Chuyên đề này trình bày các chiến lược thiết kế thuật giải như: Đệ quy, Quay lui (Backtracking), Nhánh và cận (Branch and Bound), Tham ăn (Greedy Method), Chia để trị (Divide and Conquer) vả Quy hoạch động (Dynamic Programming). Đây là các chiến lược tổng quát, nhưng mỗi phương pháp chỉ áp dụng được cho một số lớp bài toán nhất định, chứ không tồn tại một phương pháp vạn năng để thiết kế thuật toán giải quyết mọi bài toán. Các phương pháp thiết kế thuật toán trên chỉ là chiến lược, có tính định hướng tìm thuật toán. Việc áp dụng chiến lược để tìm ra thuật toán cho một bài toán cụ thể còn đòi hỏi nhiều sáng tạo. Trong chuyên đề này, ngoài phần trình bày về các phương pháp, chuyên đề còn có những ví dụ cụ thể, cùng với thuật giải và cài đặt, để có cái nhìn chi tiết từ việc thiết kế giải thuật đến xây dựng chương trình.

**Chương 1: ĐỆ QUY**

----------------------

**1. Khái niệm về đệ quy**

**1.1. Khái niệm về hình thức đệ quy**

Trong toán học và khoa học máy tính, các tính chất (hoặc cấu trúc) được gọi là đệ quy nếu trong đó một lớp các đối tượng hoặc phương pháp được xác định bằng việc xác định một số rất ít các trường hợp hoặc phương pháp đơn giản (thông thường chỉ một) và sau đó xác định quy tắc đưa các trường hợp phức tạp về các trường hợp đơn giản.

* Chẳng hạn, định nghĩa sau là định nghĩa đệ quy của tổ tiên:

Bố mẹ của một người là tổ tiên của người ấy (trường hợp cơ bản).

Bố mẹ của tổ tiên một người bất kỳ là tổ tiên của người ấy (bước đệ quy).

Các định nghĩa kiểu như vậy cũng thường thấy trong toán học (chính là quy nạp toán học)

**1.2. Khái niệm về đệ quy**

Từ những định nghĩa trên, ta có thể rút ra khái niệm cơ bản theo 2 cách:

1. Đệ quy (trong tiếng Anh là *recursion*) là phương pháp dùng trong các chương trình máy tính trong đó có một hàm tự gọi chính nó.
2. Một khái niệm X được định nghĩa theo đệ quy nếu trong định nghĩa X có sử dụng chính khái niệm X.

 ***Ví dụ:*** Sau đây là một số ví dụ điển hình về đệ quy:

1. Định nghĩa về số tự nhiên:

0 là một số tự nhiên

n là một số tự mhiên khi n-1 là 1 số tự nhiên

2. Định nghĩa về giai thừa n!:

0! = 1

Nếu n>0 thì n!=n\*(n-1)!

**1.3. Các bước để giải bài toán đệ quy**

* Thông số hóa bài toán (hiểu bài toán)
* Tìm các điều kiện biên (chặn), tìm giải thuật cho các tình huống này
* Tìm giải thuật tổng quát theo hướng đệ quy lui dần về tình huống bị chặn.

 ***Ví dụ:*** Bài toán tính giai thừa của một số tự nhiên n (tính *n*!):

* Thông số hóa bài toán: Cách tính giai thừa n!:

0! = 1

Nếu n>0 thì n!=n\*(n-1)!

* Tìm các điều kiện biên (chặn), tìm giải thuật cho các tình huống này:

if (n == 0) gt =1

* Tìm giải thuật tổng quát theo hướng đệ quy lui dần về tình huống bị chặn.

if (n > 0) gt = n\*gt(n - 1);

**2. Giải thuật đệ quy**

**2.1. Khái niệm giải thuật đệ quy**

Nếu lời giải của một bài toán T’ được thực hiện bằng lời giải của một bài toán T có dạng giống như T’, thì đó là một lời giải đệ quy Giải thuật tương ứng với lời giải như vậy gọi là giải thuật đệ quy.

Thoạt nghe thì các bạn thấy có vẻ hơi lạ nhưng điểm mấu chốt cần lưu ý là: T’ tuy có dạng giống như T, nhưng theo một nghĩa nào đó, nó phải "nhỏ" hơn T.

Trong khoa học máy tính có một phương pháp chung để giải các bài toán trong Tin học là chia bài toán thành các bài toán con đơn giản hơn cùng loại. Phương pháp này được gọi là kỹ thuật lập trình chia để trị. Chính nó là chìa khóa để thiết kế nhiều giải thuật quan trọng, là cơ sở của quy hoạch động.

**2.2. Ví dụ**

Tính giai thừa của một số tự nhiên n (tính *n*!):

if (n == 0) gt=1;

else gt = n\*gt(n - 1);

Bài toán tính n! được chia nhỏ như sau:

Ta nhận thấy rằng, để tính n! ta cần phải tính được (n-1)!, để tính được (n-1)! Thì ta phải tính được (n-2)!, … cho đến khi chúng ta sẽ gặp côngviệc tính 1!, mà 1!=1, vì thế cứ thực hiện tính ngược trở lên chúng ta sẽ tính được n!.

**3. Chương trình con đệ quy**

**3.1. Khái niệm chương trình con đệ quy**

Là một chương trình con (hàm, thủ tục) mà trong thân của nó có lời gọi đến chính nó (hay còn gọi là lời gọi đệ quy) với kích thước nhỏ hơn của tham số.

 ***Ví dụ****:* Hàm tính giai thừa của một số tự nhiên n (tính *n*!):

long gt(int n);

{

if (n == 0) gt=1;

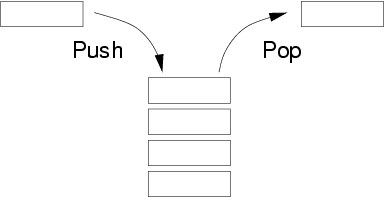
else gt = n\*gt(n - 1);

}

**3.2. Cấu trúc chính của chương trình con đệ quy**

Một chương trình con đệ qui căn bản gồm hai phần.

* *Phần neo (phần suy biến, cơ sở):* chứa các tác động của hàm hoặc thủ tục với một số giá trị cụ thể ban đầu của tham số.
  + *Ví dụ:* if (n==0) gt = 1;
* Phần đệ quy (phần tổng quát, hạ bậc) : định nghĩa tác động cần được thực hiện cho giá trị hiện thời của các tham số bằng các tác động đã được định nghĩa trước đây với kích thước tham số nhỏ hơn.
  + *Ví dụ:* gt=n\*gt(n-1);

**4. Nguyên tắc hoạt động của giải thuật đệ quy**

**4.1. Khái niệm stack:**

Stack là một cấu trúc lưu trữ, hoạt động theo nguyên tắc sau:

* Mỗi lần nộp vào hoặc lấy ra chỉ thực hiện với một phần tử.
* Phần tử nộp vào sau sẽ được lấy ra trước.

**Stack**

**4.2. Nguyên tắc hoạt động:**

* Khi thực hiện một giải thuật đệ quy thì các bước của giải thuật đệ quy sẽ lần lượt được thực hiện tuần tự.
* Khi gặp lời gọi đệ quy thì trước khi thực hiện lời gọi đệ quy, đoạn mã lệnh chưa được thực hiện xong cùng với các đối tượng dữ liệu liên quan tại thời điểm này sẽ được lưu vào stack.
* Đến lúc nào đó không thể thực hiện lời gọi đệ quy nữa thì các đối tượng được lưu trong stack sẽ lần lượt được lấy ra để xử lý.

 ***Ví dụ:*** Trong ví dụ trên, qui trình thực hiện như sau:

* Khi có lệnh gọi hàm, chẳng hạn: x = gt(3); thì máy sẽ ghi nhớ là:

gt(3) = 3 \* gt(2); và đi tính gt(2)

* Kế tiếp máy lại ghi nhớ: gt(2)= 2\*gt(1); và đi tính gt(1)
* Theo định nghĩa của hàm thì khi gt(1)= 1; máy sẽ quay ngược lại:

gt(2)= 2 \* 1; và cho kết quả là 2

* Tiếp tục: gt(3) = 3 \* 2; cho kết quả là 6
* Như vậy kết quả cuối cùng trả về là 6. Ta có: 3! = 6.

**5. Ưu điểm và hạn chế của đệ quy**

**5.1. Ưu điểm:**

* Mô tả được một số thao tác tính toán thông qua một đoạn lệnh ngắn, làm cho chương trình ngắn gọn, dễ hiểu, lộ rõ bản chất đệ quy.
* Rất thuận tiện để giải quyết các bài toán có bản chất đệ quy.
* Hiện nay vẫn có nhiều thuật toán chưa có lời giải đệ quy.
* Nhiều giải thuật rất dễ mô tả dạng đệ quy nhưng lại rất khó mô tả với giải thuật không-đệ-quy.
* Một chương trình viết theo giải thuật có tính đệ qui sẽ mang tính "người" hơn, do đó sẽ sáng sủa, dễ hiểu, nêu bật được bản chất của vấn đề.

**5.2. Hạn chế:**

* Vừa tốn bộ nhớ, chương trình chạy chậm
* Do đệ quy lưu trữ các dữ liệu trung gian vào Stack nên nếu Lưu nhiều bộ dữ liệu lớn trên stack nên có thể gây ra hiện tượng tràn Stack

**6. Một số bài toán về đệ quy**

**Bài 1: Tính lũy thừa: an** (với n là số nguyên dương)

#include <bits/stdc++.h>

using namespace std;

long long Power(int a, int b){

if (b == 0) return 1;

return a \* Power(a, b-1);

}

int main(){

int a, b;

cin >> a >> b;

cout << Power(a, b);

return 0;

}

**Bài 2**: **Tìm số fibonaxi thứ n**

Dãy số fibonacci là dãy số được tạo bằng cách như sau:

* Hai số đầu tiên là số 1.
* Các số tiếp theo lần lượt được tạo thành từ tổng của 2 số trước nó.
* Dãy fibonacci: 1, 1, 2, 3, 5, 8, 13, 21, ...

Nhập vào một số nguyên dương n, hãy đưa ra số fibonacci thứ n. Hãy thực hiện điều đó bằng giải thuật đệ quy.

**Ví dụ:**

* Test mẫu 1:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4 | 3 |

#include <bits/stdc++.h>

using namespace std;

long long fibonacci(int n)

{

if (n == 1 || n == 2) return 1;

return fibonacci(n-1) + fibonacci(n-2);

}

int main(){

int n;

cin >> n;

cout << fibonacci(n);

return 0;

}

**Bài 3**: **Tìm số đảo ngược**

Tìm số đảo ngược trong C là bài toán nhập vào một số nguyên dương n từ bàn phím. In ra số đảo ngược của số n vừa nhập. Ví dụ chúng ta nhập số 1234 thì sẽ thu về số 4321 chẳng hạn.

#include <bits/stdc++.h>

using namespace std;

/\*Hàm tìm số đảo ngược trong C\*/

void InDaoNguoc(int n)

{

if(n!=0)

{

cout<<n%10;

InDaoNguoc(n/10);

}

}

int main(){

int n;

cout << "Nhap mot so nguyen duong: ");

cin >> n;

if (n==0) {cout << "So dao nguoc: " << 0; return 0;}

else {cout << "So dao nguoc: " << InDaoNguoc(n); return 0;}

}

**Bài 4:** **Tìm ước chung lớn nhất của 2 số nguyên dương.**

Nhập vào hai số nguyên a và b. Hãy tìm ước chung lớn nhất của chúng.

(Ước chung lớn nhất của hai số nguyên là một số lớn nhất mà cả hai số đó đều chia hết).

**Ví dụ:**

* Test mẫu 1:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 10 15 | 5 |

* Với a = 10, b = 15 thì kết quả mong muốn là 5.
* Test mẫu 2:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3 7 | 1 |

* Với a = 3, b = 7 thì kết quả mong muốn là 1.

#include <bits/stdc++.h>

using namespace std;

int ucln(int a, int b)

{

if (b == 0) return a;

if (a % b == 0) return b;

return ucln(b, a%b);

}

int main()

{

int a, b;

cin >> a >> b;

cout << ucln(a, b);

return 0;

}

**Bài 5: Bài toán Tháp Hà Nội:**

* ***Đề bài:*** Có 3 cái cọc, đánh dấu A, B, C, và N cái đĩa. Mỗi đĩa đều có một lỗ chính giữa để đặt xuyên qua cọc, các đĩa đều có kích thước khác nhau. Ban đầu tất cả đĩa đều được đặt ở cọc thứ nhất theo thứ tự đĩa nhỏ hơn ở trên.

Yêu cầu của bài là chuyển tất cả các đĩa từ cọc A qua cọc C với ba ràng buộc như sau:

* + 1. Mỗi lần chỉ chuyển được một đĩa.
    2. Trong quá trình chuyển đĩa có thể dùng cọc còn lại (B) để làm cọc trung gian.
    3. Chỉ cho phép đặt đĩa có bán kính nhỏ hơn lên đĩa có bán kính lớn hơn.
* ***Phân tích bài toán:***

Trong bài toán trên hình dung một lời giải tổng quát cho trường hợp tổng quát N đĩa là không dễ dàng.

Hãy bắt đầu với các trường hợp đơn giản:

* + Với N = 1: Chỉ cần chuyển đĩa này từ cọc A qua cọc C là xong.
  + Với N = 2: Để đảm bảo ràng buộc thứ hai ta bắt buộc chuyển đĩa trên cùng từ cọc A qua cọc B. Chuyển tiếp đĩa còn lại từ cọc A qua cọc C. Chuyển tiếp đĩa đang ở cọc B sang cọc C.
  + Với N = 3: Ta phải thực hiện 7 bước như sau:

|  |  |  |
| --- | --- | --- |
|  |  | **A B C** |
| **Trạng thái ban đầu**    **Bước 1: Chuyển một đĩa từ A qua C.**  **Bước 2: Chuyển một đĩa từ A qua B.**  **Bước 3: Chuyển một đĩa từ C qua B.**  **Bước 4: Chuyển một đĩa từ A qua C.**  **Bước 5: Chuyển một đĩa từ B qua A.**  **Bước 6: Chuyển một đĩa từ B qua C.**  **Bước 7: Chuyển một đĩa từ A qua C.** |  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

*\* Nhận xét:*

Ở kết quả của bước thứ ba. Đây là một kết quả quan trọng vì nó cho ta thấy từ trường hợp N=3 bài toán đã được phân chia thành hai bài toán với kích thước nhỏ hơn: đó là bài toán chuyển 1 đĩa từ cọc A qua cọc C lấy cọc B làm trung gian và bài toán chuyển 2 đĩa (dời) từ cọc B sang cọc C lấy cọc A làm trung gian. Hai bài toán con này đã biết cách giải (trường hợp N=1 và trường hợp N=2).

Nhận xét đó cho ta gợi ý trong trường hợp tổng quát:

* Bước 1: Dời (N-1) đĩa trên cùng từ cọc A sang cọc B lấy cọc C làm trung gian.
* Bước 2: Chuyển 1 đĩa dưới cùng từ cọc A sang cọc C.
* Bước 3: Dời (N-1) đĩa đang ở cọc B sang cọc C lấy cọc A làm trung gian.

Như vây, bài toán đối với N đĩa ở trên được “đệ qui” về hai bài toán (N-1) đĩa và bài toán 1 đĩa. Quá trình đệ qui sẽ dừng lại khi N=0 (không còn đĩa để dời hoặc chuyển).

 ***Cài đặt chương trình:***

#include <bits/stdc++.h>

using namespace std;

void Tower(int n , char a, char b, char c )

{

if(n==1)

{

cout<<"\t"<<a<<"------->"<<c<<endl;

return;

}

Tower(n-1,a,c,b);

Tower(1,a,b,c);

Tower(n-1,b,a,c);

}

int main()

{

char a='A', b='B', c='C';

int n;

cout<<"Nhap n: ";

cin>>n;

Tower(n,a,b,c);

return 0;

}

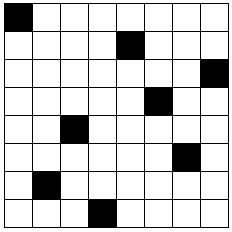
**Chương 2. Quay lui (Backtracking)**

Quay lui, vét cạn, thử sai, duyệt … là một số tên gọi tuy không đồng nghĩa nhưng cùng chỉ một phương pháp trong tin học: *tìm nghiệm của một bài toán bằng cách xem xét tất cả các phương án có thể*. Đối với con người phương pháp này thường là không khả thi vì số phương án cần kiểm tra lớn. Tuy nhiên đối với máy tính, nhờ tốc độ xử lí nhanh, máy tính có thể giải rất nhiều bài toán bằng phương pháp quay, lui vét cạn.

Ưu điểm của phương pháp quay lui, vét cạn là *luôn đảm bảo tìm ra nghiệm đúng, chính xác*. Tuy nhiên, hạn chế của phương pháp này là *thời gian thực thi lâu, độ phức tạp lớn*. Do đó vét cạn thường chỉ phù hợp với các bài toán có kích thước nhỏ.

**1. Phương pháp**

Trong nhiều bài toán, việc tìm nghiệm có thể quy về việc tìm vector hữu hạn (x1, x2, …, xn, … ), độ dài vector có thể xác định trước hoặc không. Vector này cần phải thoả mãn một số điều kiện tùy thuộc vào yêu cầu của bài toán. Các thành phần xi được chọn ra từ tập hữu hạn Ai.

Tuỳ từng trường hợp mà bài toán có thể yêu cầu: tìm một nghiệm, tìm tất cả nghiệm hoặc đếm số nghiệm.

**Ví dụ:** *Bài toán 8 quân hậu.*

Cần đặt 8 quân hậu vào bàn cờ vua 8 x 8, sao cho chúng không tấn công nhau, tức là không có hai quân hậu nào cùng hàng, cùng cột hoặc cùng đường chéo.

Ví dụ: hình bên là một cách đặt hậu thoả mãn yêu cầu bài toán, các ô được tô màu là vị trí đặt hậu.

Do các quân hậu phải nằm trên các hàng khác nhau, ta đánh số các quân hậu từ 1 đến 8, quân hậu i là quân hậu nằm trên hàng thứ i (i=1,2,…,8). Gọi xi là cột mà quân hậu i đứng. Như vậy nghiệm của bài toán là vector (x1,x2,…,x8) trong đó 1 ≤ xi ≤ 8, tức là xi được chọn từ tập Ai = {1,2,…,8}. Vector (x1,x2,…,x8) là nghiệm nếu xi ≠ xj và hai ô (i, xi), (j, xj) không nằm trên cùng một đường chéo.

Ví dụ: (1,5,8,6,3,7,2,4) là một nghiệm.

Tư tưởng của phương pháp quay lui vét cạn như sau: Ta xây dựng vector nghiệm dần từng bước, bắt đầu từ vector không ( ). Thành phần đầu tiên xi được chọn ra từ tập S1 = A1. Giả sử đã chọn được các thành phần x1,x2,…,xi-1 thì từ các điều kiện của bài toán ta xác định được tập Si (các ứng cử viên có thể chọn làm thành phần xi, Si là tập con của Ai). Chọn một phần tử xi từ Si ta mở rộng nghiệm được x1,x2,…,xi. Lặp lại quá trình trên để tiếp tục mở rộng nghiệm. Nếu không thể chọn được thành phần xi+1 (Si+1 rỗng) thì ta quay lại chọn một phần tử khác của Si cho xi. Nếu không còn một phần tử nào khác của Si ta quay lại chọn một phần tử khác của Si-1 làm xi-1 và cứ thế tiếp tục. Trong quá trình mở rộng nghiệm, ta phải kiểm tra nghiệm đang xây dựng đã là nghiệm của bài toán chưa. Nếu chỉ cần tìm một nghiệm thì khi gặp nghiệm ta dừng lại. Còn nếu cần tìm tất cả các nghiệm thì quá trình chỉ dừng lại khi tất cả các khả năng lựa chọn của các thành phần của vector nghiệm đã bị vét cạn.

***Lược đồ tổng quát của thuật toán quay lui vét cạn có thể biểu diễn bởi thủ tục Backtrack sau:***

void Backtrack()

{

S1=A1; k=1;

while (k>0)

{

while (Sk <> ∅) { *Sk khác rỗng* }

{

<chọn xk thuộc Sk>;

Sk=Sk – {xk};

if ((x1,x2,…,xk)là nghiệm ) <Đưa ra nghiệm>;

k=k+1;

<Xác định Sk>;

}

k=k-1; {*quay lui* }

};

};

Trên thực tế, thuật toán quay lui vét cạn thường được dùng bằng mô hình đệ quy như sau:

void Backtrack(i); {*xây dựng thành phần thứ i* }

{

<Xác định Si>;

for (xi thuộc Si )

{

<ghi nhận thành phần thứ i>;

if (tìm thấy nghiệm) <Đưa ra nghiệm>;

else Backtrack(i+1);

<loại thành phần i>;

};

};

Khi áp dụng lược đồ tổng quát của thuật toán quay lui cho các bài toán cụ thể, có ba vấn đề quan trọng cần làm:

- Tìm cách biểu diễn nghiệm của bài toán dưới dạng một dãy các đối tượng được chọn dần từng bước (x1,x2,…,xi).

- Xác định tập Si các ứng cử viên được chọn làm thành phần thứ i của nghiệm. Chọn cách thích hợp để biểu diễn Si.

- Tìm các điều kiện để một vector đã chọn là nghiệm của bài toán.

**2. Một số ví dụ áp dụng**

**2.1. Tổ hợp**

Một tổ hợp chập k của n là một tập con k phần tử của tập n phần tử. Chẳng hạn tập {1,2,3,4} có các tổ hợp chập 2 là:

{1,2}, {1,3, {1,4, {2,3}, {2,4}, {3,4}

Vì trong tập hợp các phần tử không phân biệt thứ tự nên tập {1,2} cũng là tập {2,1}, do đó, ta coi chúng chỉ là một tổ hợp.

***Bài toán:*** Hãy xác định tất cả các tổ hợp chập k của tập n phần tử. Để đơn giản ta chỉ xét bài toán tìm các tổ hợp của tập các số nguyên từ 1 đến n. Đối với một tập hữu hạn bất kì, bằng cách đánh số thứ tự của các phần tử, ta cũng đưa được về bài toán đối với tập các số nguyên từ 1 đến n.

Nghiệm của bài toán tìm các tổ hợp chập k của n phần tử phải thoả mãn các điều kiện sau:

- Là một vector X = (x1, x2, …, xk);

- xi lấy giá trị trong tập {1,2,…,n};

- Ràng buộc: xi < xi+1 với mọi giá trị i từ 1 đến k-1(vì tập hợp không phân biệt thứ tự phần tử nên ta sắp xếp các phần tử theo thứ tự tăng dần).

Ta có: 1 ≤ x1 <x2 < …< xk ≤ n, do đó tập Si (tập các ứng cử viên được chọn làm thành phần thứ i) là từ xi-1 + 1 đến (n-k+i). Để điều này đúng cho cả trường hợp i=1, ta thêm vào x0= 0.

Sau đây là chương trình hoàn chỉnh, chương trình sử dụng mô hình đệ quy để sinh tất cả các tổ hợp k chập n.

#include <bits/stdc++.h>

#define maxN 21

using namespace std;

long x[maxN];

long n, k;

void GhiNghiem(long x[])

{

for (long i=1; i<=k; i++) {cout << x[i] << ' ';}

cout << endl;

}

void ToHop(long i)

{

for (long j=x[i-1]+1; j<=n-k+i; j++)

{

x[i] = j;

if (i==k) GhiNghiem(x);

else ToHop(i+1);

}

}

int main()

{

cout << "Nhap n, k:";

cin >> n >> k;

x[0]=0;

ToHop(1);

return 0;

}

Ví dụ về Input / Output của chương trình:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4 2 | 1. 1 2 2. 1 3 3. 1 4 4. 2 3 5. 2 4 6. 3 4 |

Theo công thức, số lượng tổ hợp chập k=2 của n=4 là:



**2.2. Chỉnh hợp lặp**

Chỉnh hợp lặp chập k của n là một dãy k thành phần, mỗi thành phần là một phần tử của tập n phần tử, có xét đến thứ tự và không yêu cầu các thành phần khác nhau.

Một ví dụ dễ thấy nhất của chỉnh hợp lặp là các dãy nhị phân. Một dãy nhị phân độ dài m là một chỉnh hợp lặp chập m của tập 2 phần tử {0,1}. Các dãy nhị phân độ dài 3:

000, 001, 010, 011, 100, 101, 110, 111.

Vì có xét thứ tự nên dãy 101 và dãy 011 là 2 dãy khác nhau.

Như vậy, bài toán xác định tất cả các chỉnh hợp lặp chập k của tập n phần tử yêu cầu tìm các nghiệm như sau:

- Là một vector X=(x1,x2,…,xk)

- xi lấy giá trị trong tập {1,2,…,n}

- Không có ràng buộc nào giữa các thành phần.

Chú ý là cũng như bài toán tìm tổ hợp, ta chỉ xét đối với tập n số nguyên từ 1 đến n. Nếu phải tìm chỉnh hợp không phải là tập các số nguyên từ 1 đến n thì ta có thể đánh số các phần tử của tập đó để đưa về tập các số nguyên từ 1 đến n.

*{sử dụng một mảng x[1..n] để biểu diễn chỉnh hợp lặp.*

#include <bits/stdc++.h>

#define maxN 21

using namespace std;

long long x[maxN];

long n, k;

void GhiNghiem(long long x[])

{

for (long i=1; i<=k; i++) {cout << x[i] << ' ';}

cout << endl;

}

void ChinhHopLap(long i)

{

for (long j=1; j<=n; j++)

{

x[i] = j;

if (i==k) GhiNghiem(x);

else ChinhHopLap(i+1);

}

}

int main()

{

cout << "Nhap n, k:";

cin >> n >> k;

ChinhHopLap(1);

}

Ví dụ về Input/Output của chương trình:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 2 3 | 1. 1 1 1 2. 1 1 2 3. 1 2 1 4. 1 2 2 5. 2 1 1 6. 2 1 2 7. 2 2 1 8. 2 2 2 |

Theo công thức, số lượng chỉnh hợp lặp chập k=3 của n=2 là:



**2.3. Chỉnh hợp không lặp**

Khác với chỉnh hợp lặp là các thành phần được phép lặp lại (tức là có thể giống nhau), chỉnh hợp không lặp chập k của tập n (k≤n) phần tử cũng là một dãy k thành phần lấy từ tập n phần tử có xét thứ tự nhưng các thành phần không được phép giống nhau.

*Ví dụ:* Có n người, một cách chọn ra k người để xếp thành một hàng là một chỉnh hợp không lặp chập k của n.

Một trường hợp đặc biệt của chỉnh hợp không lặp là hoán vị. Hoán vị của một tập n phần tử là một chỉnh hợp không lặp chập n của n. Nói một cách trực quan thì hoán vị của tập n phần tử là phép thay đổi vị trí của các phần tử (do đó mới gọi là hoán vị).

\_

Nghiệm của bài toán tìm các chỉnh hợp không lặp chập k của tập n số nguyên từ 1 đến n là các vector X thoả mãn các điều kiện:

- X có k thành phần: X=(x1,x2,…,xk)

- xi lấy giá trị trong tập {1,2,…,n}

- Ràng buộc: các giá trị xi đôi một khác nhau, tức là xi ≠ xj với mọi i≠j

Sau đây là chương trình hoàn chỉnh, chương trình sử dụng mô hình đệ quy để sinh tất cả các chỉnh hợp không lặp chập k của n phần tử.

#include <bits/stdc++.h>

#define maxN 21

using namespace std;

long x[maxN], d[maxN]={};//Khởi tạo mảng d gồm toàn số 0

long n, k;

void GhiNghiem(long x[])

{

for (long i=1; i<=k; i++) {cout << x[i] << ' ';}

cout << endl;

}

void ChinhHopKoLap(long i)

{

for (long j=1; j<=n; j++)

if (d[j] == 0)

{

x[i] = j;

d[j] = 1;

if (i==k) GhiNghiem(x);

else ChinhHopKoLap(i+1);

d[j] = 0; //quay lui

}

}

int main()

{

cout << "Nhap n, k:";

cin >> n >> k;

ChinhHopKoLap(1);

}

Ví dụ về Input / Output của chương trình:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3 3 | 1. 1 2 3 2. 1 3 2 3. 2 1 3 4. 2 3 1 5. 3 1 2 6. 3 2 1 |

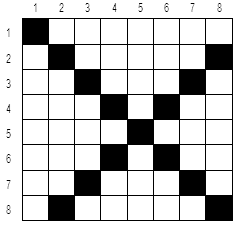
Theo công thức, số lượng chỉnh hợp không lặp chập k=3 của n=3 là:

\_



**2.4. Bài toán xếp 8 quân hậu**

Trong bài toán 8 quân hậu, nghiệm của bài toán có thể biểu diễn dưới dạng vector (x1,x2,…,x8) thoả mãn:

1. xi là tọa độ cột của quân hậu đang đứng ở dòng thứ i, 
2. Các quân hậu không đứng cùng cột tức là xi ≠ xj với i ≠ j.
3. Có thể dễ dàng nhận ra rằng hai ô (x1,y1) và (x2,y2) nằm trên cùng đường chéo chính (trên xuống dưới) nếu: x1−y1 =x2−y2 , hai ô (x1,y1) và (x2,y2) nằm trên cùng đường chéo phụ (từ dưới lên trên) nếu: x1y1 =x2y2 , nên điều kiện để hai quân hậu xếp ở hai ô (i, xi), (j, xj) không nằm trên cùng một đường chéo là:

\_

\_



Do đó, khi đã chọn được (x1,x2,…,xk-1) thì xk được chọn phải thoả mãn các điều kiện: xk ≠ xi

k – xk ≠ i – xi với mọi 1 ≤ i ≤ k

k + xk ≠ i + xi

Việc xác định tập Sk có thể thực hiện đơn giản và hiệu quả hơn bằng cách sử dụng các mảng đánh dấu. Cụ thể, khi ta đặt hậu i ở ô (i,x[i]), ta sẽ đánh dấu cột x[i] (dùng một mảng đánh dấu như ở bài toán chỉnh hợp không lặp), đánh dấu đường chéo chính (i-x[i]) và đánh dấu đường chéo phụ (i+x[i]).

Sau đây là chương trình đầy đủ, để liệt kê tất cả các cách xếp 8 quân hậu lên bàn cờ vua 8x8.

#include <bits/stdc++.h>

#define n 8

using namespace std;

int dem=0;

int x[n+1];

int cot[n+1]={}; // chỉ số cột từ 1 đến n

int cheochinh[2\*n]={}; // chỉ số của chéo chính được cộng thêm n: 1 đến 2\*n-1

int cheophu[2\*n+1]={}; // chỉ số của chéo phụ từ 2 đến 2\*n

void write\_out()

{

for (int i=1; i<=n; i++) cout << x[i] << ' ';

cout << endl;

dem++;

}

void xephau(int k)

{ // chọn hàng i cho hậu ở cột k

for (int i=1; i<=n; i++)

if ((!cot[i]) && (!cheochinh[k-i+n]) && (!cheophu[k+i]) )

{

x[k]=i;

cot[i]=1;

cheochinh[k-i+n] = 1;

cheophu[k+i] = 1;

if (k==n) write\_out();

else xephau(k+1);

cot[i]=0;

cheochinh[k-i+n]=0;

cheophu[k+i]=0;

}

}

int main()

{

xephau(1);

cout << "So cach xep: " << dem;

return 0;

}

Bài toán xếp hậu có tất cả 92 nghiệm, mười nghiệm đầu tiên mà chương trình tìm được là:

|  |
| --- |
| 1. 1 5 8 6 3 7 2 4  2. 1 6 8 3 7 4 2 5  3. 1 7 4 6 8 2 5 3  4. 1 7 5 8 2 4 6 3  5. 2 4 6 8 3 1 7 5  6. 2 5 7 1 3 8 6 4  7. 2 5 7 4 1 8 6 3  8. 2 6 1 7 4 8 3 5  9. 2 6 8 3 1 4 7 5  10.2 7 3 6 8 5 1 4 |
|
|
|
|
|
|
|
|
|

**2.5. Bài toán máy rút tiền tự động ATM**

Một máy ATM hiện có n (n≤20) tờ tiền có giá t1,t2, … ,tn. Hãy đưa ra một cách trả với số tiền đúng bằng S.

**Input:** Tệp “ATM.INP” có dạng:

- Dòng đầu là 2 số n và S

- Dòng thứ 2 gồm n số t1,t2, … ,tn

**Output :** Tệp “ATM.OUT” có dạng: Nếu có thể trả đúng S thì đưa ra cách trả lời, nếu không ghi -1.

|  |  |
| --- | --- |
| ATM.INP | ATM.OUT |
| 10 390  200 10 20 20 50 50 50 50 100 100 | 20 20 50 50 50 100 100 |

Nghiệm của bài toán là một dãy nhị phân độ dài n, trong đó thành phần thứ i bằng 1 nếu tờ tiền thứ i được sử dụng để trả, bằng 0 trong trường hợp ngược lại.

X=(x1,x2, … ,xn) là nghiệm nếu: x1.t1 + x2.t2 + … + xn.tn =S

Trong chương trình dưới đây có sử dụng một biến ok để kiểm soát việc tìm nghiệm. Ban đầu chưa có nghiệm, do đó khởi trị ok=FALSE. Khi tìm được nghiệm, ok sẽ được nhận giá trị bằng TRUE. Nếu ok=TRUE (đã tìm thấy nghiệm) ta sẽ không cần tìm kiếm nữa.

#include <bits/stdc++.h>

#define maxN 21

using namespace std;

int N;

bool ok;

long t[maxN], x[maxN], lx[maxN];

long S, sum;

void read\_input() {

freopen("ATM.INP","r", stdin);

cin >> N >> S;

for (int i=1; i<=N; i++) cin >> t[i];

}

void write\_output() {

for (int i=1; i<=N; i++)

if (x[i]==1) cout << t[i] << ' ';

}

void find(int i) {

if (ok) return;

for (int j=0; j<=1; j++) {

x[i] = j;

sum += j\*t[i];

if (i==N) {

if (sum==S) {

ok=true;

write\_output();

}

}

else find(i+1);

sum = sum - j\*t[i];

}

}

int main() {

read\_input();

freopen("ATM1.OUT","w", stdout);

ok=false;

find(1);

if (!ok) cout << -1;

return 0;

}

**Chương 3. Nhánh và cận**

**1. Phương pháp**

Trong thực tế, có nhiều bài toán yêu cầu tìm ra một phương án thoả mãn một số điều kiện nào đó, và phương án đó là tốt nhất theo một tiêu chí cụ thể. Các bài toán như vậy được gọi là bài toán tối ưu. Có nhiều bài toán tối ưu không có thuật toán nào thực sự hữu hiệu để giải quyết, mà cho đến nay vẫn phải dựa trên mô hình xem xét toàn bộ các phương án, rồi đánh giá để chọn ra phương án tốt nhất.

Phương pháp nhánh và cận là một dạng cải tiến của phương pháp quay lui, được áp dụng để tìm nghiệm của bài toán tối ưu.

Giả sử nghiệm của bài toán có thể biểu diễn dưới dạng một vector (x1,x2,…, xn), mỗi thành phần xi (i = 1,2,…,n) được chọn ra từ tập Si. Mỗi nghiệm của bài toán X = (x1,x2,…, xn), được xác định “độ tốt” bằng một hàm *f(X)* và mục tiêu cần tìm nghiệm có giá trị *f(X)* đạt giá trị nhỏ nhất (hoặc đạt giá trị lớn nhất).

Tư tưởng của phương pháp nhánh và cận như sau: Giả sử, đã xây dựng được k thành phần (x1,x2,…, xk) của nghiệm và khi mở rộng nghiệm (x1,x2,…, xk-1), nếu biết rằng tất cả các nghiệm mở rộng của nó (x1,x2,…,xk+1,…) đều không tốt bằng nghiệm tốt nhất đã biết ở thời điểm đó, thì ta không cần mở rộng từ (x1,x2,…,xk) nữa. Như vậy, với phương pháp nhánh và cận, ta không phải duyệt toàn bộ các phương án để tìm ra nghiệm tốt nhất mà bằng cách đánh giá các nghiệm mở rộng, ta có thể cắt bỏ đi những phương án (nhánh) không cần thiết, do đó việc tìm nghiệm tối ưu sẽ nhanh hơn. Cái khó nhất trong việc áp dụng phương pháp nhánh và cận là đánh giá được các nghiệm mở rộng, nếu đánh giá được tốt sẽ giúp bỏ qua được nhiều phương án không cần thiết, khi đó thuật toán nhánh cận sẽ chạy nhanh hơn nhiều so với thuật toán vét cạn.

Thuật toán nhánh cận có thể mô tả bằng mô hình đệ quy sau:

void branch\_and\_bound(i)

{

// Đánh giá các nghiệm mở rộng

if ({Các\_nghiệm\_mở\_rộng\_đều\_không\_tốt\_hơn\_best\_solution})

return;

for (value in S[i])

{

x[i] = value; // Ghi nhận thành phần thứ i.

if ({Tìm\_thấy\_nghiệm})

<Cập nhật best\_solution>; //Cập nhật lại best\_solution bằng nghiệm vừa tìm được nếu no tốt hơn nghiệm đang có.

else

branch\_and\_bound(i + 1); // Chưa tìm thấy nghiệm thì xây dựng tiếp.

{Loại\_bỏ\_thành\_phần\_thứ\_i}

}

}

Trong thủ tục trên, best\_solution là nghiệm tốt nhất đã biết ở thời điểm đó. Thủ tục <Cập nhật best\_solution> sẽ xác định “độ tốt” của nghiệm mới tìm thấy, nếu nghiệm mới tìm thấy tốt hơn best\_solution nthì best\_solution sẽ được cập nhật lại là nghiệm mới tìm được.

**2. Giải bài toán người du lịch bằng phương pháp nhánh cận.**

***Bài toán.*** Cho n thành phố đánh số từ 1 đến n và các tuyến đường giao thông hai chiều giữa chúng, mạng lưới giao thông này được cho bởi mảng C[1..n,1..n], ở đây Cij = Cji là chi phí đi đoạn đường trực tiếp từ thành phố i đến thành phố j.

Một người du lịch xuất phát từ thành phố 1, muốn đi thăm tất cả các thành phố còn lại mỗi thành phố đúng 1 lần và cuối cùng quay lại thành phố 1. Hãy chỉ ra cho người đó hành trình với chi phí ít nhất. Bài toán được gọi là bài toán người du lịch hay bài toán người chào hàng (Travelling Salesman Problem - TSP).

**Input:** Tệp “TSP.INP” có dạng:

- Dòng đầu chứa số n(1<n≤20), là số thành phố.

- n dòng tiếp theo, mỗi dòng n số mô tả mảng C

**Output:** Tệp “TSP.OUT” có dạng: - Dòng đầu là chi phí ít nhất

- Dòng thứ hai mô tả hành trình

**Ví dụ 1:**

|  |  |  |
| --- | --- | --- |
| TSP.INP | TSP.OUT | Hình minh họa |
| 4  0 20 35 42  20 0 34 30  35 34 0 12  42 30 12 0 | 97  1->2->4->3->1 |  |

**Ví dụ 2:**

|  |  |  |  |
| --- | --- | --- | --- |
| TSP.INP | TSP.OUT | Hình minh họa | |
| 4  0 20 35 10  20 0 90 50  35 90 0 12  10 50 12 0 | 117  1->2->4->3->1 |  |  |
|  |  |  | |

***Giải***

1) Hành trình cần tìm có dạng (x1 = 1, x2, ..., xn, xn+1 = 1), ở đây giữa xi và xi+1: hai thành phố liên tiếp trong hành trình phải có đường đi trực tiếp; trừ thành phố 1, không thành phố nào được lặp lại hai lần, có nghĩa là dãy (x1, x2, ..., xn) lập thành một hoán vị của (1, 2, ..., n).

2) **Duyệt quay lui:** x2 có thể chọn một trong các thành phố mà x1 có đường đi trực tiếp tới, với mỗi cách thử chọn x2 như vậy thì x3 có thể chọn một trong các thành phố mà x2 có đường đi tới (ngoài x1). Tổng quát: xi có thể chọn 1 trong các thành phố chưa đi qua mà từ xi-1 có đường đi trực tiếp tới (2in).

3) **Nhánh cận:** Khởi tạo cấu hình BestSolution có chi phí bằng +. Với mỗi bước thử chọn xi xem chi phí đường đi cho tới lúc đó có nhỏ hơn chi phí của cấu hình BestSolution không? nếu không nhỏ hơn thì thử giá trị khác ngay bởi có đi tiếp cũng chỉ tốn thêm. Khi thử được một giá trị xn ta kiểm tra xem xn có đường đi trực tiếp về 1 không ? Nếu có đánh giá chi phí đi từ thành phố 1 đến thành phố xn cộng với chi phí từ xn đi trực tiếp về 1, nếu nhỏ hơn chi phí của đường đi BestSolution thì cập nhật lại BestSolution bằng cách đi mới.

Trong cài đặt dưới đây, giả thiết rằng giữa mọi cặp thành phố đều tồn tại đường đi, và chi phí *cu*,*v*​ luôn bằng 0 nếu như *u*=*v*.

Mảng ***visited***dùng để đánh dấu một thành phố đã được thăm hay chưa trong một cấu hình *X*. Mảng ***x*** sử dụng để lưu cấu hình hiện tại, còn mảng ***x\_best*** sử dụng để lưu cấu hình tốt nhất với ***best\_cost*** là chi phí tốt nhất tìm được.

#include <bits/stdc++.h>

#define inf 1e9 + 7

using namespace std;

const int maxn = 21;

int n, current\_cost, best\_cost;

int visited[maxn], x\_best[maxn], x[maxn], c[maxn][maxn];

void enter()

{

cin >> n;

for (int i = 1; i <= n; ++i)

for (int j = 1; j <= n; ++j)

cin >> c[i][j];

// Khởi tạo trước thành phố đầu tiên là 1, đồng thời đánh dấu nó đã thăm.

x[1] = 1;

visited[1] = 1;

// Khởi tạo chi phí tối ưu bằng +oo, giả sử phương án hiện tại đang rất tệ.

best\_cost = inf;

}

// Cập nhật kết quả tốt nhất.

void update\_best\_solution(int current\_cost)

{

if (current\_cost + c[x[n]][1] < best\_cost)

{

best\_cost = current\_cost + c[x[n]][1];

for (int i = 1; i <= n; ++i)

x\_best[i] = x[i];

}

}

// In ra phương án tốt nhất tìm được.

void print\_best\_solution()

{

cout << best\_cost << endl;

for (int i = 1; i <= n; ++i)

cout << x\_best[i] << "->";

cout << 1;

}

// Giải thuật nhánh và cận.

void branch\_and\_bound(int i)

{

if (current\_cost >= best\_cost)

return;

for (int j = 2; j <= n; ++j)

if (!visited[j])

{

visited[j] = 1;

x[i] = j;

current\_cost += c[x[i - 1]][j];

// Đã sinh xong một cấu hình, cập nhật chi phí tốt nhất.

if (i == n)

update\_best\_solution(current\_cost);

// Chưa sinh xong, tiếp tục sinh thành phần tiếp theo với chi phí tăng thêm.

else

branch\_and\_bound(i + 1);

visited[j] = 0;

current\_cost -= c[x[i - 1]][j];

}

}

int main()

{

ios\_base::sync\_with\_stdio(false);

cin.tie(nullptr);

enter();

branch\_and\_bound(2);

print\_best\_solution();

return 0;

}

Chương trình trên là một giải pháp nhánh cận rất thô sơ giải bài toán TSP, có thể có nhiều cách đánh giá nhánh cận chặt hơn nữa làm tăng hiệu quả của chương trình.

**3. Bài toán máy rút tiền tự động ATM Bài toán**

**Bài toán:** Một máy ATM hiện có n (n≤20) tờ tiền có giá t1,t2,…,tn. Hãy tìm cách *trả ít tờ nhất* với số tiền đúng bằng S.

**Input:** Tệp “ATM.INP” có dạng:

- Dòng đầu là 2 số n và S

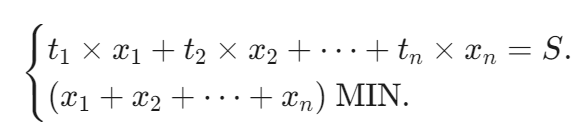
- Dòng thứ 2 gồm n số t1,t2,…,tn

**Output :** Tệp “ATM.OUT” có dạng: Nếu có thể trả tiền đúng bằng S thì đưa ra số tờ ít nhất cần trả và đưa ra cách trả, nếu không ghi -1.

|  |  |
| --- | --- |
| ATM.INP | ATM.OUT |
| 10 390  200 10 20 20 50 50 50 50 100 100 | 5  20 20 50 100 200 |

***Giải***

Nghiệm của bài toán có thể biểu diễn dưới dạng một vector gồm toàn các bit nhị phân 0-1 là *x*1, *x*2​,…, *xn*​ với ý nghĩa: *xi*​=0 là tờ tiền thứ *i* không được chọn, *xi*​=1 là tờ tiền thứ *i* được chọn.

Mục tiêu chúng ta đang cần tìm một bộ nghiệm sao cho:

Giả sử các bạn đã xây dựng được *i* thành phần của nghiệm là (*x*1​,*x*2​,…,*xi*​), tổng số tờ tiền đã sử dụng là *cnt* và số tiền đã trả được là *sum*, thì ta nhận xét thấy:

* Số tiền còn lại cần trả là *S*−*sum*.
* Nếu gọi *tmax*​[*i*+1] là giá trị của tờ tiền lớn nhất trong các tờ tiền còn lại (*tmax*​[*i*+1]=max(*ti*+1​,*ti*+2​,…,*tn*​)), thì ít nhất ta cần sử dụng thêm  ​ tờ tiền nữa, tức là tổng số tờ tiền tối thiểu cần dùng của nhánh phương án này là *cnt*+

Gọi số tờ tiền của cách trả tốt nhất hiện tại là ***c\_best***, thì nếu như ***c +***  >= ***c\_best*** ta sẽ không cần phải mở rộng các nghiệm từ (*x*1​,*x*2​,…,*xi*​) nữa.

Để kiểm soát các tờ tiền được chọn, ta sử dụng thêm hai mảng là ***x***để đánh dấu các tờ tiền được chọn trong một phương án, và ***x\_best*** để đánh dấu các tờ tiền được chọn trong phương án tốt nhất.

#include <bits/stdc++.h>

using namespace std;

const int maxn = 22;

int n, s, c, c\_best, sum,t[maxn], t\_max[maxn], x[maxn], x\_best[maxn];

void enter()

{

freopen("ATM.INP","r", stdin);

cin >> n >> s;

for (int i = 1; i <= n; ++i)

cin >> t[i];

}

void create\_data()

{

// c\_best là số tờ tiền sử dụng trong phương án tốt nhất.

// Ban đầu chưa có phương án nào, gán c\_best = n + 1.

c\_best = n + 1;

sum=0; c=0;

// t\_max[i] lưu giá trị của tờ tiền lớn nhất từ i tới n.

t\_max[n] = t[n];

for (int i = n - 1; i >= 0; --i)

t\_max[i] = max(t\_max[i + 1], t[i]);

}

// Nếu tìm được một phương án tốt hơn thì cập nhật lại kết quả.

void update\_best\_solution()

{

if ((sum == s) && (c < c\_best))

{

c\_best = c;

for (int i = 1; i <= n; i++)

x\_best[i] = x[i];

}

}

// In kết quả.

void printf\_result()

{

freopen("ATM.OUT","w",stdout);

// Không tìm được cách trả tiền, in ra -1.

if (c\_best == n + 1)

cout << '-1';

else // Tìm được thì in ra cách trả đó.

{

cout << c\_best << endl;

for (int i = 1; i <= n; ++i)

if (x\_best[i])

cout << t[i] << ' ';

}

}

void branch\_and\_bound(int i)

{

// Nếu nghiệm mở rộng của nhánh này không tốt hơn thì return.

if (c + (s - sum)/t\_max[i] >= c\_best) return;

for (int j=0; j<=1; j++)

{

// Ghi nhận thành phần thứ i

x[i]=j;

sum += t[i]\*x[i];

c += j;

if (i==n) update\_best\_solution();

else if (sum <= s) branch\_and\_bound(i + 1);

// Loại bỏ thành phần thứ i

sum -= t[i]\*x[i];

c -= j;

}

}

int main()

{

enter();

create\_data();

branch\_and\_bound(1);

printf\_result();

return 0;

}

**Chương 4. Tham lam (Greedy Method)**

Phương pháp nhánh cận là cải tiến phương pháp quy lui, đã đánh giá được các nghiệm mở rộng để loại bỏ đi những phương án không cần thiết, giúp cho việc tìm nghiệm tối ưu nhanh hơn. Tuy nhiên, không phải lúc nào chúng ta cũng có thể đánh giá được nghiệm mở rộng, hoặc nếu có đánh giá được thì số phương án cần xét vẫn rất lớn, không thể đáp ứng được trong thời gian cho phép. Khi đó, người ta chấp nhận tìm những nghiệm gần đúng so với nghiệm tối ưu. Phương pháp tham lam được sử dụng trong các trường hợp như vậy. Ưu điểm nổi bật của phương pháp tham lam là độ phức tạp nhỏ, thường nhanh chóng tìm được lời giải.

**1. Phương pháp**

Giả sử nghiệm của bài toán có thể biểu diễn dưới dạng một vector (x1,x2,…,xn), mỗi thành phần xi (i = 1,2,…,n) được chọn ra từ tập Si. Mỗi nghiệm của bài toán X = (x1,x2,…,xn), được xác định “độ tốt” bằng một hàm *f(X)* và mục tiêu cần tìm nghiệm có giá trị *f(X)* càng lớn càng tốt (hoặc càng nhỏ càng tốt).

Tư tưởng của phương pháp tham lam như sau: Ta xây dựng vector nghiệm X dần từng bước, bắt đầu từ vector không ( ). Giả sử đã xây dựng được (k-1) thành phần (x1,x2,…,xk-1), của nghiệm và khi mở rộng nghiệm ta sẽ chọn xk *”tốt nhất”* trong các ứng cử viên trong tập Sk để được (x1,x2,…,xk). Việc lựa chọn như thế được thực hiện bởi một hàm chọn. Cứ tiếp tục xây dựng, cho đến khi xây dựng xong hết thành phần của nghiệm.

\_

Lược đồ tổng quát của phương pháp tham lam.

* Lược đồ 1:

**void Greedy1()  
{** Sort(D);  
 for (i=1; i<=n; i++)  
 {  
 - Chọn v là giá trị tốt nhất trong D và thỏa điều kiện bài toán

- xi = v;  
 - Bỏ v khỏi D  
 }  
**}**

* \* Lược đồ 2:

**void Greedy2()**  
**{**  
 X=();  
 for (i=1; i<=n; i++)  
 {  
 Xác định Di;  
 xi = SelectBest(Di); }  
**}**

Trong lược đồ tổng quát trên, SelectBest là hàm chọn, để chọn ra từ tập các ứng cử viên Si một ứng cử viên được xem là tốt nhất, nhiều hứa hẹn nhất.

Cần nhấn mạnh rằng, thuật toán tham lam trong một số bài toán, nếu xây dựng được hàm thích hợp có thể cho nghiệm tối ưu. Trong nhiều bài toán, thuật toán tham lam chỉ tìm được nghiệm gần đúng với nghiệm tối ưu.

**2. Bài toán người du lịch (Bài toán ở mục 2.2)**

Có nhiều thuật toán tham lam cho bài này, một thuật toán với ý tưởng đơn giản như sau: Xuất phát từ thành phố 1, tại mỗi bước ta sẽ chọn thành phố tiếp theo là thành phố chưa đến thăm mà chi phí từ thành phố hiện tại đến thành phố đó là nhỏ nhất, cụ thể:

+ Hành trình cần tìm có dạng (x1 = 1, x2, ..., xn, xn+1 = 1), trong đó dãy (x1, x2, ..., xn) lập thành một hoán vị của (1, 2, ..., n).

+ Ta xây dựng nghiệm từng bước, bắt đầu từ x1=1, chọn x2 là thành phố gần x1 nhất, sau đó chọn x3 là thành phố gần x2 nhất (x3 khác x1)... Tổng quát: chọn xi là thành phố chưa đi qua mà gần xi-1 nhất (2 i n).

#include <bits/stdc++.h>

#define MAX 100

#define oo 1e9+7

using namespace std;

long c[MAX+1][MAX+1];

long x[MAX+1],d[MAX+1]={};

long n, sum;

void input()

{

freopen("TSP.INP","r", stdin);

cin >> n;

for (int i = 1; i <= n; i++)

for (int j = 1; j <= n; j++)

cin >> c[i][j];

}

void output()

{

freopen("TSP.OUT","w",stdout);

cout << sum << endl;

for (int i = 1; i <= n; ++i)

cout << x[i] << "->";

cout << 1;

}

void Greedy()

{

x[1]=1;

d[1]=1;

int i=1;

while (i<n)

{

i++;

//Chọn ứng cử viên tốt nhất

long best=oo;

long xi;

for (int j=1; j<=n; j++)

if ((d[j]==0) && (c[x[i-1]][j]<best))

{

best=c[x[i-1]][j];

xi=j;

}

x[i]=xi; //ghi nhận thành phần nghiệm thứ i

d[xi]=1;

sum=sum+c[x[i-1]][x[i]];

}

sum=sum+c[x[n]][x[1]];

}

int main()

{

//ios\_base::sync\_with\_stdio(false);

//cin.tie(nullptr);

input();

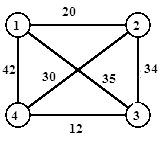
Greedy();

output();

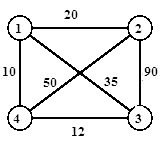
return 0;

}

**Ví dụ 1.** Xuất phát từ thành phố 1, ta xây dựng được hành trình 1🡪2🡪4🡪3🡪1 với chi phí 97, đây là phương án tối ưu.

****

**Ví dụ 2.** Xuất phát từ thành phố 1, ta xây dựng được hành trình 1🡪4🡪3🡪2🡪1 với chi phí 132, nhưng kết quả tối ưu là 117.

**3. Bài toán máy rút tiền tự động ATM (bài toán ở mục 2.3)**

Thuật toán với ý tưởng tham lam đơn giản, hàm chọn như sau: Tại mỗi bước ta sẽ chọn tờ tiền lớn nhất còn lại không vượt quá lượng tiền còn phải trả, cụ thể:

- Sắp xếp các tờ tiền giảm dần theo giá trị.

- Lần lượt xét các tờ tiền từ giá trị lớn đến giá trị nhỏ, nếu vẫn còn chưa lấy đủ S và tờ tiền đang xét có giá trị nhỏ hơn hoặc bằng S thì lấy luôn tờ tiền đó.

#include <bits/stdc++.h>

#define MAX 100

using namespace std;

int x[MAX+1]={},t[MAX+1];

int c, n, s;

void input()

{

freopen("ATM.INP","r", stdin);

cin >> n >> s;

for (int i = 1; i <= n; i++)

cin >> t[i];

}

void output()

{

freopen("ATM.OUT","w",stdout);

if (s==0) {

cout << c << endl;

for (int i = 1; i <= n; ++i)

if (x[i]==1) cout << t[i] << ' ';

}

else cout << '-1';

}

bool comp(int &i, int &j){

return i > j;

}

void Greedy()

{

//sắp xếp các tờ tiền theo thứ tự giảm dần

sort(t+1,t+n+1,comp);

c=0;

for (int i=1; i <= n; i++)

if(s>=t[i])

{

c++; //Tăng số lượng tờ tiền được lấy

x[i]=1; //Tờ i được lấy

s=s-t[i];

}

}

int main()

{

//ios\_base::sync\_with\_stdio(false);

//cin.tie(nullptr);

input();

Greedy();

output();

return 0;

}

Các bộ test thử nghiệm

|  |  |  |
| --- | --- | --- |
| test | Dữ liệu vào | Kết quả tìm được |
| 1 | 10 390  200 10 20 20 50 50 50 50 100 100 | 5  200 100 50 20 20 |
| 2 | 11 100  50 20 20 20 20 20 2 2 2 2 2 | 8  50 20 20 2 2 2 2 2 |
| 3 | 6 100  50 20 20 20 20 20 | -1 |

Với bộ test (1), thuật toán tham lam cũng cho được nghiệm tối ưu. Tuy nhiên, với bộ test (2), thuật toán tham lam không cho nghiệm tối ưu và với bộ test (3), thuật toán tham lam không tìm nghiệm mặc dù có nghiệm.

**4. Bài toán lập lịch giảm thiểu trễ hạn**

***Bài toán:*** Có n công việc đánh số từ 1 đến n và có một máy để thực hiện, biết:

- pi là thời gian cần thiết để hoàn thành công việc i.

- di là thời hạn hoàn thành công việc i.

Máy bắt đầu hoạt động từ thời điểm 0. Mỗi công việc cần được thực hiện liên tục từ lúc bắt đầu cho tới khi kết thúc, không được phép ngắt quãng. Giả sử ci là thời điểm hoàn thành công việc i. Khi đó, nếu ci > di ta nói công việc i bị hoàn thành trễ hạn, còn nếu ci ≤ di thì ta nói công việc i được hoàn thành đúng hạn.

*Yêu cầu:* Tìm trình tự thực hiện các công việc sao cho số công việc hoàn thành trễ hạn là ít nhất (hay số công việc hoàn thành đúng hạn là nhiều nhất).

**Input:** Tệp “JS.INP” có dạng:

- Dòng đầu là số n (n≤100) là số công việc

- Dòng thứ hai gồm n số là thời gian thực hiện các công việc

- Dòng thứ ba gồm n số là thời hạn hoàn thành các công việc

**Output:** Tệp “JS.OUT” có dạng: gồm một dòng là trình tự thực hiện các công việc.

Ví dụ: giả sử có 5 công việc với thời gian thực hiện và thời gian hoàn thành như sau:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| i | 1 | 2 | 3 | 4 | 5 |
| pi | 6 | 3 | 5 | 7 | 2 |
| di | 8 | 4 | 15 | 20 | 3 |

Nếu thực hiện theo thứ tự 1, 2, 3, 4, 5 thì sẽ có 3 công việc bị trễ hạn là công việc 2, 4 và 5. Còn nếu thực hiện theo thứ tự 5, 1, 3, 4, 2 thì chỉ có 1 công việc bị trễ hạn là công việc 2, đây là thứ tự thực hiện mà số công việc bị trễ hạn ít nhất (nghiệm tối ưu).

**Giải**

Ta có hai nhận xét sau:

+ Nếu thứ tự thực hiện các công việc mà có công việc bị trễ hạn được xếp trước một công việc đúng hạn thì ta sẽ nhận được trình tự tốt hơn bằng cách chuyển công việc trễ hạn xuống cuối cùng (vì đằng nào công việc này cũng bị trễ hạn).

*Ví dụ:* thứ tự 1, 2, 3, 4, 5 có công việc 2 bị trễ hạn xếp trước công việc 3 đúng hạn, ta chuyển công việc 2 xuống cuối cùng để nhận được thứ tự: 1, 3, 4, 5, 2, thứ tự này chỉ có 2 công việc bị quá hạn là công việc 5 và 2.

Như vậy, ta chỉ quan tâm đến việc xếp lịch cho các công việc hoàn thành đúng hạn, còn các công việc bị trễ hạn có thể thực hiện theo trình tự bất kì.

+ Giả sử Js là tập gồm k công việc (mà cả công việc này đều có thể thực hiện đúng hạn) và σ = (i1,i2,…,ik) là một hoán vị của các công việc trong Js sao cho di1≤di2≤…≤dik thì thứ tự σ là thứ tự để hoàn thành đúng hạn được cả k công việc.

*Ví dụ:* Js gồm 4 công việc 1, 3, 4, 5 (4 công việc này đều có thể thực hiện đúng hạn), ta có thứ tự thực hiện σ = (5,1,3,4) vì

d5 = 3 ≤ d1 = 8 ≤ d3 = 15 ≤ d4 =20

để cả 4 công việc đều thực hiện đúng hạn.

Sử dụng chiến lược tham lam, ta xây dựng tập công việc Js theo từng bước, ban đầu Js = ∅ . Hàm chọn được xây dựng như sau: tại mỗi bước ta sẽ chọn công việc Jobi mà có thời gian thực hiện nhỏ nhất trong số các công việc còn lại cho vào tập Js . Nếu sau khi kết nạp Jobi, các công việc trong tập Js đều có thể thực hiện đúng hạn thì cố định việc kết nạp Jobi vào tập Js, nếu không thì không kết nạp Jobi. Để đơn giản, ta giả sử rằng các công việc được đánh số theo thứ tự thời gian thực hiện tăng dần p1 ≤ p2 ≤ …≤ pn.

Sau đây là chương trình hoàn chỉnh:

#include <bits/stdc++.h>

#define MAX 101

using namespace std;

struct TJob {

int id, p, d;

} jobs[MAX], Js[MAX];

int d[MAX]={}, m, n;

void input()

{

freopen("JS.INP","r", stdin);

cin >> n;

for (int i=1; i<=n; i++) jobs[i].id=i;

for (int i=1; i<=n; i++) cin >> jobs[i].p;

for (int i=1; i<=n; i++) cin >> jobs[i].d;

}

void hoanvi(TJob &j1, TJob &j2)

{

TJob tmp = j1;

j1 = j2;

j2 = tmp;

}

bool check(TJob Jss[], int nJob)

{

//Sắp xếp lại công việc theo thứ tự tăng dần d trong k việc đầu tiên

for (int i=1; i<nJob; i++)

for (int j=i+1; j<=nJob; j++)

if (Jss[i].d > Jss[j].d) hoanvi(Jss[i],Jss[j]);

for (int i=1; i<=n; i++) cout << Jss[i].id << ' ';

int t=0;

for (int i=1; i<=nJob; i++)

{

if (t+Jss[i].p>Jss[i].d) return false;

t=t+Jss[i].p;

}

return true;

}

void gan\_mang(TJob J1[], TJob J2[])

{

for (int i=1; i<=n; i++)

J1[i]=J2[i];

}

void Greedy()

{

//Sắp xếp lại công việc theo thứ tự tăng dần p

for (int i=1; i<n; i++)

for (int j=i+1; j<=n; j++)

if (jobs[i].p>jobs[j].p) hoanvi(jobs[i],jobs[j]);

for (int i=1; i<=n; i++) cout << jobs[i].id << ' ';

m=0;

for (int i=1; i<=n; i++)

{

TJob Js2[MAX];

gan\_mang(Js2, Js);

Js2[m+1]=jobs[i];

if (check(Js2,m+1))

{

m++;

gan\_mang(Js, Js2);

d[i]=1;

}

}

for (int i=1; i<=n; i++)

if (d[i]==0){

m++;

Js[m]=jobs[i];

}

}

void printResult()

{

freopen("JS.OUT","w",stdout);

for (int i=1; i<=n; i++)

cout << Js[i].id << ' ';

}

int main()

{

input();

Greedy();

printResult();

return 0;

}

*Chú ý:* Thuật toán tham lam trình bày trên luôn cho phương án tối ưu.

**Chương 5. Chia để trị (Divide and Conquer)**

**1. Phương pháp**

Tư tưởng của chiến lược chia để trị như sau: Người ta phân bài toán cần giải thành các bài toán con. Các bài toán con lại được tiếp tục phân thành các bài toán con nhỏ hơn, cứ thế tiếp tục cho tới khi ta nhận được các bài toán con hoặc đã có thuật giải hoặc là có thể dễ ràng đưa ra thuật giải. Sau đó ta tìm cách kết hợp các nghiệm của các bài toán con để nhận được nghiệm của bài toán con lớn hơn, để cuối cùng nhận được nghiệm của bài toán cần giải. Thông thường các bài toán con nhận được trong quá trình phân chia là cùng dạng với bài toán ban đầu, chỉ có cỡ của chúng là nhỏ hơn.

Thuật toán chia để trị có thể biểu diễn bằng mô hình đệ quy như sau:

**void** DivideConquer(A,x); // tìm nghiệm x của bài toán A

**{**

**if** (A đủ nhỏ) **then** Solve(A)

//Solve(A) là thuật giải bài toán A trong trường hợp A có cỡ đủ nhỏ

**else**

**{**

[Phân A thành các bài toán con A1, A2,…,Am];

**for** (int i=1; i<=m; i++) DivideConquer(Ai, xi);

[Kết hợp các nghiệm xi (i=1,2,..,m) của các bài toán con Ai để nhận được nghiệm của bài toán A];

**}**

**}**

Trong thủ tục trên, Solve(A) là thuật giải bài toán A trong trường hợp A có cỡ đủ nhỏ.

Trong thuật toán tìm kiếm nhị phân và thuật toán sắp xếp nhanh-QuickSort (ở chuyên đề sắp xếp) là hai thuật toán được thiết kế dựa trên chiến lược chia để trị. Sau đây, chúng ta sẽ tìm hiểu một số ví dụ minh họa cho phương pháp chia để trị.

**2. Bài toán tính an**

***Bài toán:*** Cho số a và số nguyên dương n, tính an

Áp dụng kĩ thuật chia để trị, ta tính an dựa vào ak (trong đó k=n/2) như sau:

- nếu n chẵn: an = ak . ak

- nếu n lẻ: an = ak . ak . a1

Để tính ak ta lại dựa vào ak/2, quá trình chia nhỏ cho đến khi nhận được bài toán tính a1 thì dừng.

*Ví dụ:* tính 913

- bài toán được tính dựa trên bài toán con 96, ta có 913 = 96 . 96 . 91

- bài toán 96 được tính dựa trên bài toán con 93, ta có 96 = 93 . 93

- bài toán 93 được tính dựa trên bài toán con 91, ta có 93 = 91 . 91 . 91

Thủ tục đệ quy power(int a, int n) sau thể hiện ý tưởng trên.

#include <bits/stdc++.h>

using namespace std;

int a, n, p;

int power(int a, int n)

{

if (n==1) return a;

else

{

int tmp=power(a,n/2);

if (n%2==1) return tmp\*tmp\*a;

else return tmp\*tmp;

}

}

int main()

{

cout << "Nhap a, n:";

cin >> a >> n;

cout << power(a,n);

return 0;

}

**Chương 6. Quy hoạch động (Dynamic programming)**

**1. Phương pháp**

Trong chiến lược chia để trị, người ta phân bài toán cần giải thành các bài toán con. Các bài toán con lại được tiếp tục phân thành các bài toán con nhỏ hơn, cứ thế tiếp tục cho tới khi ta nhận được các bài toán con có thể giải được dễ dàng. Tuy nhiên, trong quá trình phân chia như vậy, có thể ta sẽ gặp rất nhiều lần cùng một bài toán con. Tư tưởng cơ bản của phương pháp quy hoạch động là sử dụng một bảng để lưu giữ lời giải của các bài toán con đã được giải. Khi giải một bài toán con cần đến nghiệm của bài toán con cỡ nhỏ hơn, ta chỉ cần lấy lời giải ở trong bảng mà không cần phải giải lại. Chính vì thế mà các thuật toán được thiết kế bằng quy hoạch động sẽ rất hiệu quả.

Để giải quyết một bài toán bằng phương pháp quy hoạch động, chúng ta cần tiến hành những công việc sau:

- Tìm nghiệm của các bài toán con nhỏ nhất.

- Tìm ra công thức (hoặc quy tắc) xây dựng nghiệm của bài toán con thông qua nghiệm của các bài toán con cỡ nhỏ hơn.

- Tạo ra một bảng lưu giữ các nghiệm của các bài toán con. Sau đó tính nghiệm của các bài toán con theo công thức đã tìm ra và lưu vào bảng.

- Từ các bài toán con đã giải để tìm nghiệm của bài toán.

Sau đây, chúng ta sẽ tìm hiểu một số ví dụ minh họa cho phương pháp quy hoạch động.

**2. Số Fibonacci**

Số Fibonacci được xác định bởi công thức:

F0 = 0

F1 = 1

Fn = Fn-1 + Fn-2 với n ≥ 2

*Cách 1:* Áp dụng phương pháp chia để trị, ta tính Fn dựa vào Fn-1 và Fn-2.

#include <bits/stdc++.h>

using namespace std;

**int** F(int n)

**{**

**if** (n <=1) return n;

**else** return (F(i - 1) + F(i - 2));

**}**

int main()

{

cout << "Nhap n:"; cin >> n;

cout << F(n) << endl;

return 0;

}

Hàm đệ quy F(n) để tính số Fibonacci thứ n. Ví dụ n= 6, 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).

F(6)

F(4)

F(5)

F(2)

F(3)

F(3)

F(4)

F(1)

F(2)

F(2)

F(1)

F(2)

F(3)

F(1)

F(2)

*Cách 2:* Phương pháp quy hoạch động.

Ta sử dụng mảng S[0..MaxN], S[i] để lưu lại lời giải cho bài toán tính số Fibonacci thứ i.

Ta nhận thấy, mỗi bài toán con chỉ được giải đúng một lần. Hãy cài đặt cả 2 chương trình trên và thử chạy với n = 40 để thấy được sự khác biệt!

Ta cũng có thể cài đặt phương pháp quy hoạch động cho bài toán như sau:

#include <bits/stdc++.h>

#define MaxN 51

using namespace std;

long long s[MaxN]={-1}; //Khởi tạo mảng s toàn các giá trị -1

long long n;

long long F(long long n)

{

if (s[n]=-1) //Bài toán chưa được giải thì tiến hành giải

{

if (n<=1) s[n]=n;

else s[n]=F(n-1) + F(n-2);

}

return s[n];

}

int main()

{

cout << "Nhap n:"; cin >> n;

cout << F(n) << endl;

//for (int i=0; i<=n; i++) cout << s[i] << ' ';

return 0;

}

Trước hết nó tính sẵn S[0] và S[1], từ đó tính tiếp S[2], lại tính tiếp được S[3], S[4],.., S[n]. Đảm bảo rằng mỗi giá trị Fibonacci chỉ phải tính 1 lần.

**3. Dãy con đơn điệu tăng dài nhất**

***Bài toán:*** Cho dãy số nguyên A = a1, a2, ..., an. (n ≤ 1000, -10000 ≤ ai ≤ 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 2n dãy con.

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

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

(1, 2, 3, 4, 5, 6, 7, 8).

***Giải***

Bổ sung vào A hai phần tử: a0 = -∞ và an+1 = +∞. Khi đó dãy con đơn điệu tăng dài nhất chắc chắn sẽ bắt đầu từ a0 và kết thúc ở an+1.

Với ∀i: 0 ≤ i ≤ n + 1. Ta sẽ tính L[i] = độ dài dãy con đơn điệu tăng dài nhất bắt đầu tại ai.

**a. Bài toán nhỏ nhất**

L[n + 1] = Độ dài dãy con đơn điệu tăng dài nhất bắt đầu tại an+1 = +.

Dãy con này chỉ gồm mỗi một phần tử (+) nên L[n + 1] = 1.

**b. Công thức**

Giả sử với i từ n đến 0, ta cần tính L[i]: độ dài dãy con tăng dài nhất bắt đầu tại ai. L[i] được tính trong điều kiện L[i + 1], L[i + 2], ..., L[n + 1] đã biết:

*Dãy con đơn điệu tăng dài nhất bắt đầu từ ai sẽ được thành lập bằng cách lấy ai ghép vào đầu một trong số những dãy con đơn điệu tăng dài nhất bắt đầu tại vị trí aj đứng sau ai.*

Ta sẽ chọn dãy nào để ghép ai vào đầu? Tất nhiên là chỉ được ghép ai vào đầu những dãy con bắt đầu tại aj nào đó lớn hơn ai (để đảm bảo tính tăng) và dĩ nhiên ta sẽ chọn dãy dài nhất để ghép ai vào đầu (để đảm bảo tính dài nhất). Vậy L[i] được tính như sau:

*Xét tất cả các chỉ số j trong khoảng từ i + 1 đến n + 1 mà aj > ai, chọn ra chỉ số jmax có L[jmax] lớn nhất. Đặt L[i] := L[jmax] + 1.*

**c. Truy vết**

Tại bước xây dựng dãy L, mỗi khi tính L[i] := L[jmax] + 1, ta đặt T[i] = jmax.

Để lưu lại: Dãy con dài nhất bắt đầu tại ai sẽ có phần tử thứ hai kế tiếp là ajmax. Sau khi tính xong hay dãy L và T, ta bắt đầu từ 0:

T[0] là phần tử đầu tiên được chọn,

T[T[0]] là phần tử thứ hai được chọn,

T[T[T[0]]] là phần tử thứ ba được chọn ...

Quá trình truy vết có thể diễn tả như sau:

i = T[0];

**while** (i != n + 1)

//Chừng nào chưa duyệt đến số an+1=+ở cuối

**{**

<Thông báo chọn ai>

i = T[i];

**}**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0  - | 1  5 | 2  2 | 3  3 | 4  4 | 5  9 | 6  10 | 7  5 | 8  6 | 9  7 | 10  8 | 11  + |
| ai | -∞ | 5 | 2 | 3 | 4 | 9 | 10 | 5 | 6 | 7 | 8 | +∞ |
| Length[i] | 9 | 5 | 8 | 7 | 6 | 3 | 2 | 5 | 4 | 3 | 2 | 1 |
| Trace[i] | 2 | 8 | 3 | 4 | 7 | 6 | 11 | 8 | 9 | 10 | 11 |  |

Truy vết

Ví dụ: với A = (5, 2, 3, 4, 9, 10, 5, 6, 7, 8). Hai dãy Length và Trace sau khi tính sẽ là:

#include <bits/stdc++.h>

#define MaxN 1002

#define oo 1e9+7

using namespace std;

int a[MaxN], L[MaxN], T[MaxN];

int n;

//Nhập dữ liệu

void Enter()

{

freopen("DONDIEU.INP","r", stdin);

cin >> n;

for (int i=1; i<=n; i++) cin >> a[i];

}

//Quy hoạch động

void Optimize()

{

a[0]=-oo; a[n+1]=oo; //Thêm hai phần tử canh hai đầu dãy a

L[n+1] = 1; //Điền cơ sở quy hoach động vào bảng phương án

for (int i=n; i>=0; i--) //Tính bảng phương án

{

//Chọn trong các chỉ số j đứng sau i thoả mãn a[j] > a[i] ra chỉ số jmax có L[jmax] lớn nhất

int jmax=n+1;

for (int j=i+1; j<=n+1; j++)

if ((a[j] > a[i]) && (L[j] > L[jmax]))

jmax=j;

//Lưu độ dài dãy con tăng dài nhất bắt đầu tại a[i]

L[i] = L[jmax] + 1;

//Lưu vết: phần tử đứng liền sau a[i] trong dãy con tăng dài nhất đó là a[jmax]

T[i] = jmax;

}

}

void InKQ()

{

freopen("DONDIEU.OUT","w",stdout);

//Chiều dài dãy con tăng dài nhất

cout << "Length of result : " << L[0]-2 << endl;

//Bắt đầu truy vết tìm nghiệm

int i=T[0];

while (i!=n+1)

{

cout << a[i] << ' ';

i=T[i];

}

}

int main()

{

Enter();

Optimize();

InKQ();

return 0;

}

**4. Dãy con chung dài nhất**

Cho hai số nguyên dương M, N (0 < M, N ≤ 100) và hai dãy số nguyên: A1, A2,...,AM và B1, B2,...,BN. Tìm một dãy dài nhất C là dãy con chung dài nhất của hai dãy A và B, nhận được từ A bằng cách xoá đi một số số hạng và cũng nhận được từ B bằng cách xoá đi một số số hạng.

**Input:** Tệp LCS.INP có dạng:

+ Dòng thứ nhất chứa số M, N

+ Dòng thứ hai chứa M số A1, A2,...,AM

+ Dòng thứ ba chứa N số B1, B2,...,BN.

**Output:** Tệp LCS.OUT có dạng:

+ Dòng thứ nhất ghi số k là số số hạng của dãy C.

+ Dòng thứ hai chứa k số là các số hạng của dãy C.

***Giải:***

Cần xây dựng mảng L[0..M, 0..N] với ý nghĩa: L[i, j] là độ dài của dãy chung dài nhất của hai dãy A[0.. i] và B[0..j].

Đương nhiên nếu một dãy là rỗng (số phần tử là 0) thì dãy con chung cũng là rỗng. Vì vậy, L[0, j] = 0 ∀j, j = 1.. N,

L[i, 0] = 0 ∀i, i = 1.. M.

Với M ≥ i > 0 và N ≥ j > 0 thì L[i, j] được tính theo công thức truy hồi sau:

L[i,j] = Max{L[i, j-1], L[i-1, j], L[i-1, j-1] + x}

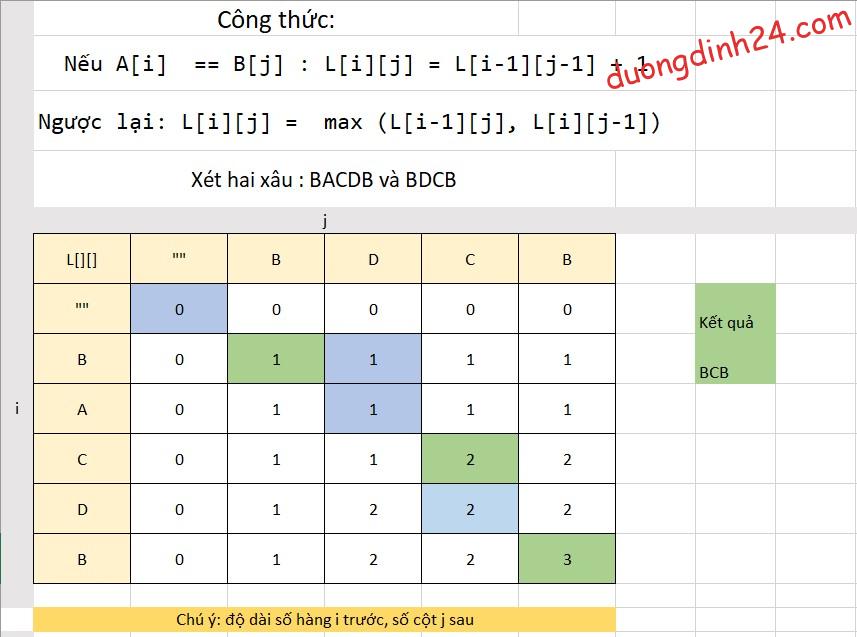
với x = 0 nếu A[i] ≠ B[j] ,

x = 1 nếu A[i]=B[j])

Tóm lại:

A number with numbers and symbols

Description automatically generated with medium confidence



#include<bits/stdc++.h>

#define fi "LCS.INP"

#define fo "LCS.OUT"

#define MaxMN 100

using namespace std;

int a[MaxMN], b[MaxMN], p[MaxMN], l[MaxMN][MaxMN];

int m, n, dem;

void Enter(){

    memset(p,0,sizeof(p));

    freopen(fi,"r",stdin);

    cin >> m >> n;

    for (int i=1; i<=m; i++)

        cin >> a[i];

    for (int i=1; i<=n; i++)

        cin >> b[i];

}

void Optimize(){

    memset(l,0,sizeof(l));

    for (int i=1; i<=m; i++)

        for (int j=1; j<=n; j++)

            if (a[i] == b[j])

                l[i][j] = l[i-1][j-1] + 1;

            else

                l[i][j] = max(l[i][j-1],l[i-1][j]);

}

void Trace(){

    freopen(fo,"w",stdout);

    cout << l[m][n] << endl;

    dem = 0;

    int i=m; int j=n;

    while ((i>0) && (j>0)){

        if (a[i]==b[j]){

            dem++;

            p[dem] = a[i];

            i--; j--;

        }

        else {

            if (l[i][j] == l[i][j-1]) j--;

            else i--;

        }

    }

    for (int i=dem; i>=1; i--)

        cout << p[i] << ' ';

}

int main(){

    ios\_base::sync\_with\_stdio(false);

    cin.tie(0); cout.tie(0);

    Enter();

    Optimize();

    Trace();

    return 0;

}

**5. Bài toán cái túi**

***Bài toán:*** Trong siêu thị có n gói hàng (n 100), gói hàng thứ i có trọng lượng là Wi (Wi 100) và trị giá Vi (Vi 100). Một tên trộm đột nhập vào siêu thị, sức của tên trộm không thể mang được trọng lượng vượt quá M (M 100). Hỏi tên trộm sẽ lấy đi những gói hàng nào để được tổng giá trị lớn nhất.

***Giải***

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

**a. Công thức tính B[i, j].**

Với giới hạn trọng lượng j, việc chọn tối ưu trong số các gói {1, 2, ...,i - 1, i} để có giá trị lớn nhất sẽ có hai khả năng:

 Nếu không chọn gói thứ i thì B[i, j] là giá trị lớn nhất có thể bằng cách chọn trong số các gói {1, 2, ..., i - 1} với giới hạn trọng lượng là j. Tức là

B[i, j] = B[i - 1, j]

 Nếu có chọn gói thứ i (tất nhiên chỉ xét tới trường hợp này khi mà Wi j) thì B[i, j] bằng giá trị gói thứ i là Vi cộng với giá trị lớn nhất có thể có được bằng cách chọn trong số các gói {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 nên nó sẽ là max trong hai giá trị thu được ở trên.

**b. Cơ sở quy hoạch động:**

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

**c. Tính bảng phương án:**

Bảng phương án B gồm n + 1 dòng, M + 1 cột, trước tiên được điền cơ sở quy hoạch động: Dòng 0 gồm toàn số 0. Sử dụng công thức truy hồi, dùng dòng 0 tính dòng 1, dùng dòng 1 tính dòng 2, v.v... đến khi tính hết dòng n.

0 1 ... M

0 0 0 0 0

1

2

... ...

n

**d. Truy vết:**

Tính xong bảng phương án thì ta quan tâm đến b[n, M] đó chính là giá trị lớn nhất

thu được khi chọn trong cả n gói với giới hạn trọng lượng M.

Nếu b[n, M] = b[n -1, M] thì tức là không chọn gói thứ n, ta truy tiếp b[n - 1, M]. Còn nếu b[n, M] b[n - 1, M] thì ta thông báo rằng phép chọn tối ưu có chọn gói thứ n và truy tiếp b[n - 1, M - Wn].

Cứ tiếp tục cho tới khi truy lên tới hàng 0 của bảng phương án.

#include <bits/stdc++.h>

using namespace std;

int k[100][100];

int n; long long m;

int a[100], b[100];

int max(int x, int y) {

   return (x > y) ? x : y;

}

int knapsack(long long m, int a[], int b[], int n) {

   int i, t;

   for (i = 0; i <= n; i++) {

      for (t = 0; t <= m; t++) {

         if (i == 0 || t == 0)

            k[i][t] = 0;

         else if (b[i - 1] <= t)

            k[i][t] = max(a[i - 1] + k[i - 1][t - b[i - 1]], k[i - 1][t]);

         else

        k[i][t] = k[i - 1][t];

      }

   }

   return k[n][m];

}

void Trace(){

   while (n != 0) {

      if (k[n][m] != k[n-1][m]){

         cout << "Pack " << n << " W= " << b[n-1] << " Value= " << a[n-1] << endl;

         m=m - a[n];

      }

      n--;

   }

}

int main() {

    //freopen("DRSEM.INP","r",stdin);

    //freopen("DRSEM.OUT","w",stdout);

    cin >> n >> m;

    for (int i = 0; i < n; i++) {

      cin >> a[i] >> b[i];

    }

   cout << knapsack(m, a, b, n) << endl;

   Trace();

   return 0;

}

**Sử dụng vecto:**

#include <bits/stdc++.h>

using namespace std;

int n, m;

struct Item {

    int weight;

    int value;

};

void knapsack(int n, int m, const vector<Item>& items) {

    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

    for (int i = 1; i <= n; i++) {

        for (int j = 0; j <= m; j++) {

                if (items[i-1].weight <= j) {

                    dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - items[i - 1].weight] + items[i - 1].value);

                }

                else {

                    dp[i][j] = dp[i - 1][j];

                }

        }

    }

    cout << dp[n][m] << endl;

    //Truy vet

    while (n != 0) {

      if (dp[n][m] != dp[n-1][m]){

         cout << "Pack " << n << " W= " << items[n-1].weight << " Value= " << items[n-1].value << endl;

         m = m - items[n-1].weight;

      }

      n--;

   }

}

int main() {

    //freopen("DRSEM.INP","r",stdin);

    //freopen("DRSEM.OUT","w",stdout);

    cin >> n >> m;

    vector<Item> items(n);

    for (int i = 0; i < n; i++) {

        cin >> items[i].value >> items[i].weight;

    }

    knapsack(n, m, items);

    return 0;

}

**6. Ví dụ về các bài toán có thể giải bằng phương pháp quy hoạch động**

**6.1. Bài toán Tính N!**

Ta có định nghĩa như sau: *n*! =  nếu 

Cho một số nguyên dương *n* (0 ≤ *n* ≤ 13).

**Yêu cầu:** Hãy tính *n*! bằng phương pháp quy hoạch động (lập bảng phương án).

**Dữ liệu vào:** Ghi trong file văn bản GT.INP có cấu trúc như sau:

*- Dòng 1:* Ghi số nguyên dương *n*.

**Dữ liệu ra**: Ghi ra file văn bản GT.OUT theo cấu trúc như sau:

*- Dòng 1:* Ghi giá trị tính được của *n*!

**Ví dụ:**

|  |  |
| --- | --- |
| GT.INP | GT.OUT |
| 3 | 6 |

**Thuật toán:**

Gọi GT[i] là giá trị của i! (0 ≤ *i* ≤ 13)

Ta có công thức quy hoạch động như sau:

GT[i] := GT[i-1]\*i;

Như vậy, việc tính *n*! sẽ được thực hiện bằng vòng lặp:

GT[0] :=1;

For i:=1 to n do

GT[i] := GT[i-1]\*i;

Kết quả: giá trị của *n*! nằm trong phần tử GT[n].

**6.2. Bài toán Tính dãy Fibonaci**

Ta có định nghĩa như sau: *F*(*n*) =  nếu   

Cho một số nguyên dương *n* (0 ≤ *n* ≤ 50).

***Yêu cầu:*** Hãy tính *F*(*n*) bằng phương pháp quy hoạch động (lập bảng phương án).

***Dữ liệu vào:*** Ghi trong file văn bản FIBO.INP có cấu trúc như sau:

*- Dòng 1:* Ghi số nguyên dương *n*.

***Dữ liệu ra***: Ghi ra file văn bản FIBO.OUT theo cấu trúc như sau:

*- Dòng 1:* Ghi giá trị tính được của *F*(*n)*.

**Ví dụ:**

|  |  |
| --- | --- |
| FIBO.INP | FIBO.OUT |
| 5 | 8 |

**Thuật toán:**

Gọi F[i] là giá trị Fibonaci của *f*i (0 ≤ *i* ≤ 50).

Ta có công thức quy hoạch động như sau:

F[i] := F[i-1] + F[i-2];

Như vậy, việc tính *f*n được thực hiện bằng vòng lặp:

F[0] := 0;

F[1] := 1;

For i := 2 to n do

F[i] := F[i-1] + F[i-2];

Kết quả: giá trị *f*n nằm trong F[n].

**6.3. Bài toán Tính tổng của dãy số**

Cho dãy số nguyên gồm *n* phần tử *a*1, *a*2, …, *an* (1 ≤ *n* ≤ 105) và hai số nguyên dương *p* và *q* (1 ≤ *p* ≤ *q* ≤ *n*).

**Yêu cầu:** Hãy tính tổng của các phần tử liên tiếp từ *ap* … *aq* bằng phương pháp quy hoạch động (lập bảng phương án).

**Dữ liệu vào:** Ghi trong file văn bản SUM.INP có cấu trúc như sau:

*- Dòng 1:* Ghi số nguyên dương *n* và *k*, hai số được ghi cách nhau một dấu cách.

*- Dòng 2:* Ghi *n* số nguyên *a*1, *a*2, …, *an*, các số được ghi cách nhau ít nhất một dấu cách (-32000 ≤ *a*i ≤ 32000).

*- Dòng thứ i trong k dòng tiếp theo:* Mỗi dòng ghi hai số nguyên dương *pi* và *qi*, hai số được ghi cách nhau một dấu cách (1 ≤ *pi* ≤ *qi* ≤ *n*).

**Dữ liệu ra**: Ghi ra file văn bản SUM.OUT theo cấu trúc như sau:

*- Dữ liệu được ghi trên k dòng:* Dòng thứ *i* ghi một số nguyên là tổng giá trị của các phần tử trong đoạn 

**Ví dụ:**

|  |  |
| --- | --- |
| SUM.INP | SUM.OUT |
| 5 3  2 9 -3 5 8  1 5  2 3  4 4 | 21  6  5 |

**Thuật toán:**

Gọi A[i] là giá trị của phần tử thứ *i* trong dãy số *a*1, *a*2, …, *an*.

Gọi T[i] là tổng giá trị các phần tử *a*1, *a*2, …, *ai* (1 ≤ *i* ≤ *n*).

Ta có công thức quy hoạch động để tính T[i] như sau:

T[i] := T[i - 1] + A[i];

Như vậy, việc tính T[n] được thực hiện bằng vòng lặp:

T[0] := 0;

For i:=1 to n do

T[i] := T[i - 1] + A[i];

Kết quả: Tổng các phần tử liên tiếp từ *a*p đến *a*q được tính theo công thức:

Sum := A[q] - A[p-1];