THUẬT TOÁN QUY HOẠCH ĐỘNG

1. Cấu trúc cơ bản của thuật toán Quy hoạch động

Thuật toán Quy hoạch động trong C++ chủ yếu sử dụng mảng hoặc bảng 2D để lưu trữ các kết quả của các bài toán con. Các kết quả này được sử dụng lại để xây dựng kết quả cho bài toán tổng thể, tránh việc tính toán lại các bài toán con lặp lại, giúp giảm độ phức tạp.

---

2. Ví dụ: Tính số Fibonacci thứ n

Đề bài: Tính số Fibonacci thứ nnn, với F(0)= 0, F(1)= 1, và công thức đệ quy là:
F(n)=F(n−1)+F(n−2)vớin≥2

Cách giải bài toán:

1. Giải quyết bài toán bằng phương pháp đệ quy thông thường (không tối ưu hóa):

```
#include <iostream>
using namespace std;
int fibonacci(int n) {
    if (n <= 1)
        return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
int main() {
    int n;
    cout << "Nhap n: ";
    cin >> n;
    cout << "Fibonacci(" << n << ") = " << fibonacci(n) << endl;
    return 0;
}
```

Vấn đề: Phương pháp này có độ phức tạp thời gian là $O(2^n)$ do tính toán lại các giá trị đã tính trước đó, dẫn đến hiệu suất kém.

---

2. Giải quyết bài toán bằng Memoization (Top-Down)

Memoization là một kỹ thuật giúp lưu trữ kết quả của các bài toán con trong một mảng hoặc bảng. Khi cần tính toán một giá trị mà đã được tính toán trước đó, ta sẽ lấy giá trị từ bảng thay vì tính lại.

```
#include <iostream>
#include <vector>
using namespace std;
int fibonacciMemo(int n, vector<int>& memo) {
    if (n <= 1)
        return n;
    if (memo[n] != -1) // Nếu giá trị đã được tính, trả về giá trị đó
        return memo[n];
    memo[n] = fibonacciMemo(n - 1, memo) + fibonacciMemo(n - 2, memo);
    return memo[n];
}
int main() {
    int n;
    cout << "Nhap n: ";
    cin >> n;

    // Khởi tạo mảng memo với giá trị -1 (chưa tính)
    vector<int> memo(n + 1, -1);

    cout << "Fibonacci(" << n << ") = " << fibonacciMemo(n, memo) << endl;
    return 0;
```

```
}
```
Giải thích:

- Mảng `memo` sẽ lưu trữ giá trị Fibonacci đã tính toán để tránh tính lại nhiều lần.
- Độ phức tạp thời gian của thuật toán này là O(n), vì mỗi giá trị Fibonacci chỉ được tính một lần.

---

3. Giải quyết bài toán bằng Tabulation (Bottom-Up)

Tabulation là một kỹ thuật giải quyết bài toán từ dưới lên, tính dần dần từ các bài toán con nhỏ nhất lên bài toán tổng thể. Chúng ta sẽ sử dụng một mảng để lưu trữ kết quả.

```
int fibonacciTab(int n) {
    if (n <= 1)
        return n;
    vector<int> dp(n + 1, 0);  // Mảng dp lưu kết quả
    dp[1] = 1; // Fibonacci(1) = 1

    for (int i = 2; i <= n; ++i) {
        dp[i] = dp[i - 1] + dp[i - 2];  // Công thức Fibonacci
    }

    return dp[n];  // Trả về kết quả cuối cùng
}
int main() {
    int n;
    cout << "Nhap n: ";
    cin >> n;

    cout << "Fibonacci(" << n << ") = " << fibonacciTab(n) << endl;
    return 0;
}
```

Giải thích:

- Mảng `dp` lưu trữ kết quả của các giá trị Fibonacci từ F(0)đến F(n)
- Thuật toán này có độ phức tạp thời gian O(n) và độ phức tạp không gian cũng là O(n)

---

3. Bài toán Knapsack (Ba lô)

Một bài toán rất phổ biến khác để minh họa Quy hoạch động là bài toán **Knapsack** (Ba lô). Đây là bài toán tối ưu hóa trong đó ta cần chọn một số vật phẩm sao cho tổng giá trị là lớn nhất, với điều kiện tổng trọng lượng không vượt quá dung tích của ba lô.

Đề bài: Cho một ba lô có trọng lượng tối đa là W và một danh sách các vật phẩm, mỗi vật phẩm có trọng lượng wi và giá trị vi . Làm sao để chọn các vật phẩm sao cho tổng giá trị là lớn nhất mà tổng trọng lượng không vượt quá W.

Cách giải bằng Quy hoạch động:

```
int knapsack(int W, const vector<int>& weights, const vector<int>& values, int n)
{
    vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));
        // dp[i][j] là giá trị tối ưu khi có i vật phẩm và dung lượng ba lô j

    for (int i = 1; i <= n; ++i) {
        for (int w = 1; w <= W; ++w) {
            if (weights[i - 1] <= w) {
               // Nếu vật phẩm thứ i có thể chứa trong ba lô
     dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1]);
            } else {
                dp[i][w] = dp[i - 1][w];  // Không lấy vật phẩm thứ i
```

```
            }
        }
    }
    return dp[n][W];  // Giá trị tối ưu cuối cùng
}
int main() {
    int n, W;
    cout << "Nhap so luong vat pham: ";
    cin >> n;
    cout << "Nhap dung luong ba lo: ";
    cin >> W;

    vector<int> weights(n), values(n);
    cout << "Nhap trong luong va gia tri cua tung vat pham: " << endl;
    for (int i = 0; i < n; ++i) {
        cin >> weights[i] >> values[i];
    }

    cout << "Gia tri toi uu: " << knapsack(W, weights, values, n) << endl;
    return 0;
}
```

Giải thích:

- Mảng `dp[i][w]` lưu trữ giá trị tối đa có thể đạt được với i vật phẩm và dung lượng ba lô w.
- Với mỗi vật phẩm, ta kiểm tra xem có thể đưa nó vào ba lô hay không (nếu trọng lượng vật phẩm nhỏ hơn hoặc bằng dung lượng còn lại của ba lô).
- Thuật toán có độ phức tạp thời gian là O(n×W), trong đó n là số vật phẩm và W là dung lượng ba lô.

Đề bài:

- Có một ba lô với dung lượng tối đa là W=7(dung lượng ba lô).
- Có 4 vật phẩm, với trọng lượng và giá trị như sau:

Vật phẩm	Trọng lượng (kg)	Giá trị (VND)
1	3	4
2	4	5
3	5	7
4	6	10

Mục tiêu: Tìm ra giá trị tối đa có thể đạt được mà không làm quá tải ba lô, tức là tổng trọng lượng không vượt quá dung lượng ba lô W=7.

Cách giải quyết bằng thuật toán Quy hoạch động:

Để giải bài toán này, chúng ta sẽ xây dựng một bảng DP, trong đó dp[i][w]dp[i][w]dp[i][w] sẽ lưu trữ giá trị tối ưu khi chúng ta xem xét các vật phẩm từ 1 đến iii và ba lô có dung lượng tối đa www.

Mảng DP:

- dp[i][w]: Giá trị tối đa có thể có với i vật phẩm và ba lô có dung lượng w.

Quy trình tính toán:

1. Nếu không chọn vật phẩm i, giá trị tối ưu là giá trị tối ưu của bài toán con dp[i−1][w].
2. Nếu chọn vật phẩm i, giá trị tối ưu là giá trị tối ưu của bài toán con dp[i−1][w−wi]+vi , với wi là trọng lượng của vật phẩm thứ i, và vi là giá trị của vật phẩm thứ i.
3. Ta chọn giá trị lớn nhất giữa hai trường hợp trên.

Xây dựng bảng DP:

Bây giờ, ta sẽ đi qua từng bước chi tiết để tính toán giá trị tối ưu cho bài toán này.

Khởi tạo bảng DP:

Khởi tạo một bảng DP với kích thước (n+1)×(W+1) trong đó nnn là số vật phẩm và W là dung lượng ba lô.

Bảng DP

Vật phẩm \ Dung lượng	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4
2	0	0	0	4	5	5	5	9
3	0	0	0	4	5	7	7	9
4	0	0	0	4	5	7	10	10

Giải thích:

- **Hàng đầu tiên (vật phẩm 0):** Nếu không có vật phẩm nào, giá trị tối ưu luôn bằng 0 cho mọi dung lượng ba lô.
- **Cột đầu tiên (dung lượng 0):** Nếu ba lô có dung lượng bằng 0, giá trị tối ưu cũng là 0 vì không thể chứa bất kỳ vật phẩm nào.

Bước tính toán:

Giờ ta sẽ tính toán từng giá trị trong bảng DP, từng bước một.

Vật phẩm 1 (Trọng lượng = 3, Giá trị = 4):

- Với dung lượng ba lô từ 0 đến 2, ta không thể chứa vật phẩm này, nên giá trị tối ưu là 0.
- Với dung lượng ba lô từ 3 trở đi, ta có thể chứa vật phẩm này, nên giá trị tối ưu là 4 (là giá trị của vật phẩm 1).

Cập nhật bảng DP:

Vật phẩm \ Dung lượng	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4

Vật phẩm 2 (Trọng lượng = 4, Giá trị = 5):

- Với dung lượng ba lô từ 0 đến 3, ta không thể chứa vật phẩm này, nên giá trị tối ưu là giá trị trước đó.

- Với dung lượng ba lô từ 4 trở đi, ta có thể chọn giữa việc không lấy vật phẩm này (giá trị tối ưu là giá trị trước đó), hoặc lấy vật phẩm này. Nếu chọn vật phẩm này, giá trị tối ưu là 555 (giá trị của vật phẩm 2).

Cập nhật bảng DP:

Vật phẩm \ Dung lượng	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4
2	0	0	0	4	5	5	5	9

Vật phẩm 3 (Trọng lượng = 5, Giá trị = 7):

- Với dung lượng ba lô từ 0 đến 4, ta không thể chứa vật phẩm này, nên giá trị tối ưu là giá trị trước đó.
- Với dung lượng ba lô từ 5 trở đi, ta có thể chọn giữa việc không lấy vật phẩm này (giá trị tối ưu là giá trị trước đó), hoặc lấy vật phẩm này. Nếu chọn vật phẩm này, giá trị tối ưu là 777.

Cập nhật bảng DP:

Vật phẩm \ Dung lượng	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4
2	0	0	0	4	5	5	5	9
3	0	0	0	4	5	7	7	9

Vật phẩm 4 (Trọng lượng = 6, Giá trị = 10):

- Với dung lượng ba lô từ 0 đến 5, ta không thể chứa vật phẩm này, nên giá trị tối ưu là giá trị trước đó.
- Với dung lượng ba lô từ 6 trở đi, ta có thể chọn giữa việc không lấy vật phẩm này (giá trị tối ưu là giá trị trước đó), hoặc lấy vật phẩm này. Nếu chọn vật phẩm này, giá trị tối ưu là 101010.

Cập nhật bảng DP:

Vật phẩm \ Dung lượng	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	0	0	0	0	0	0	0

Vật phẩm \ Dung lượng	0	1	2	3	4	5	6	7
1	0	0	0	4	4	4	4	4
2	0	0	0	4	5	5	5	9
3	0	0	0	4	5	7	7	9
4	0	0	0	4	5	7	10	10

Kết quả:

- Giá trị tối đa có thể đạt được là **10**, và được lưu trữ ở ô dp[4][7]dp[4][7]dp[4][7], tức là khi xem xét tất cả 4 vật phẩm và ba lô có dung lượng 777.
- Các vật phẩm được chọn là vật phẩm thứ 3 (trọng lượng 5, giá trị 7) và vật phẩm thứ 4 (trọng lượng 6, giá trị 10).

Tóm tắt:

- Bảng DP được xây dựng và cập nhật từ dưới lên.
- Mỗi ô trong bảng DP lưu trữ giá trị tối ưu tại một bước tính toán.
- Giá trị tối đa cuối cùng có thể đạt được là 10, với dung lượng ba lô 7, và ta chọn vật phẩm 3 và vật phẩm 4.

DÃY CON TĂNG DÀI NHẤT

Giải thích chi tiết về thuật toán Quy hoạch động:

1. **Khởi tạo:** Mỗi phần tử trong dãy đều có thể là một dãy con tăng dài nhất, ít nhất là chính nó, do đó khởi tạo `dp[i] = 1` cho tất cả các phần tử.
2. **Duyệt qua các phần tử:** Để tìm dãy con tăng dài nhất kết thúc tại phần tử thứ `i`, ta sẽ duyệt qua các phần tử trước `i` (tức là từ `0` đến `i-1`):
   - Nếu `arr[j] < arr[i]` (nghĩa là phần tử tại vị trí `j` nhỏ hơn phần tử tại vị trí `i`), thì ta có thể mở rộng dãy con tăng kết thúc tại `arr[j]` bằng cách thêm `arr[i]`.
   - Cập nhật `dp[i] = max(dp[i], dp[j] + 1)`.
3. **Kết quả:** Dãy con tăng dài nhất sẽ có độ dài là giá trị lớn nhất trong mảng `dp[]`.

```
#include <iostream>
#include <vector>
```

```
#include <algorithm>
using namespace std;
// Hàm tìm dãy con tăng dài nhất (LIS)
int longestIncreasingSubsequence(const vector<int>& arr) {
    int n = arr.size();
    if (n == 0) return 0;  // Nếu mảng rỗng, độ dài LIS là 0.
    // Mảng dp, dp[i] lưu trữ độ dài LIS kết thúc tại arr[i]
    vector<int> dp(n, 1);  // Khởi tạo dp[i] = 1 vì mỗi phần tử ban đầu là một LIS dài 1.
    // Duyệt qua từng phần tử trong mảng để cập nhật giá trị LIS cho từng dp[i]
    for (int i = 1; i < n; ++i) {
        for (int j = 0; j < i; ++j) {
            // Nếu arr[i] > arr[j], ta có thể nối arr[i] vào dãy con LIS kết thúc tại arr[j]
            if (arr[i] > arr[j]) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
    }
    // Trả về độ dài LIS là giá trị lớn nhất trong dp[]
    return *max_element(dp.begin(), dp.end());
}
int main() {
    // Ví dụ đầu vào
    vector<int> arr = {10, 22, 9, 33, 21, 50, 41, 60, 80};
    cout << "Dai LIS: " << longestIncreasingSubsequence(arr) << endl;
    return 0;
}
```

1. **Khởi tạo mảng `dp[]`:**
   - Mảng `dp[]` có kích thước bằng độ dài của mảng đầu vào `arr[]`.
   - `dp[i]` lưu trữ độ dài của dãy con tăng dài nhất kết thúc tại phần tử `arr[i]`.
   - Khởi tạo tất cả các phần tử trong `dp[]` bằng 1 vì mỗi phần tử đều có thể là một dãy con tăng dài nhất chỉ có nó, nghĩa là độ dài LIS ban đầu là 1.
2. **Duyệt qua từng phần tử trong mảng `arr[]`:**
   - Duyệt qua từng phần tử `arr[i]` từ vị trí `1` đến `n-1`.
   - Với mỗi `i`, duyệt qua tất cả các phần tử trước đó từ `0` đến `i-1` (vị trí `j`).
   - Nếu `arr[i] > arr[j]`, có nghĩa là ta có thể nối phần tử `arr[i]` vào dãy con kết thúc tại `arr[j]`, do đó cập nhật giá trị của `dp[i]` là `max(dp[i], dp[j] + 1)`.
3. **Tìm độ dài dãy con tăng dài nhất:**
   - Cuối cùng, giá trị lớn nhất trong mảng `dp[]` sẽ là độ dài của dãy con tăng dài nhất (LIS).

Ví dụ:

Dãy đầu vào:

arr = {10, 22, 9, 33, 21, 50, 41, 60, 80}

1. **Khởi tạo mảng dp[]:**

dp = {1, 1, 1, 1, 1, 1, 1, 1, 1}

2. **Duyệt qua các phần tử:**
   - Với i=1(arr[1] = 22):
     - Duyệt qua các phần tử trước đó:
       - j=0, arr[1] > arr[0] (\Rightarrow dp[1] = max(dp[1], dp[0] + 1) = 2`
   - Mảng dp sau cập nhật:

dp = {1, 2, 1, 1, 1, 1, 1, 1, 1}

   - Với i=2 (arr[2] = 9):
     - Duyệt qua các phần tử trước đó:
       - j=0, arr[2] > arr[0] (\Rightarrow dp[2] = 2`
       - j=1, arr[2] < arr[1] (không thay đổi)
   - Mảng dp sau cập nhật:

dp = {1, 2, 1, 1, 1, 1, 1, 1, 1}

- Với i=3 (arr[3] = 33):
  - Duyệt qua các phần tử trước đó:
    - j=0j = 0j=0, arr[3] > arr[0] (\Rightarrow dp[3] = 2`
    - j=1j = 1j=1, arr[3] > arr[1] (\Rightarrow dp[3] = max(dp[3], dp[1] + 1) = 3`
    - j=2j = 2j=2, arr[3] > arr[2] (không thay đổi)
- Mảng dp sau cập nhật:

dp = {1, 2, 1, 3, 1, 1, 1, 1, 1}

- Tiếp tục cho các phần tử còn lại, mảng dp cuối cùng sẽ trở thành:

dp = {1, 2, 1, 3, 3, 4, 4, 5, 6}

3. **Kết quả cuối cùng:**

   Độ dài của dãy con tăng dài nhất là giá trị lớn nhất trong mảng dp[], tức là **6**.

Tóm tắt:

- **Dãy con tăng dài nhất (LIS)** của dãy arr = {10, 22, 9, 33, 21, 50, 41, 60, 80} có độ dài **6**.
- **Dãy con tăng dài nhất (LIS)** có thể là {10, 22, 33, 50, 60, 80}.

Độ phức tạp:

- **Thời gian:** $O(n^2)$, vì phải duyệt qua tất cả các cặp phần tử trong mảng.
- **Không gian:** O(n), mảng dp[] có kích thước bằng số phần tử trong mảng arr[].

Tối ưu hóa (O(n log n)):

Có một cách tối ưu hóa thuật toán này xuống O(nlogn) bằng cách sử dụng mảng phụ và thực hiện **binary search** để tìm vị trí của phần tử mới trong mảng phụ. Tuy nhiên, phương pháp này phức tạp hơn và bạn có thể yêu cầu thêm nếu muốn tìm hiểu về cách tối ưu hóa này.

Phương pháp:

Ý tưởng của phương pháp này là sử dụng một mảng phụ để duy trì các phần tử của dãy con tăng dài nhất mà chúng ta tìm được trong quá trình duyệt qua mảng. Mảng phụ này sẽ không chứa dãy con tăng dài nhất hoàn chỉnh, nhưng nó sẽ chứa các phần tử sao cho:

- Mảng này luôn có các phần tử theo thứ tự tăng dần.
- Mảng này có độ dài là độ dài của LIS.

Chúng ta sử dụng **binary search** để tìm vị trí thích hợp để chèn mỗi phần tử vào mảng phụ sao cho mảng phụ luôn giữ được tính chất là dãy con tăng dần.

Cách hoạt động:

1. Duyệt qua từng phần tử trong mảng arr[].
2. Dùng **binary search** để tìm vị trí mà phần tử hiện tại có thể thay thế trong mảng phụ.
3. Nếu phần tử có thể thay thế một phần tử trong mảng phụ (vị trí của nó tìm được qua binary search), ta thay thế.

4. Nếu không thể thay thế (tức là phần tử lớn hơn tất cả phần tử trong mảng phụ), ta thêm phần tử vào cuối mảng phụ.

Mảng phụ:

- Mảng phụ sẽ lưu trữ các giá trị của dãy con tăng dài nhất cho đến thời điểm hiện tại, nhưng không nhất thiết phải là dãy con liên tiếp.
- Cuối cùng, kích thước của mảng phụ chính là độ dài của dãy con tăng dài nhất.

Thuật toán:

1. **Khởi tạo:** Một mảng phụ rỗng sub[].
2. **Duyệt qua các phần tử:** Duyệt qua các phần tử trong mảng arr[]. Dùng binary search để tìm vị trí thích hợp trong mảng phụ sub[].
3. **Kết quả:** Kích thước của mảng sub[] chính là độ dài của LIS.

```
#include <iostream>
#include <vector>
#include <algorithm>  // For binary_search, lower_bound
using namespace std;

// Hàm tìm dãy con tăng dài nhất (LIS) tối ưu với O(n log n)
int daycon(const vector<int>& arr) {
    vector<int> sub;  // Mảng phụ để lưu các phần tử của LIS

    for (int num : arr) {
        // Tìm vị trí mà num có thể chèn vào mảng phụ sao cho mảng vẫn tăng dần
        auto it = lower_bound(sub.begin(), sub.end(), num);

        // Nếu không tìm thấy vị trí, thêm num vào cuối mảng phụ
        if (it == sub.end()) {
            sub.push_back(num);
        }
        // Nếu tìm thấy vị trí, thay thế giá trị tại vị trí đó
        else {
            *it = num;
        }
    }
for(auto x:sub)cout<<x<<" ";
    // Kích thước của mảng phụ là độ dài LIS
```

```
    return sub.size();
}

int main() {
    int n;
    // Ví dụ đầu vào
    cin>>n;
    vector<int> a(n);
    for(int i=0;i<n;i++)cin>>a[i];

    cout << daycon(a) << endl;

    return 0;
}
```

// a= {10, 22, 9, 33, 21, 50, 41, 60, 80};

Kq: 6

9 21 33 41 60 80