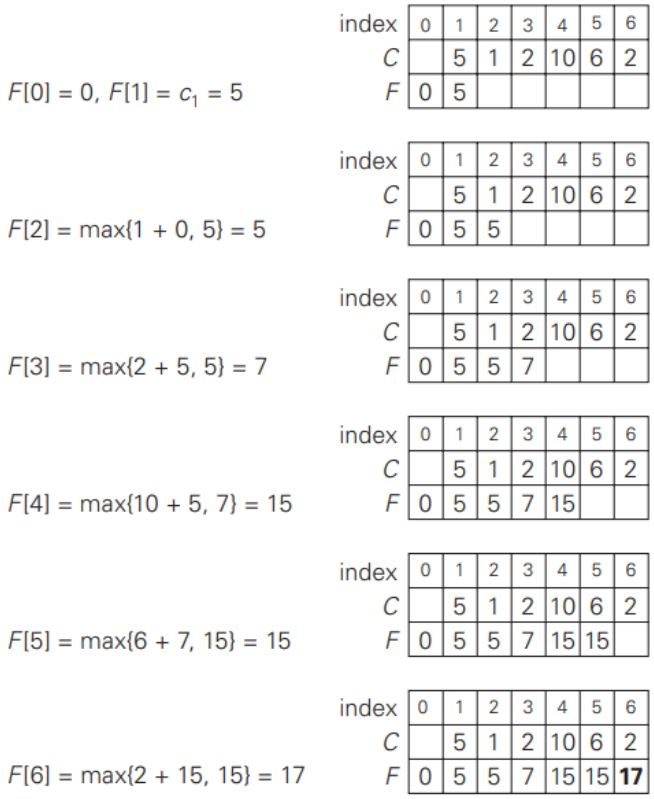
// Output: Số tiền lớn nhất có thể chọn được

F [0]← 0; F [1]← C [1]

for $i \leftarrow 2$ **to** n **do**

    $F[i] \leftarrow \max(C[i] + F[i - 2], F[i - 1])$

return $F[n]$



Hình 2. Áp dụng thuật toán *CoinRow* với dãy tiền xu cho trước 5, 1, 2, 10, 6, 2.

Giải thuật này được áp dụng cho dãy tiền xu với các mệnh giá lần lượt là 5, 1, 2, 10, 6, 2 như được minh họa trong Hình 2. Nó cho ra số tiền lớn nhất là 17. Nên nhớ rằng ở thuật toán trên, thực ra chúng ta cũng đã giải quyết bài toán cho *i* đồng xu đầu tiên trong dãy, với 1 ≤ i ≤ 6. Ví dụ, với *i* = 3, số tiền lớn nhất là *F* (3) = 7.

Để tìm tập các đồng xu cho ra lời giải tối ưu, ta cần truy ngược lại các lần tính toán để xem nhánh nào trong hai khả năng, c_n + *F* (*n* - 2), *F* (*n* - 1), đã cho ra giá trị tối đa trong công thức (**3**). Trong lần áp dụng công thức cuối cùng, tổng c_6 + *F* (4) cho ra giá trị lớn nhất, có nghĩa là đồng xu c_6 = 2 là một phần của lời giải tối ưu. Chuyển sang *F* (4), số tiền lớn nhất được tạo ra bởi tổng c_4 + *F* (2), có nghĩa là đồng xu c_4 = 10 cũng là một phần của lời giải tối ưu. Cuối cùng, số tiền lớn nhất trong việc tính *F* (2) được tạo ra bởi *F* (1), cho thấy đồng xu c_1 = 5 là một phần của lời giải tối ưu và đồng xu c_2 không phải. Do đó, lời giải tối ưu là {c_1, c_4, c_6}. Để tránh việc tính toán lại những giá trị giống nhau trong quá trình truy ngược, thông tin về khả năng nào trong hai nhánh của công thức (**3**) lớn hơn có thể được ghi vào trong một mảng phụ khi tính toán giá trị của *F*.

Sử dụng *CoinRow* để tìm *F* (*n*), số tiền lớn nhất có thể chọn, cũng như tập các đồng xu tạo thành một bộ lời giải tối ưu, rõ r sẽ tốn tànghời gian Θ(*n*) và không gian Θ(*n*). Ta thấy cách tiếp cận bottom-up này vượt trội hơn so với các phương pháp khác như áp dụng trực

tiếp phương pháp top-down của công thức (**3**) và giải quyết bài toán bằng exhaustive search.

VÍ DỤ 2: *Bài toán trả tiền thối*

Xét trường hợp chung của vấn đề sau: Tìm số tiền thối nhỏ nhất bằng cách sử dụng số lượng đồng xu ít nhất có thể với các mệnh giá $d_1 < d_2 < ... < d_m$. Giả sử có sẵn số lượng không giới hạn đồng xu cho mỗi trong m mệnh giá $d_1 < d_2 < ... < d_m$ trong đó $d_1 = 1$.

Gọi $F(n)$ là số đồng xu ít nhất có giá trị bằng n; để thuận tiện ta định nghĩa $F(0) = 0$. Số tiền n chỉ có thể được thu được bằng cách thêm một đồng xu với mệnh giá d_j vào số tiền $n - d_j$ với $j = 1, 2, ..., m$ sao cho $n \geq d_j$. Do đó, ta có thể xem xét tất cả các mệnh giá đó và chọn một mệnh giá để tối thiểu hóa F (n - dj) + 1. Vì 1 là một hằng số, ta có thể tìm $F(n - d_j)$ nhỏ nhất trước đó và sau đó thêm 1 vào nó. Do đó, chúng ta có quan hệ truy hồi sau đây cho $F(n)$:

$$F(n) = \min \{ F(n - d_j) \} + 1 \text{ với } n > 0 \text{ và j: } n \geq d_j$$

$$F(0) = 0. \qquad (4)$$

Ta có thể tính $F(n)$ bằng cách điền vào một bảng từ trái sang phải theo cách tương tự như đã làm ở trên cho bài toán *CoinRow*, nhưng với hàm min thay vì max.

THUẬT TOÁN *ChangeMaking* ($D[1..m]$, n)

// Áp dụng quy hoạch động để tìm số tiền xu ít nhất với các

// mệnh giá $d_1 < d_2 < ... < d_m$ trong đó $d_1 = 1$

// để thỏa mãn tổng số tiền được cho trước là n

// Input: số nguyên dương n và mảng $D[1..m]$ gồm các số nguyên dương tăng dần

// thể hiện các mệnh giá tiền xu, trong đó $D[1] = 1$

// Output: Số tiền xu ít nhất cần dùng để tổng số tiền thối lại là n

$F[0] \leftarrow 0$

for $i \leftarrow 1$ **to** n **do**

    $temp \leftarrow \infty$; $j \leftarrow 1$

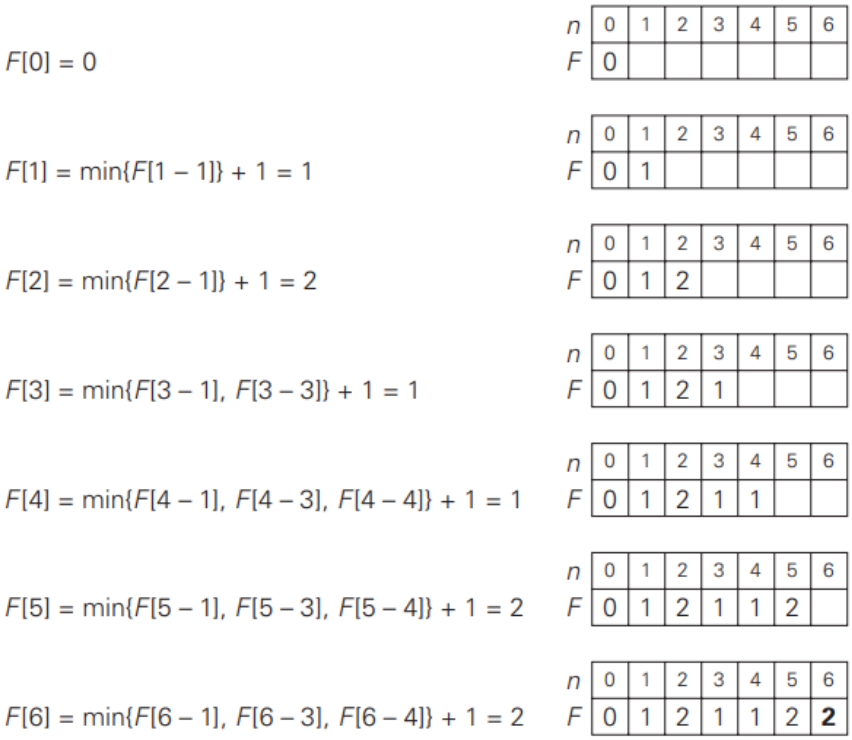
    while $j \leq m$ **and** $i \geq D[j]$ **do**

        $temp \leftarrow \min(F[i - D[j]], temp)$

        $j \leftarrow j + 1$

    $F[i] \leftarrow temp + 1$

return $F[n]$



Hình 3. Áp dụng thuật toán *ChangeMaking* với số lượng $n = 6$ và các mệnh giá xu 1, 3 và 4.

Việc áp dụng thuật toán với số tiền $n = 6$ và các mệnh giá là 1, 3, 4 được thể hiện trong Hình 3. Kết quả thu được là hai đồng xu. Độ phức tạp thời gian và không gian của thuật toán lần lượt là $O(nm)$ và $\Theta(n)$.

Để tìm tập các đồng xu của một lời giải tối ưu, ta cần truy ngược lại các lần tính toán để xem mệnh giá nào đã cho ra giá trị nhỏ nhất trong công thức (**4**). Lấy ví dụ với trường hợp ở trên, lần áp dụng cuối cùng của công thức (cho $n = 6$), giá trị nhỏ nhất được tạo ra bởi $d_2 = 3$. Giá trị nhỏ thứ hai (cho $n = 6 - 3$) cũng được tạo ra bởi đồng xu d_2. Do đó, tập đồng xu ít nhất cho $n = 6$ là hai đồng xu có mệnh giá là 3.

VÍ DỤ 3: *Bài toán Robot thu thập tiền xu*

Một số đồng xu được đặt trong các ô của bảng n x m, mỗi ô chỉ có tối đa một đồng xu. Một robot, được đặt tại ô trên cùng bên trái của bảng, cần thu thập nhiều đồng xu nhất có thể và mang chúng đến ô dưới cùng bên phải. Trong mỗi bước, robot có thể di chuyển một ô sang phải hoặc một ô xuống từ vị trí hiện tại. Khi robot đến một ô có đồng xu, nó luôn nhặt đồng xu đó. Hãy thiết kế một thuật toán để tìm số lượng đồng xu tối đa mà robot có thể thu thập và đường đi khi nó thực hiện.

Gọi $F(i, j)$ là số lượng đồng xu lớn nhất mà robot có thể thu thập và mang đến ô (i, j) ở hàng i và cột j của bảng. Nó có thể đến ô này từ ô kề cạnh $(i - 1, j)$ phía trên hoặc từ ô kề cạnh $(i, j - 1)$ bên trái của nó. Số lượng tiền xu lớn nhất mà có thể mang đến các ô này lần lượt là $F(i - 1, j)$ và $F(i, j - 1)$. Tất nhiên, không có ô kề cạnh phía trên cho các ô trong hàng

đầu tiên và không có ô kề cạnh bên trái cho các ô trong cột đầu tiên. Đối với các ô này, ta giả định rằng $F(i - 1, j)$ và $F(i, j - 1)$ bằng 0 cho các ô kề cạnh không tồn tại của chúng. Do đó, số lượng đồng xu lớn nhất mà robot có thể mang đến ô (i, j) là số lớn nhất của hai số này cộng thêm một đồng xu có thể có tại ô (i, j) hiện tại. Nói cách khác, chúng ta có công thức sau cho $F(i, j)$:

$$F(i, j) = \max\{F(i - 1, j), F(i, j - 1)\} + c_{ij} \text{ với } 1 \le i \le n, 1 \le j \le m$$

$$F(0, j) = 0 \text{ với } 1 \le j \le m \text{ và } F(i, 0) = 0 \text{ với } 1 \le i \le n, \quad (\mathbf{5})$$

ở đây, $c_{ij} = 1$ nếu có một đồng xu trong ô (i, j), và $c_{ij} = 0$ trong trường hợp ngược lại.

Sử dụng các công thức này, ta có thể điền vào bảng $F(i, j)$ kích thước $n \times m$ theo từng hàng hoặc từng cột, như thường thấy trong các thuật toán quy hoạch động liên quan đến bảng hai chiều.

THUẬT TOÁN *RobotCoinCollection* (C [1..n, *1*..m])

// Áp dụng quy hoạch động để tìm số lượng đồng xu lớn nhất mà robot có thể

// thu thập trên bảng n x m bằng cách bắt đầu từ ô (1, 1) di chuyển đến ô góc

// dưới bên phải, chỉ được đi sang phải hoặc xuống dưới.

// Input: Ma trận C [1..n, *1*..m] với các phần tử bằng 1 và 0 cho các ô có và không có

// đồng xu, tương ứng.

// Output: Số lượng đồng xu lớn nhất mà robot có thể mang đến ô (n, m).

$F[1, 1] \leftarrow C[1, 1]$;

for j ← 2 **to** m **do**

    $F[1, j] \leftarrow F[1, j - 1] + C[1, j]$

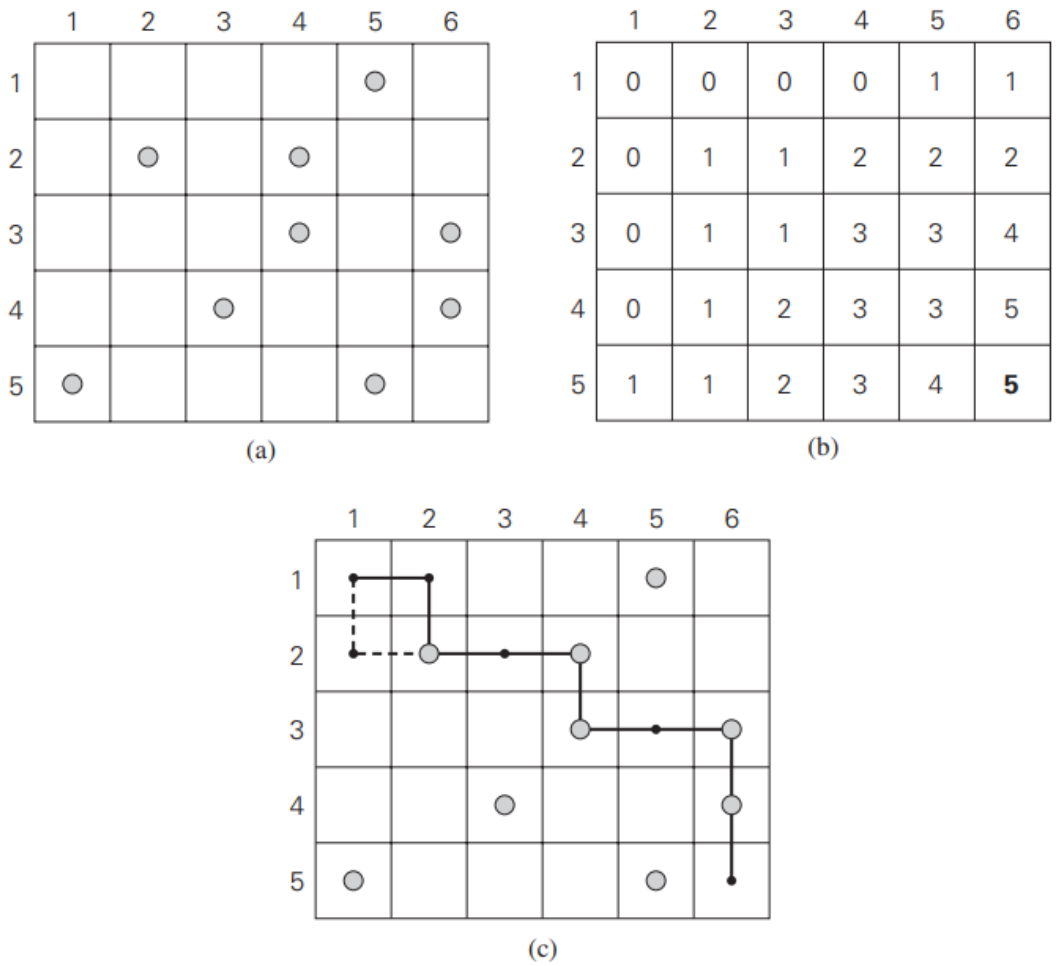
for $i \leftarrow 2$ **to** n **do**

    $F[i, 1] \leftarrow F[i - 1, 1] + C[i, 1]$

    for j ← 2 **to** m **do**

        $F[i, j] \leftarrow \max(F[i - 1, j], F[i, j - 1]) + C[i, j]$

return $F[n, m]$



Hình 4. (a) Vị trí các đồng xu. (b) Kết quả sau khi chạy thuật toán trên. (c) Hai đường đi qua 5 đồng xu, cũng là số đồng xu tối đa có thể thu thập được.

Thuật toán được minh họa trong Hình 4b cho vị trí các đồng xu trong Hình 4a. Bởi vì mỗi lần tính toán giá trị của $F(i, j)$ bằng công thức (**5**) cho mỗi ô trong bảng chỉ mất một khoảng thời gian hằng số, do đó độ phức tạp thời gian của thuật toán là $\Theta(nm)$. Độ phức tạp không gian của nó cũng là $\Theta(nm)$.

Để tìm ra đường đi tối ưu, ta cần truy ngược lại các lần tính toán: nếu $F(i - 1, j) > F(i, j - 1)$, một đường đi tối ưu đến ô (i, j) phải đến từ ô phía trên; nếu $F(i - 1, j) < F(i, j - 1)$, một đường đi tối ưu đến ô (i, j) phải đến từ ô bên trái; và nếu $F(i - 1, j) < F(i, j - 1)$, nó có thể đến được ô (i, j) từ cả hai hướng. Điều này cho phép có hai đường đi tối ưu cho trường hợp trong Hình 4a, được hiển thị trong Hình 4c. Một đường đi tối ưu có thể được tìm ra trong thời gian $\Theta(n + m)$.

Công thức chung dùng đệ quy để giải các bài toán quy hoạch động "đếm cách"

1. Tìm cách để chia bài đếm ra thành tổng của những bài toán nhỏ tương đương.

2. Tìm sopping conditon của bài đếm (trường hợp nào return 1? return 0?)
3. Nhớ sử dụng bảng nhớ phụ để giúp đệ quy tránh tính toán lại.

Công thức chung dùng đệ quy để giải các bài toán quy hoạch động "dùng ít nhất"

1. Quan hệ truy hồi: sử dụng hàm min để tính tối ưu của từng điểm nhằm xây dựng tối ưu của toàn bài. Cách làm: Giá trị hiện tại (chỉ có khi tìm chi phí, không có khi tìm cách) + min(nhánh 1, nhánh 2).
2. Stopping condition của bài toán tối ưu phải trả về 2 trường hợp: giá trị kết thúc có trên dữ liệu và giá trị kết thúc ở ngoài biên để khiến hàm min chọn / không chọn giá trị có trên dữ liệu (tương ứng inf hoặc 0):
    - Tối ưu toàn cục: lưu tổng các phương pháp tối ưu cục bộ => có thể làm hỏng các tính toán, hỏng tính kế thừa riêng biệt của từng đoạn, dẫn đến kết quả sau cùng của toàn bài sẽ nhỏ hơn kết quả tối ưu phải có.
    - Chỉ so sánh và trả về kết quả tối ưu ở bước cuối cùng nhằm tìm lời giải toàn cục.
3. Sử dụng bảng phụ để giúp đệ quy tránh phải tính toán:
    - Nên kiểm tra xem kết quả nhớ phụ có chung tính chất với kết quả tối ưu đoạn con hay không (bài toán thối tiền).
    - Nếu hai kết quả không chung tính chất, có thể lưu kết quả để xây dựng tối ưu đoạn con riêng (bằng bảng nhớ), trả về kết quả tối ưu của các nhánh con bằng đệ quy.

4. Bài toán Knapsack và hàm bộ nhớ:

Bài toán đặt ra: Cho n vật phẩm có trọng lượng đã biết $w_1, \dots, w_n$, các giá trị của vật phẩm $v_1, \dots, v_n$ và một chiếc túi đựng có sức chứa W, hãy tìm tập con các vật phẩm phù hợp với túi đựng sao cho tổng giá trị của các vật phẩm đã chọn đạt giá trị lớn nhất. Ở đây, ta giả sử rằng tất cả các trọng lượng và sức chứa của túi đựng là các số nguyên dương; các giá trị của vật phẩm không nhất thiết phải là số nguyên.

Để thiết kế một thuật toán quy hoạch động, chúng ta cần rút ra một quan hệ truy hồi thể hiện lời giải cho bài toán cái túi đựng dưới dạng lời giải cho các bài toán nhỏ hơn. Ta hãy xem xét một trường hợp được xác định bởi i vật phẩm đầu tiên, $1 \leq i \leq n$, với các trọng số $w_1, \dots, w_i$, giá trị của vật phẩm $v_1, \dots, v_i$ và sức chứa túi đựng j, $1 \leq j \leq W$. Giả sử $F(i, j)$ là giá trị của một giải pháp tối ưu cho trường hợp này, tức là tổng giá trị của i vật phẩm đầu tiên phù hợp với chiếc túi đựng có sức chứa j đạt giá trị lớn nhất. Ta có thể chia tất cả các tập hợp con của i vật phẩm đầu tiên phù hợp với chiếc túi đựng có sức chứa j thành hai loại: những loại không bao gồm vật phẩm thứ i và những loại có chứa vật phẩm thứ i.

Lưu ý :

1. Trong số các tập con không bao gồm vật phẩm thứ i, giá trị của một tập con tối ưu, theo định nghĩa, là $F(i - 1, j)$.

2. Trong số các tập hợp con bao gồm vật phẩm thứ i (tại i , $j - w_i \geq 0$), một tập hợp con tối ưu được tạo thành từ vật phẩm này và một tập hợp con tối ưu của $i - 1$ vật phẩm đầu tiên phù hợp với chiếc túi đựng có sức chứa $j - W_i$. Giá trị của một tập hợp con tối ưu như vậy là $v_i + F\,(i-1, j - w_i)$.

Do đó, giá trị của một lời giải tối ưu trong tất cả các tập con khả thi của i vật phẩm đầu tiên là giá trị lớn nhất trong hai giá trị đó. Tất nhiên, nếu vật phẩm thứ i không vừa với túi đựng, thì giá trị của tập hợp con tối ưu được chọn từ i vật phẩm đầu tiên sẽ giống như giá trị của tập hợp con tối ưu được chọn từ i - 1 vật phẩm đầu tiên. Từ quan sát này, ta có thể suy ra công thức sau:

$$F(i,\ j) = \begin{cases} \max\{F\ (i-1,\ j\ ), v_i\ +\ F(i\ -\ 1,\ j\ -\ w_i)\},\ j\ -\ w_i \geq 0 \\ F(i-1,\ j) \qquad\qquad\qquad\qquad\qquad\qquad ,\ j\ -\ w_i < 0 \end{cases} \tag{6}$$

Với điều kiện ban đầu

$$F(0, j) =\ 0 \text{ khi } j\ \geq 0 \text{ và } F(i, 0) = 0\ khi\ i \geq 0 \tag{7}$$

Mục tiêu của chúng ta là tìm $F\,(n, W)$, giá trị lớn nhất của một tập hợp các vật phẩm từ n vật phẩm được cung cấp và vừa vào túi đựng với sức chứa W, cùng với tập hợp vật phẩm tối ưu.

Hình 5 minh họa lời giải sử dụng công thức (**6**) và (**7**) thông qua dạng bảng. Với $i, j >$ 0, để tính giá trị trong hàng thứ i và cột thứ j, tức $F(i,\ j)$, ta tính giá trị lớn nhất của hàng trước và cùng cột, so với tổng của v_i và giá trị trong hàng trước và w_i cột bên trái. Ta có thể điền các giá trị vào bảng theo từng hàng hoặc theo từng cột.

		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
w_i, v_i	i	0		$F(i, j)$	
	n	0			goal

Hình 5. Bảng lời giải bài toán Knapsack sử dụng phương pháp quy hoạch động.

VÍ DỤ Xét một ví dụ với dữ kiện như sau:

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity $W = 5$.

i \ j	0	1	2	3	4	5
0	$F(0,j) = 0$	$F(0,j) = 0$	$F(0,j) = 0$	$F(0,j) = 0$	$F(0,j) = 0$	$F(0,j) = 0$
1	$F(i,0) = 0$	$F(0,1) = 0$ vì $(j - w_1 < 0)$	$\max\{F(0,2), 12 + F(0,0)\} = 12$	12	12	12
2	$F(i,0) = 0$	10	12	22	22	22
3	$F(i,0) = 0$	10	12	22	30	32
4	$F(i,0) = 0$	10	15	25	30	**37**

Bảng lời giải bằng phương pháp quy hoạch động, sử dụng các công thức (**6**) và (**7**) để tìm các mục trong bảng.

Do đó, giá trị lớn nhất là *F* (4, 5) = $37. Chúng ta có thể tìm ra cách bố trí của tập hợp vật phẩm tối ưu bằng cách truy ngược các tính toán của mục này trong bảng. Vì *F* (4, 5)> *F* (3, 5), vật phẩm 4 phải nằm trong một lời giải tối ưu cùng với một tập hợp tối ưu để điền vào 5 - 2 = 3 đơn vị còn lại của sức chứa của túi đựng. Vì *F* (3, 3) = *F* (2, 3), vật phẩm 3 không cần phải có trong một tập hợp tối ưu. Vì *F* (2, 3) > *F* (1, 3), vật phẩm 2 là một phần của lời giải tối ưu, và dựa vào phần tử *F* (1, 3 - 1) để chỉ ra sự bố trí những phần tử còn lại. Tương tự, vì *F* (1, 2) > *F* (0, 2), vật phẩm 1 là phần tử cuối cùng của lời giải tối ưu {vật phẩm 1, vật phẩm 2, vật phẩm 4}.

Hiệu quả thời gian và hiệu quả không gian của thuật toán này đều là trong $\Theta(nW)$. Thời gian cần thiết để tìm cách bố trí của một lời giải tối ưu là $O(n)$.

Memory Functions

Như chúng ta đã tìm hiểu ở trên, quy hoạch động xử lý các bài toán mà lời giải của chúng tuân theo một mối quan hệ truy hồi với các bài toán con chồng chéo nhau. Cách tiếp cận trực tiếp top-down để tìm kiếm giải pháp cho một mối quan hệ như vậy khiến thuật toán giải quyết các vấn đề con nhiều hơn một lần và do đó rất không hiệu quả (thường là hàm mũ hoặc tệ hơn). Trong khi đó, quy hoạch động bottom-up cổ điển tìm ra lời giải bằng cách điền vào bảng các lời giải cho tất cả các bài toán con nhỏ hơn, nhưng mỗi bài toán con chỉ được giải một lần. Tuy nhiên, vấn đề của cách tiếp cận này là các giải pháp cho những bài toán con nhỏ hơn thường không cần thiết để suy ra giải pháp cho bài toán chính. Và nhược điểm này không xuất hiện trong cách tiếp cận top-down, nên ta cần tìm cách kết hợp điểm mạnh từ cả hai cách tiếp cận. Mục đích là để có được một phương pháp chỉ giải các bài toán con cần thiết và chỉ giải một lần. Vậy nên có sự ra đời của ***memory functions***.

Phương pháp này giải quyết vấn đề theo cách top-down, nhưng thêm vào đó, duy trì một bảng nhớ như trong quy hoạch động bottom-up. Ban đầu, tất cả các mục của bảng được khởi tạo với một giá trị "null" để chỉ ra rằng chúng chưa được tính toán. Sau đó, khi một giá trị mới cần được tính toán, phương pháp kiểm tra mục tương ứng trong bảng trước: nếu mục này không phải là "null", nó đơn giản được truy xuất từ bảng; nếu không, nó sẽ được tính toán bằng lời gọi đệ quy và kết quả sau đó được ghi lại trong bảng.

Thuật toán sau đây thực hiện ý tưởng này cho bài toán cái túi. Sau khi khởi tạo bảng, hàm đệ quy cần được gọi với $i = n$ (số lượng vật phẩm) và $j = W$ (sức chứa cái túi).

THUẬT TOÁN *MFKnapsack* (*i, j*)

// Thực hiện phương pháp hàm bộ nhớ cho bài toán cái túi

// Input: Một số nguyên không âm *i* cho biết số lượng vật phẩm đang xét và một số nguyên không âm *j* cho biết sức chứa của cái túi

// Output: Giá trị của một tập con tối ưu có thể đựng được của *i* món hàng đầu tiên

// Chú ý: Sử dụng các biến toàn cục là các mảng đầu vào *Weights* [1..*n*], *Values* [1..*n*]
// và bảng *F* [0..*n*, 0..*W*] với các giá trị ban đầu là -1 ngoại trừ dòng 0 và cột 0 có giá
// trị ban đầu là 0

if *F* [*i*, *j*] < 0

if *j* < *Weights* [*i*]

value ← *MFKnapsack* (*i* - 1, *j*)

else

value ← max (*MFKnapsack* (*i* - 1, *j*)

Values [*i*] + *MFKnapsack* (*i* - 1, *j* - *Weights* [*i*]))

F [*i*, *j*] ← *value*

return $F[i, j]$

i \ j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	-	12	22	-	22
3	0	-	-	22	-	32
4	0	-	-	-	-	**37**

Bảng lời giải bằng phương pháp quy hoạch động sử dụng hàm bộ nhớ.

Ta thấy mục $V(1, 2)$ không quan trọng trong bảng chỉ cần truy xuất lại mà không cần tính toán lại. Đối với các bài toán lớn hơn, tỷ lệ các mục như vậy có thể đáng kể hơn.

Nhìn chung, hiệu quả thời gian của phương pháp hàm bộ nhớ cũng tương tự như phương pháp bottom-up, bởi vì chúng có độ phức tạp thời gian bằng nhau. Tuy nhiên, trên thực tế, hàm bộ nhớ sẽ hiệu quả hơn phương pháp bottom-up bởi nó loại bỏ những tính toán dư thừa, tránh giải một bài toán con nhiều lần. Ta cũng nên nhớ rằng một thuật toán hàm bộ nhớ có thể không hiệu quả về không gian hơn một phiên bản bottom-up tối ưu hiệu quả về không gian của cùng bài toán.