

HỒ SĨ ĐÀM (Chủ biên)
ĐỖ ĐỨC ĐÔNG – LÊ MINH HOÀNG – NGUYỄN THANH HÙNG

TÀI LIỆU GIÁO KHOA
CHUYÊN TIN
QUYỂN 1

NHÀ XUẤT BẢN GIÁO DỤC VIỆT NAM

Công ty Cổ phần dịch vụ xuất bản Giáo dục Hà Nội - Nhà xuất bản Giáo dục Việt Nam
giữ quyền công bố tác phẩm.

349-2009/CXB/43-644/GD

Mã số : 8I746H9

LỜI NÓI ĐẦU

Bộ Giáo dục và Đào tạo đã ban hành chương trình chuyên tin học cho các lớp chuyên 10, 11, 12. Dựa theo các chuyên đề chuyên sâu trong chương trình nói trên, các tác giả biên soạn bộ sách chuyên tin học, bao gồm các vấn đề cơ bản nhất về cấu trúc dữ liệu, thuật toán và cài đặt chương trình.

Bộ sách gồm ba quyển, quyển 1, 2 và 3. Cấu trúc mỗi quyển bao gồm: phần lý thuyết, giới thiệu các khái niệm cơ bản, cần thiết trực tiếp, thường dùng nhất; phần áp dụng, trình bày các bài toán thường gặp, cách giải và cài đặt chương trình; cuối cùng là các bài tập. Các chuyên đề trong bộ sách được lựa chọn mang tính hệ thống từ cơ bản đến chuyên sâu.

Với trải nghiệm nhiều năm tham gia giảng dạy, bồi dưỡng học sinh chuyên tin học của các trường chuyên có truyền thống và uy tín, các tác giả đã lựa chọn, biên soạn các nội dung cơ bản, thiết yếu nhất mà mình đã sử dụng để dạy học với mong muốn bộ sách phục vụ không chỉ cho giáo viên và học sinh chuyên PTTH mà cả cho giáo viên, học sinh chuyên tin học THCS làm tài liệu tham khảo cho việc dạy và học của mình.

Với kinh nghiệm nhiều năm tham gia bồi dưỡng học sinh, sinh viên tham gia các kì thi học sinh giỏi Quốc gia, Quốc tế Hội thi Tin học trẻ Toàn quốc, Olympiad Sinh viên Tin học Toàn quốc, Kì thi lập trình viên Quốc tế khu vực Đông Nam Á, các tác giả đã lựa chọn giới thiệu các bài tập, lời giải có định hướng phục vụ cho không chỉ học sinh mà cả sinh viên làm tài liệu tham khảo khi tham gia các kì thi trên.

Lần đầu tập sách được biên soạn, thời gian và trình độ có hạn chế nên chắc chắn còn nhiều thiếu sót, các tác giả mong nhận được ý kiến đóng góp của bạn đọc, các đồng nghiệp, sinh viên và học sinh để bộ sách được ngày càng hoàn thiện hơn.

Các tác giả

THUẬT TOÁN VÀ PHÂN TÍCH THUẬT TOÁN

1. Thuật toán

Thuật toán là một trong những khái niệm quan trọng nhất trong tin học. Thuật ngữ thuật toán xuất phát từ nhà khoa học Ả-rập Abu Ja'far Mohammed ibn Musa al Khowarizmi. Ta có thể hiểu *thuật toán là dãy hữu hạn các bước, mỗi bước mô tả chính xác các phép toán hoặc hành động cần thực hiện, để giải quyết một vấn đề*. Để hiểu đầy đủ ý nghĩa của khái niệm thuật toán chúng ta xem xét 5 đặc trưng sau của thuật toán:

- Đầu vào (Input): Thuật toán nhận dữ liệu vào từ một tập nào đó.
- Đầu ra (Output): Với mỗi tập các dữ liệu đầu vào, thuật toán đưa ra các dữ liệu tương ứng với lời giải của bài toán.
- Chính xác: Các bước của thuật toán được mô tả chính xác.
- Hữu hạn: Thuật toán cần phải đưa được đầu ra sau một số hữu hạn (có thể rất lớn) bước với mọi đầu vào.
- Đơn trị: Các kết quả trung gian của từng bước thực hiện thuật toán được xác định một cách đơn trị và chỉ phụ thuộc vào đầu vào và các kết quả của các bước trước.
- Tổng quát: Thuật toán có thể áp dụng để giải mọi bài toán có dạng đã cho.

Để biểu diễn thuật toán có thể biểu diễn bằng danh sách các bước, các bước được diễn đạt bằng ngôn ngữ thông thường và các kí hiệu toán học; hoặc có thể biểu diễn thuật toán bằng sơ đồ khối. Tuy nhiên, để đảm bảo tính xác định của thuật toán, thuật toán cần được viết bằng các ngôn ngữ lập trình. Một chương trình là sự biểu diễn của một thuật toán trong ngôn ngữ lập trình đã chọn. Trong tài liệu này, chúng ta sử dụng ngôn ngữ tựa Pascal để trình bày các thuật toán. Nói là tựa Pascal, bởi vì nhiều trường hợp, để cho ngắn gọn, chúng ta không hoàn toàn tuân

theo quy định của Pascal. Ngôn ngữ Pascal là ngôn ngữ đơn giản, khoa học, được giảng dạy trong nhà trường phổ thông.

Ví dụ: Thuật toán kiểm tra tính nguyên tố của một số nguyên dương n ($n \geq 2$), viết trên ngôn ngữ lập trình Pascal.

```
function is_prime(n):boolean;  
begin  
    for k:=2 to n-1 do  
        if (n mod k=0) then exit(false);  
    exit(true);  
end;
```

2. Phân tích thuật toán

2.1. Tính hiệu quả của thuật toán

Khi giải một bài toán, chúng ta cần chọn trong số các thuật toán một thuật toán mà chúng ta cho là “tốt” nhất. Vậy dựa trên cơ sở nào để đánh giá thuật toán này “tốt” hơn thuật toán kia? Thông thường ta dựa trên hai tiêu chuẩn sau:

1. Thuật toán đơn giản, dễ hiểu, dễ cài đặt (dễ viết chương trình).
2. Thuật toán hiệu quả: Chúng ta thường đặc biệt quan tâm đến thời gian thực hiện của thuật toán (gọi là độ phức tạp tính toán), bên cạnh đó chúng ta cũng quan tâm tới dung lượng không gian nhớ cần thiết để lưu giữ các dữ liệu vào, ra và các kết quả trung gian trong quá trình tính toán.

Khi viết chương trình chỉ để sử dụng một số ít lần thì tiêu chuẩn (1) là quan trọng, nhưng nếu viết chương trình để sử dụng nhiều lần, cho nhiều người sử dụng thì tiêu chuẩn (2) lại quan trọng hơn. Trong trường hợp này, dù thuật toán có thể phải cài đặt phức tạp, nhưng ta vẫn sẽ lựa chọn để nhận được chương trình chạy nhanh hơn, hiệu quả hơn.

2.2. Tại sao cần thuật toán có tính hiệu quả?

Kỹ thuật máy tính tiên bộ rất nhanh, ngày nay các máy tính lớn có thể đạt tốc độ tính toán hàng nghìn tỉ phép tính trong một giây. Vậy có cần phải tìm thuật toán hiệu quả hay không? Chúng ta **xem lại ví dụ** bài toán kiểm tra tính nguyên tố của một số nguyên dương n ($n \geq 2$).

```
function is_prime(n):boolean;  
begin
```

```

for k:=2 to n-1 do
    if (n mod k=0) then exit(false);
    exit(true);
end;

```

Đễ dàng nhận thấy rằng, nếu n là một số nguyên tố chúng ta phải mất $n - 2$ phép toán *mod*. Giả sử một siêu máy tính có thể tính được trăm nghìn tỉ (10^{14}) phép *mod* trong một giây, như vậy để kiểm tra một số khoảng 25 chữ số mất khoảng $\frac{10^{25}}{10^{14} \times 60 \times 60 \times 24 \times 365} \sim 3170$ năm. Trong khi đó, nếu ta có nhận xét việc thử k từ 2 đến $n - 1$ là không cần thiết mà chỉ cần thử k từ 2 đến \sqrt{n} , ta có:

```

function is_prime(n):boolean;
begin
    for k:=2 to trunc(sqrt(n)) do
        if (n mod k=0) then exit(false);
        exit(true);
    end;
{hàm sqrt(n) là hàm tính  $\sqrt{n}$ , trunc(x) là hàm làm tròn x }

```

Như vậy để kiểm tra một số khoảng 25 chữ số mất khoảng $\frac{\sqrt{10^{25}}}{10^{14}} \sim 0.03$ giây!

2.3. Đánh giá thời gian thực hiện thuật toán

Có hai cách tiếp cận để đánh giá thời gian thực hiện của một thuật toán. Cách thứ nhất bằng thực nghiệm, chúng ta viết chương trình và cho chạy chương trình với các dữ liệu vào khác nhau trên một máy tính. Cách thứ hai bằng phương pháp lí thuyết, chúng ta coi thời gian thực hiện thuật toán như hàm số của cỡ dữ liệu vào (cỡ của dữ liệu vào là một tham số đặc trưng cho dữ liệu vào, nó có ảnh hưởng quyết định đến thời gian thực hiện chương trình. Ví dụ đối với bài toán kiểm tra số nguyên tố thì cỡ của dữ liệu vào là số n cần kiểm tra; hay với bài toán sắp xếp dãy số, cỡ của dữ liệu vào là số phần tử của dãy). Thông thường cỡ của dữ liệu vào là một số nguyên dương n , ta sử dụng hàm số $T(n)$ trong đó n là cỡ của dữ liệu vào để biểu diễn thời gian thực hiện của một thuật toán.

Xét ví dụ bài toán kiểm tra tính nguyên tố của một số nguyên dương n (cỡ dữ liệu vào là n), nếu n là một số chẵn ($n > 2$) thì chỉ cần một lần thử chia 2 để kết luận n không phải là số nguyên tố. Nếu n ($n > 3$) không chia hết cho 2 nhưng lại chia hết cho 3 thì cần 2 lần thử (chia 2 và chia 3) để kết luận n không nguyên tố. Còn nếu n là một số nguyên tố thì thuật toán phải thực hiện nhiều lần thử nhất.

Trong tài liệu này, chúng ta hiểu hàm số $T(n)$ là thời gian nhiều nhất cần thiết để thực hiện thuật toán với mọi bộ dữ liệu đầu vào cỡ n .

Sử dụng kí hiệu toán học ô lớn để mô tả độ lớn của hàm $T(n)$. Giả sử n là một số nguyên dương, $T(n)$ và $f(n)$ là hai hàm thực không âm. Ta viết $T(n) = O(f(n))$ nếu và chỉ nếu tồn tại các hằng số dương c và n_0 , sao cho $T(n) \leq c \times f(n)$, với mọi $n \geq n_0$.

Nếu một thuật toán có thời gian thực hiện $T(n) = O(f(n))$ chúng ta nói rằng thuật toán có thời gian thực hiện cấp $f(n)$.

Ví dụ: Giả sử $T(n) = n^2 + 2n$, ta có $n^2 + 2n \leq n^2 + 2n^2 = 3n^2$ với mọi $n \geq 1$

Vậy $T(n) = O(n^2)$, trong trường hợp này ta nói thuật toán có thời gian thực hiện cấp n^2 .

2.4. Các quy tắc đánh giá thời gian thực hiện thuật toán

Để đánh giá thời gian thực hiện thuật toán được trình bày bằng ngôn ngữ tựa Pascal, ta cần biết cách đánh giá thời gian thực hiện các câu lệnh của Pascal. Trước tiên, chúng ta hãy xem xét các câu lệnh chính trong Pascal. Các câu lệnh trong Pascal được định nghĩa đệ quy như sau:

1. Các phép gán, đọc, viết là các câu lệnh (được gọi là lệnh đơn).

2. Nếu S_1, S_2, \dots, S_m là câu lệnh thì

```
Begin S1; S2; ...; Sm; End;
```

là câu lệnh (được gọi là lệnh hợp thành hay khối lệnh).

3. Nếu S_1 và S_2 là các câu lệnh và E là biểu thức logic thì

```
If E then S1 else S2;
```

là câu lệnh (được gọi là lệnh rẽ nhánh hay lệnh If).

4. Nếu S là câu lệnh và E là biểu thức logic thì

```
While E do S;
```

là câu lệnh (được gọi là lệnh lặp điều kiện trước hay lệnh While).

5. Nếu S_1, S_2, \dots, S_m là các câu lệnh và E là biểu thức logic thì

```
Repeat  
S1; S2; ...; Sm;  
Until E;
```

là câu lệnh (được gọi là lệnh lặp điều kiện sau hay lệnh Repeat)

6. Nếu S là lệnh, E_1 và E_2 là các biểu thức cùng một kiểu thứ tự đếm được thì

`For i:=E1 to E2 do S;`

là câu lệnh (được gọi là lệnh lặp với số lần xác định hay lệnh For).

Để đánh giá, chúng ta phân tích chương trình xuất phát từ các lệnh đơn, rồi đánh giá các lệnh phức tạp hơn, cuối cùng đánh giá được thời gian thực hiện của chương trình, cụ thể:

1. Thời gian thực hiện các lệnh đơn: gán, đọc, viết là $O(1)$
2. Lệnh hợp thành: giả sử thời gian thực hiện của S_1, S_2, \dots, S_m tương ứng là $O(f_1(n)), O(f_2(n)), \dots, O(f_m(n))$. Khi đó thời gian thực hiện của lệnh hợp thành là: $O(\max(f_1(n), f_2(n), \dots, f_m(n)))$.
3. Lệnh If: giả sử thời gian thực hiện của S_1, S_2 tương ứng là $O(f_1(n)), O(f_2(n))$. Khi đó thời gian thực hiện của lệnh If là: $O(\max(f_1(n), f_2(n)))$.
4. Lệnh lặp While: giả sử thời gian thực hiện lệnh S (thân của lệnh While) là $O(f(n))$ và $g(n)$ là số lần lặp tối đa thực hiện lệnh S . Khi đó thời gian thực hiện lệnh While là $O(f(n)g(n))$.
5. Lệnh lặp Repeat: giả sử thời gian thực hiện khối lệnh

`Begin S1; S2; ...; Sm; End;`

là $O(f(n))$ và $g(n)$ là số lần lặp tối đa. Khi đó thời gian thực hiện lệnh Repeat là $O(f(n)g(n))$.
6. Lệnh lặp For: giả sử thời gian thực hiện lệnh S là $O(f(n))$ và $g(n)$ là số lần lặp tối đa. Khi đó thời gian thực hiện lệnh For là $O(f(n)g(n))$.

2.5. Một số ví dụ

Ví dụ 1: Phân tích thời gian thực hiện của chương trình sau:

```
var i, j, n      :longint;
    s1, s2      :longint;
BEGIN
{1}    readln(n);
{2}    s1:=0;
{3}    for i:=1 to n do
{4}        s1:=s1 + i;
{5}    s2:=0;
```

```

{6}    for j:=1 to n do
{7}        s2:=s2 + j*j;
{8}    writeln('1+2+...+',n,'=',s1);
{9}    writeln('1^2+2^2+...+',n,'^2=',s2);
END.

```

Thời gian thực hiện chương trình phụ thuộc vào số n .

Các lệnh {1}, {2}, {4}, {5}, {7}, {8}, {9} có thời gian thực hiện là $O(1)$.

Lệnh lặp For {3} có số lần lặp là n , như vậy lệnh {3} có thời gian thực hiện là $O(n)$. Tương tự lệnh lặp For {6} cũng có thời gian thực hiện là $O(n)$.

Vậy thời gian thực hiện của chương trình là:

$$\max(O(1), O(1), O(n), O(1), O(n), O(1), O(1)) = O(n)$$

Ví dụ 2: Phân tích thời gian thực hiện của đoạn chương trình sau:

```

{1}    c:=0;
{2}    for i:=1 to 2*n do
{3}        c:=c+1;
{4}    for i:=1 to n do
{5}        for j:=1 to n do
{6}            c:=c+1;

```

Thời gian thực hiện chương trình phụ thuộc vào số n .

Các lệnh {1}, {3}, {6} có thời gian thực hiện là $O(1)$.

Lệnh lặp For {2} có số lần lặp là $2n$, như vậy lệnh {2} có thời gian thực hiện là $O(n)$.

Lệnh lặp For {5} có số lần lặp là n , như vậy lệnh {5} có thời gian thực hiện là $O(n)$. Lệnh lặp For {4} có số lần lặp là n , như vậy lệnh {4} có thời gian thực hiện là $O(n^2)$.

Vậy thời gian thực hiện của đoạn chương trình trên là:

$$\max(O(1), O(n), O(n^2)) = O(n^2)$$

Ví dụ 3: Phân tích thời gian thực hiện của đoạn chương trình sau:

```

{1}    for i:=1 to n do
{2}        for j:=1 to i do
{3}            c:=c+1;

```

Thời gian thực hiện chương trình phụ thuộc vào số n .

Các lệnh {3} có thời gian thực hiện là $O(1)$.

Khi $i = 1$, j chạy từ 1 đến 1 \rightarrow lệnh lặp For {2} lặp 1 lần

Khi $i = 2$, j chạy từ 1 đến 2 \rightarrow lệnh lặp For {2} lặp 2 lần

...

Khi $i = n$, j chạy từ 1 đến $n \rightarrow$ lệnh lặp For {2} lặp n lần

Như vậy lệnh {3} được lặp: $1 + 2 + \dots + n = \frac{n(n+1)}{2}$ lần, do đó lệnh {1} có thời gian thực hiện là $O(n^2)$

Vậy thời gian thực hiện của đoạn chương trình trên là: $O(n^2)$

Bài tập

1.1. Phân tích thời gian thực hiện của đoạn chương trình sau:

```
for i:=1 to n do
    if i mod 2=0 then c:=c+1;
```

1.2. Phân tích thời gian thực hiện của đoạn chương trình sau:

```
for i:=1 to n do
    if i mod 2=0 then c1:=c1+1
    else c2:=c2+1;
```

1.3. Phân tích thời gian thực hiện của đoạn chương trình sau:

```
for i:=1 to n do
    if i mod 2=0 then
        for j:=1 to n do c:=c+1
```

1.4. Phân tích thời gian thực hiện của đoạn chương trình sau:

```
a:=0;
b:=0;
c:=0;
for i:=1 to n do
    begin
        a:=a + 1;
        b:=b + i;
        c:=c + i*i;
    end;
```

1.5. Phân tích thời gian thực hiện của đoạn chương trình sau:

```
i:=n;
d:=0;
```

```
while i>0 do
  begin
    i:=i-1;
    d:=d + i;
  end;
```

1.6. Phân tích thời gian thực hiện của đoạn chương trình sau:

```
i:=0;
d:=0;
repeat
  i:=i+1;
  if i mod 3=0 then d:=d + i;
until i>n;
```

1.7. Phân tích thời gian thực hiện của đoạn chương trình sau:

```
d:=0;
for i:=1 to n-1 do
  for j:=i+1 to n do d:=d+1;
```

1.8. Phân tích thời gian thực hiện của đoạn chương trình sau:

```
d:=0;
for i:=1 to n-2 do
  for j:=i+1 to n-1 do
    for k:=j+1 to n do d:=d+1;
```

1.9. Phân tích thời gian thực hiện của đoạn chương trình sau:

```
d:=0;
while n>0 do
  begin
    n:=n div 2;
    d:=d+1;
  end;
```

1.10. Cho một dãy số gồm n số nguyên dương, xác định xem có tồn tại một dãy con liên tiếp có tổng bằng k hay không?

- a) Đưa ra thuật toán có thời gian thực hiện $O(n^3)$.
- b) Đưa ra thuật toán có thời gian thực hiện $O(n^2)$.
- c) Đưa ra thuật toán có thời gian thực hiện $O(n)$.

CÁC KIẾN THỨC CƠ BẢN

1. Hệ đếm

Hệ đếm được hiểu là tập các kí hiệu và quy tắc sử dụng tập các kí hiệu đó để biểu diễn và xác định giá trị các số. Trong hệ đếm cơ số b ($b > 1$), các kí hiệu được dùng có các giá trị tương ứng $0, 1, \dots, b - 1$. Giả sử N có biểu diễn:

$$d_n d_{n-1} d_{n-2} \dots d_1 d_0, d_{-1} d_{-2} \dots d_{-m}$$

trong đó $n + 1$ số các chữ số bên trái, m là số các chữ số bên phải dấu phân chia phần nguyên và phần phân của số N và các d_i phải thoả mãn điều kiện

$$0 \leq d_i < b \quad (-m \leq i \leq n).$$

Khi đó giá trị của số N được tính theo công thức:

$$N = d_n b^n + d_{n-1} b^{n-1} + \dots + d_0 b^0 + d_{-1} b^{-1} + \dots + d_{-m} b^{-m} \quad (1)$$

Chú ý: Để phân biệt số được biểu diễn ở hệ đếm nào người ta viết cơ số làm chỉ số dưới của số đó. Ví dụ: N_b là biểu diễn N ở hệ đếm b .

1.1. Các hệ đếm thường dùng:

Hệ thập phân (hệ cơ số 10) dùng 10 kí hiệu 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Ví dụ: $28,9_{10} = 2 \times 10^1 + 8 \times 10^0 + 9 \times 10^{-1}$

Hệ nhị phân (hệ cơ số 2) chỉ dùng hai kí hiệu 0, 1

Ví dụ: $10_2 = 1 \times 2^1 + 0 \times 2^0 = 2_{10}$

$101,1_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} = 5,5$

Hệ cơ số mười sáu, còn gọi là hệ hexa, sử dụng các kí hiệu 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, trong đó A, B, C, D, E, F có các giá trị tương ứng 10, 11, 12, 13, 14, 15 trong hệ thập phân

Ví dụ: $AF0_{16} = 10 \times 16^2 + 15 \times 16^1 + 0 \times 16^0 = 2800_{10}$

1.2. Chuyển đổi biểu diễn số ở hệ thập phân sang hệ đếm cơ số khác

Để chuyển đổi biểu diễn một số ở hệ thập phân sang hệ đếm cơ số khác, trước hết ta tách phần nguyên và phần phân rồi tiến hành chuyển đổi từng phần, sau đó ghép lại.

Chuyển đổi biểu diễn phần nguyên: Từ (1) ta lấy phần nguyên:

$$X = d_n b^n + d_{n-1} b^{n-1} + \dots + d_0 \text{ (trong đó } 0 \leq d_i < b \text{)}.$$

Do $0 \leq d_0 < b$ nên khi chia X cho b thì phần dư của phép chia đó là d_0 còn thương số $X1$ sẽ là: $d_n b^{n-1} + d_{n-1} b^{n-2} + \dots + d_1$. Tương tự d_1 là phần dư của phép chia $X1$ cho b . Quá trình được lặp cho đến khi nhận được thương bằng 0.

Chuyển đổi biểu diễn phần phân: Từ (1) ta lấy phần sau dấu phẩy:

$$Y = d_{-1} b^{-1} + \dots + d_{-m} b^{-m}.$$

$$Y1 = Y \times b = d_{-1} + d_{-2} b^{-1} + \dots + d_{-m} b^{-(m-1)}$$

Ta nhận thấy d_{-1} chính là phần nguyên của kết quả phép nhân, còn phần phân của kết quả là $Y2 = d_{-2} b^{-1} + \dots + d_{-m} b^{-(m-1)}$. Quá trình được lặp cho đến khi nhận đủ số chữ số cần tìm.

2. Số nguyên tố

Một số tự nhiên p ($p > 1$) là số nguyên tố nếu p có đúng hai ước số là 1 và p .

Ví dụ các số nguyên tố: 2, 3, 5, 7, 11, 13, 17, 19, 23, ...

2.1. Kiểm tra tính nguyên tố

a) Để kiểm tra số nguyên dương n ($n > 1$) có là số nguyên tố không, ta kiểm tra xem có tồn tại một số nguyên k ($2 \leq k \leq n - 1$) mà k là ước của n (n chia hết k) thì n không phải là số nguyên tố, ngược lại n là số nguyên tố.

Nếu n ($n > 1$) không phải là số nguyên tố, ta luôn có thể tách $n = k_1 \times k_2$ mà $2 \leq k_1 \leq k_2 \leq n - 1$. Vì $k_1 \times k_1 \leq k_1 \times k_2 = n$ nên $k_1 \leq \sqrt{n}$. Do đó, việc kiểm tra với k từ 2 đến $n - 1$ là không cần thiết, mà chỉ cần kiểm tra k từ 2 đến \sqrt{n} .

```
function is_prime(n:longint):boolean;  
var k :longint;  
begin  
    if n=1 then exit(false);
```

```

for k:=2 to trunc(sqrt(n)) do
  if (n mod k=0) then exit(false);
exit(true);
end;

```

Hàm `is_prime(n)` trên tiến hành kiểm tra lần lượt từng số nguyên k trong đoạn $[2, \sqrt{n}]$, để cải tiến, cần giảm thiểu số các số cần kiểm tra. Ta có nhận xét, để kiểm tra số nguyên dương n ($n > 1$) có là số nguyên tố không, ta kiểm tra xem có tồn tại một số nguyên tố k ($2 \leq k \leq \sqrt{n}$) mà k là ước của n thì n không phải là số nguyên tố, ngược lại n là số nguyên tố. Thay vì kiểm tra các số k là nguyên tố ta sẽ chỉ kiểm tra các số k có tính chất giống với tính chất của số nguyên tố, có thể sử dụng một trong hai tính chất đơn giản sau của số nguyên tố:

- 1) Trừ số 2 và các số nguyên tố là số lẻ.
- 2) Trừ số 2, số 3 các số nguyên tố có dạng $6k \pm 1$ (vì số có dạng $6k \pm 2$ thì chia hết cho 2, số có dạng $6k \pm 3$ thì chia hết cho 3).

Hàm `is_prime2(n)` dưới đây kiểm tra tính nguyên tố của số n bằng cách kiểm tra xem n có chia hết cho số 2, số 3 và các số có dạng $6k \pm 1$ trong đoạn $[5, \sqrt{n}]$.

```

function is_prime2(n:longint):boolean;
var k,sqrt_n:longint;
begin
  if (n=2) or (n=3) then exit(true);
  if (n=1) or (n mod 2=0) or (n mod 3=0) then exit(false);
  sqrt_n:=trunc(sqrt(n));
  k:=-1;
  repeat
    inc(k,6);
    if (n mod k=0) or (n mod (k+2)=0) then break;
  until k>sqrt_n;
  exit(k>sqrt_n);
end;

```

b) Phương pháp kiểm tra số nguyên tố theo xác suất

Từ định lý nhỏ Fermat:

|| nếu p là số nguyên tố và a là số tự nhiên thì $a^p \bmod p = a$

Ta có cách kiểm tra tính nguyên tố của Fermat:

$\text{nếu } 2^n \bmod n \neq 2 \text{ thì } n \text{ không là số nguyên tố}$
 $\text{nếu } 2^n \bmod n = 2 \text{ thì nhiều khả năng } n \text{ là số nguyên tố}$

Ví dụ:

$2^9 \bmod 9 = 512 \bmod 9 = 8 \neq 2$, do đó số 9 không là số nguyên tố.

$2^3 \bmod 3 = 8 \bmod 3 = 2$, do đó nhiều khả năng 3 là số nguyên tố, thực tế 3 là số nguyên tố.

$2^{11} \bmod 11 = 2048 \bmod 11 = 2$, do đó nhiều khả năng 11 là số nguyên tố, thực tế 11 là số nguyên tố.

2.2. Liệt kê các số nguyên tố trong đoạn $[1, N]$

Cách thứ nhất là thử lần lượt các số m trong đoạn $[1, N]$, rồi kiểm tra tính nguyên tố của m .

```

procedure generate(N:longint);
var m :longint;
begin
  for m:=2 to N do
    if is_prime(m) then writeln(m);
end;

```

Cách này đơn giản nhưng chạy chậm, để cải tiến có thể sử dụng các tính chất của số nguyên tố để loại bỏ trước những số không phải là số nguyên tố và không cần kiểm tra các số này.

Cách thứ hai là sử dụng sàng số nguyên tố, như sàng Eratosthene, liệt kê được các số nguyên tố nhanh, tuy nhiên nhược điểm của cách này là tốn nhiều bộ nhớ. Cách làm được thực hiện như sau:

Trước tiên xoá bỏ số 1 ra khỏi tập các số nguyên tố. Số tiếp theo số 1 là số 2, là số nguyên tố, xoá tất cả các bội của 2 ra khỏi bảng. Số đầu tiên không bị xoá sau số 2 (số 3) là số nguyên tố, xoá các bội của 3... Giải thuật tiếp tục cho đến khi gặp số nguyên tố lớn hơn \sqrt{N} thì dừng lại. Tất cả các số chưa bị xoá là số nguyên tố.

```

{$M 1100000}
procedure Eratosthene(N:longint);
const  MAX      = 1000000;
var     i,j      :longint;
        Prime    :array [1..MAX] of byte;
begin

```



```

fillchar(Prime,sizeof(Prime),0);
for i:=2 to trunc(sqrt(N)) do
  if Prime[i]=0 then
    begin
      j:=i*i;
      while j<=N do
        begin
          Prime[j]:=1;
          j:=j+i;
        end;
    end;
  for i:=2 to N do
    if Prime[i]=0 then writeln(i);
  end;

```

3. Ước số, bội số

3.1. Số các ước số của một số

Giả sử N được phân tích thành thừa số nguyên tố như sau:

$$N = a^i \times b^j \times \dots \times c^k$$

Ước số của N có dạng: $a^p \times b^q \times \dots \times c^r$ trong đó

$$0 \leq p \leq i, 0 \leq q \leq j, \dots, 0 \leq r \leq k.$$

Do đó, số các ước số của N là $(i+1) \times (j+1) \times \dots \times (k+1)$.

Ví dụ:

$N = 100 = 2^2 \times 5^2$, số ước số của 100 là: $(2+1)(2+1) = 9$ ước số (các ước số đó là: 1, 2, 4, 5, 10, 20, 25, 50, 100).

$N = 24 = 2^3 \times 3$, số ước số của 24 là: $(3+1)(1+1) = 8$ ước số (các ước số đó là: 1, 2, 3, 4, 6, 8, 12, 24).

3.2. Tổng các ước số của một số

$$N = a^i \times b^j \times \dots \times c^k$$

Đặt $N_1 = b^j \times \dots \times c^k$

Gọi $F(t)$ là tổng các ước của t , ta có,

$$F(N) = F(N_1) + a \times F(N_1) + \dots + a^i \times F(N_1)$$

$$\begin{aligned}
&= (1 + a + \dots + a^i) \times F(N1) = \frac{(a^{i+1} - 1)}{a - 1} \times F(N1) \\
&= \frac{(a^{i+1} - 1)}{a - 1} \times \frac{(b^{j+1} - 1)}{b - 1} \times \dots \times \frac{(c^{k+1} - 1)}{c - 1}
\end{aligned}$$

Ví dụ: Tổng các ước của 24 là:

$$\frac{(2^{3+1} - 1)}{2 - 1} \times \frac{(3^{1+1} - 1)}{3 - 1} = 60$$

3.3. Ước số chung lớn nhất của hai số

Ước số chung lớn nhất (USCLN) của 2 số được tính theo thuật toán Euclid

$$USCLN(a, b) = USCLN(b, (a \bmod b))$$

```
function USCLN(a,b:longint):longint;
var tmp :longint;
begin
  while b>0 do begin
    a:=a mod b;
    tmp:=a; a:=b; b:=tmp;
  end;
  exit(a);
end;
```

3.4. Bội số chung nhỏ nhất của hai số

Bội số chung nhỏ nhất (BSCNN) của hai số được tính theo công thức:

$$BSCNN(a, b) = \frac{a \times b}{USCLN(a, b)} = \frac{a}{USCLN(a, b)} \times b$$

4. Lí thuyết tập hợp

4.1. Các phép toán trên tập hợp

1. Phần bù của A trong X , kí hiệu \bar{A} , là tập hợp các phần tử của X không thuộc A :

$$\bar{A} = \{x \in X: x \notin A\}$$

2. Hợp của A và B , kí hiệu $A \cup B$, là tập hợp các phần tử hoặc thuộc vào A hoặc thuộc vào B :

$$A \cup B = \{x: x \in A \text{ hoặc } x \in B\}$$

3. Giao của A và B , kí hiệu $A \cap B$, là tập hợp các phần tử đồng thời thuộc cả A và B

$$A \cap B = \{x: x \in A \text{ và } x \in B\}$$

4. Hiệu của A và B , kí hiệu là $A \setminus B$, là tập hợp các phần tử thuộc tập A nhưng không thuộc B .

$$A \setminus B = \{x: x \in A \text{ và } x \notin B\}$$

4.2. Các tính chất của phép toán trên tập hợp

1. Kết hợp

$$(A \cup B) \cup C = A \cup (B \cup C)$$

$$(A \cap B) \cap C = A \cap (B \cap C)$$

2. Giao hoán

$$A \cup B = B \cup A$$

$$A \cap B = B \cap A$$

3. Phân bố

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

4. Đối ngẫu

$$\overline{A \cup B} = \bar{A} \cap \bar{B}$$

$$\overline{A \cap B} = \bar{A} \cup \bar{B}$$

4.3. Tích Đề-các của các tập hợp

Tích Đề-các ghép hai tập hợp:

$$A \times B = \{(a, b) | a \in A, b \in B\}$$

Tích Đề-các mở rộng ghép nhiều tập hợp:

$$A_1 \times A_2 \times \dots \times A_k = \{(a_1, a_2, \dots, a_k) | a_i \in A_i, i = 1, 2, \dots, k\}$$

4.4. Nguyên lí cộng

Nếu A và B là hai tập hợp rời nhau thì

$$|A \cup B| = |A| + |B|$$

Nguyên lí cộng mở rộng cho nhiều tập hợp đôi một rời nhau:

Nếu $\{A_1, A_2, \dots, A_k\}$ là một phân hoạch của tập X thì:

$$|X| = |A_1| + |A_2| + \dots + |A_k|$$

4.5. Nguyên bù trừ

Nếu A và B không rời nhau thì

$$|A \cup B| = |A| + |B| - |A \cap B|$$

Nguyên lí mở rộng cho nhiều tập hợp:

Giả sử A_1, A_2, \dots, A_m là các tập hữu hạn:

$$|A_1 \cup A_2 \cup \dots \cup A_m| = N_1 - N_2 + \dots + (-1)^{m-1} N_m$$

trong đó N_k là tổng phần tử của tất cả các giao của k tập lấy từ m tập đã cho

4.6. Nguyên lí nhân

Nếu mỗi thành phần a_i của bộ có thứ tự k thành phần (a_1, a_2, \dots, a_k) có n_i khả năng lựa chọn ($i = 1, 2, \dots, k$), thì số bộ sẽ được tạo ra là tích số của các khả năng này $n_1 \times n_2 \times \dots \times n_k$

Một hệ quả trực tiếp của nguyên lí nhân:

$$|A_1 \times A_2 \times \dots \times A_k| = |A_1| \times |A_2| \times \dots \times |A_k|$$

4.7. Chinh hợp lặp

Xét tập hữu hạn gồm n phần tử $A = \{a_1, a_2, \dots, a_n\}$

Một chỉnh hợp lặp chập k của n phần tử là một bộ có thứ tự gồm k phần tử của A , các phần tử có thể lặp lại. Một chỉnh hợp lặp chập k của n có thể xem như một phần tử của tích Đềcac A^k . Theo nguyên lí nhân, số tất cả các chỉnh hợp lặp chập k của n sẽ là n^k .

$$\bar{A}_n^k = n^k$$

4.8. Chinh hợp không lặp

Một chỉnh hợp không lặp chập k của n phần tử ($k \leq n$) là một bộ có thứ tự gồm k thành phần lấy từ n phần tử của tập đã cho. Các thành phần không được lặp lại.

Để xây dựng một chỉnh hợp không lặp, ta xây dựng dần từng thành phần đầu tiên. Thành phần này có n khả năng lựa chọn. Mỗi thành phần tiếp theo, số khả năng

lựa chọn giảm đi 1 so với thành phần đứng trước, do đó, theo nguyên lí nhân, số chỉnh hợp không lặp chập k của n sẽ là $n(n-1) \dots (n-k+1)$.

$$A_n^k = n(n-1) \dots (n-k+1) = \frac{n!}{(n-k)!}$$

4.9. Hoán vị

Một hoán vị của n phần tử là một cách xếp thứ tự các phần tử đó. Một hoán vị của n phần tử được xem như một trường hợp riêng của chỉnh hợp không lặp khi $k = n$. Do đó số hoán vị của n phần tử là $n!$

4.10. Tổ hợp

Một tổ hợp chập k của n phần tử ($k \leq n$) là một bộ không kể thứ tự gồm k thành phần khác nhau lấy từ n phần tử của tập đã cho.

$$C_n^k = \frac{n(n-1) \dots (n-k+1)}{k!} = \frac{n!}{k!(n-k)!}$$

Một số tính chất

- $C_n^k = C_n^{n-k}$
- $C_n^0 = C_n^n = 1$
- $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$ (với $0 < k < n$)

5. Số Fibonacci

Số Fibonacci được xác định bởi công thức sau:

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \text{ với } n \geq 2 \end{cases}$$

Một số phần tử đầu tiên của dãy số Fibonacci:

n	0	1	2	3	4	5	6	...
<i>Fibonacci_n</i>	0	1	1	2	3	5	8	...

Số Fibonacci là đáp án của các bài toán:

a) Bài toán cổ về việc sinh sản của các cặp thỏ như sau:

- Các con thỏ không bao giờ chết;

- Hai tháng sau khi ra đời, mỗi cặp thỏ mới sẽ sinh ra một cặp thỏ con (một đực, một cái);
- Khi đã sinh con rồi thì cứ mỗi tháng tiếp theo chúng lại sinh được một cặp con mới.

Giả sử từ đầu tháng 1 có một cặp mới ra đời thì đến giữa tháng thứ n sẽ có bao nhiêu cặp.

Ví dụ, $n = 5$, ta thấy:

Giữa tháng thứ 1:

1 cặp (cặp ban đầu)

Giữa tháng thứ 2:

1 cặp (cặp ban đầu vẫn chưa đẻ)

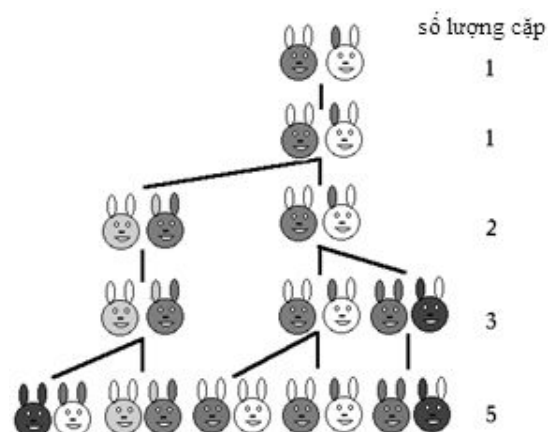
Giữa tháng thứ 3:

2 cặp (cặp ban đầu đẻ ra thêm 1 cặp con)

Giữa tháng thứ 4:

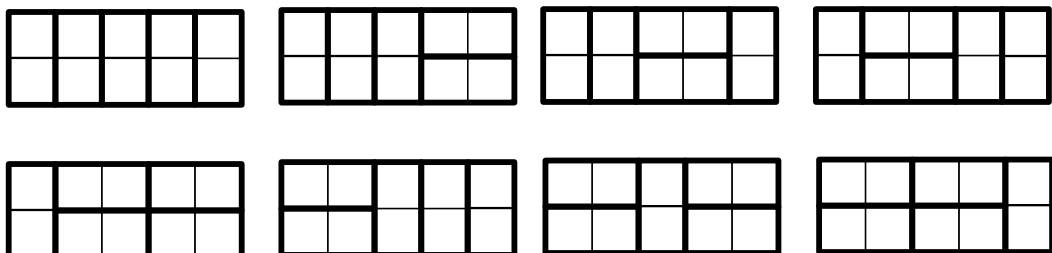
3 cặp (cặp ban đầu tiếp tục đẻ)

Giữa tháng thứ 5: 5 cặp.



b) Đếm số cách xếp $n - 1$ thanh DOMINO có kích thước 2×1 phủ kín bảng có kích thước $2 \times (n - 1)$.

Ví dụ: Có tất cả 8 cách khác nhau để xếp các thanh DOMINO có kích thước 2×1 phủ kín bảng 2×5 ($n = 6, \text{Fibonacci}_6 = 8$).



Hàm tính số Fibonacci thứ n bằng phương pháp lặp sử dụng công thức

$$F_n = F_{n-1} + F_{n-2} \text{ với } n \geq 2 \text{ và } F_0 = 0, F_1 = 1.$$

```
function Fibo(n : longint):longint;
var fi_1, fi_2, fi, i :longint;
begin
```

```

if n<=1 then exit(n);
fi_2:=0; fi_1:=1;
for i:=2 to n do begin
    fi:=fi_1 + fi_2;
    fi_2:=fi_1;
    fi_1:=fi;
end;
exit(fi);
end;

```

Công thức tổng quát $F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$

6. Số Catalan

Số Catalan được xác định bởi công thức sau:

$$Catalan_n = \frac{1}{n+1} C_{2n}^n = \frac{(2n)!}{(n+1)!n!} \text{ với } n \geq 0$$

Một số phân tử đầu tiên của dãy số Catalan là:

n	0	1	2	3	4	5	6	...
$Catalan_n$	1	1	2	5	14	42	132	...

Số Catalan là đáp án của các bài toán:

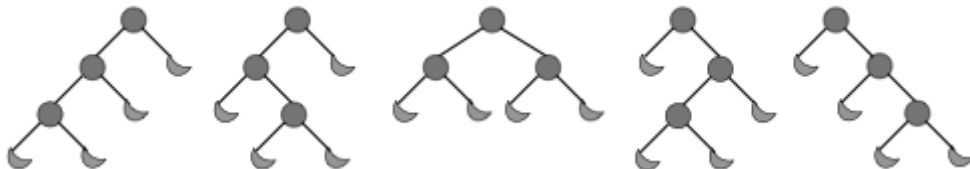
1) Có bao nhiêu cách khác nhau đặt n dấu ngoặc mở và n dấu ngoặc đóng đúng đắn?

Ví dụ: $n = 3$ ta có 5 cách sau:

$((())), (() ()), (()) (), () (()), () () ()$

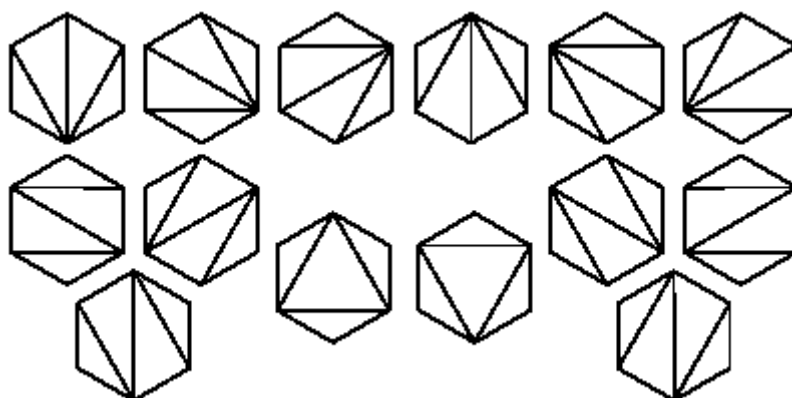
2) Có bao nhiêu cây nhị phân khác nhau có đúng $(n + 1)$ lá?

Ví dụ: $n = 3$



3) Cho một đa giác lồi $(n + 2)$ đỉnh, ta chia đa giác thành các tam giác bằng cách vẽ các đường chéo không cắt nhau trong đa giác. Hỏi có bao nhiêu cách chia như vậy?

Ví dụ: $n = 4$



7. Xử lý số nguyên lớn

Nhiều ngôn ngữ lập trình cung cấp kiểu dữ liệu nguyên khá lớn, chẳng hạn trong Free Pascal có kiểu số 64 bit (khoảng 19 chữ số). Tuy nhiên để thực hiện các phép tính với số nguyên ngoài phạm vi biểu diễn được cung cấp (có hàng trăm chữ số chẳng hạn), chúng ta cần tự thiết kế cách biểu diễn và các hàm thực hiện các phép toán cơ bản với các số nguyên lớn.

7.1. Biểu diễn số nguyên lớn

Thông thường người ta sử dụng các cách biểu diễn số nguyên lớn sau:

- **Xâu ký tự:** Đây là cách biểu diễn tự nhiên và đơn giản nhất, mỗi ký tự của chuỗi tương ứng với một chữ số của số nguyên lớn tính từ trái qua phải.
- **Mảng các số:** Sử dụng mảng lưu các chữ số (hoặc một nhóm chữ số), và một biến ghi nhận số chữ số để thuận tiện trong quá trình xử lý.
- **Danh sách liên kết các số:** Sử dụng danh sách liên kết các chữ số (hoặc một nhóm chữ số), cách làm này sẽ linh hoạt hơn trong việc sử dụng bộ nhớ.

Trong phần này, sử dụng cách biểu diễn thứ nhất, biểu diễn số nguyên lớn bằng chuỗi ký tự và chỉ xét các số nguyên lớn không âm.

```
Type bigNum = string;
```


7.2. Phép so sánh

Để so sánh hai số nguyên lớn a , b được biểu diễn bằng xâu kí tự, trước tiên ta thêm các chữ số 0 vào đầu số có số chữ số nhỏ hơn để hai số có số lượng chữ số bằng nhau. Sau đó sử dụng trực tiếp phép toán so sánh trên xâu kí tự.

Hàm *cmp* so sánh hai số nguyên lớn a , b . Giá trị hàm trả về

$$\begin{cases} 0 & \text{nếu } a = b \\ 1 & \text{nếu } a > b \\ -1 & \text{nếu } a < b \end{cases}$$

```
function      cmp(a,b : bigNum): integer;
begin
    while length(a)<length(b) do a:='0'+a;
    while length(b)<length(a) do b:='0'+b;
    if a = b then exit(0);
    if a > b then exit(1);
    exit(-1);
end;
```

7.3. Phép cộng

Phép cộng hai số nguyên được thực hiện từ phải qua trái và phần nhớ được mang sang trái.

```
function      add(a,b : bigNum): bigNum;
var  sum, carry, i, x, y : integer;
    c      : bigNum;
begin
    carry:=0;c:='';
    while length(a)<length(b) do a:='0'+a;
    while length(b)<length(a) do b:='0'+b;
    for i:=length(a) downto 1 do
        begin
            x:= ord(a[i])-ord('0'); {ord('0')=48}
            y:= ord(b[i])-ord('0');

            sum:=x + y + carry;
            carry:=sum div 10;
```

```

        c:=chr(sum mod 10 +48)+c;
    end;
    if carry>0 then c:='1'+c;
    add:=c;
end;

```

7.4. Phép trừ

Thực hiện phép trừ ngược lại với việc nhớ ở phép cộng ta phải chú ý đến việc vay mượn từ hàng cao hơn. Trong hàm trừ dưới đây, chỉ xét trường hợp số lớn trừ số nhỏ hơn.

```

function  sub(a,b:bigNum):bigNum;
var      c                :bigNum;
s,borrow,i  :integer;
begin
    borrow:=0;c:='';
    while length(a)<length(b) do a:='0'+a;
    while length(b)<length(a) do b:='0'+b;
    for i:=length(a) downto 1 do
        begin
            s:=ord(a[i])-ord(b[i])-borrow;
            if s<0 then
                begin
                    s:=s+10;
                    borrow:=1;
                end else borrow:=0;
            c:=chr(s +48)+c;
        end;
    while (length(c)>1)and(c[1]='0') do delete(c,1,1);
    sub:=c;
end;

```

7.5. Phép nhân một số lớn với một số nhỏ

Số nhỏ ở đây được hiểu là số nguyên do ngôn ngữ lập trình cung cấp (như: longint, integer,...). Hàm multiply1(a:bigNum;b:longint):bigNum, trả về là một số nguyên lớn (bigNum) là kết quả của phép nhân một số nguyên lớn a (bigNum) với một số b (longint).

```

function      multiply1(a:bigNum;b:longint):bigNum;
var i          :integer;
    carry,s    :longint;
    c,tmp      :bigNum;
begin
    c:='';
    carry:=0;
    for i:=length(a) downto 1 do
        begin
            s:=(ord(a[i])-48) * b + carry;
            carry:= s div 10;
            c:=chr(s mod 10 + 48)+c;
        end;
    if carry>0 then str(carry,tmp) else tmp:='';
    multiply1:=tmp+c;
end;

```

7.6. Phép nhân hai số nguyên lớn

```

function      multiply2(a,b:bigNum):bigNum;
var  sum,tmp  :bigNum;
    m,i,j     :integer;
begin
    m:=-1;sum:='';
    for i:=length(a) downto 1 do
        begin
            m:=m+1;
            tmp:=multiply1(b,ord(a[i])-48);
            {có thể thay câu lệnh tmp:=multiply1(b,ord(a[i])-48);
             bằng cách cộng nhiều lần như sau:
            tmp:="";
            for j:=1 to ord(a[i])-48 do tmp:=add(tmp,b);
            như vậy hàm nhân multiply2 chỉ gọi hàm cộng hai số nguyên lớn add}
            for j:=1 to m do tmp:=tmp+'0';
            sum:=add(tmp,sum);
        end;
    multiply2:=sum;
end;

```

7.7. Phép toán chia lấy thương nguyên (div) của một số lớn với một số nhỏ

```
function bigDiv1(a:bigNum;b:longint):bigNum;
var s,i,hold:longint;
c:bigNum;
begin
    hold:=0;s:=0; c:='';
    for i:=1 to length(a) do
        begin
            hold:=hold*10 + ord(a[i])-48;
            s:=hold div b;
            hold:=hold mod b;
            c:=c+chr(s+48);
        end;
    while (length(c)>1) and(c[1]='0') do
        delete(c,1,1);
        bigDiv1:=c;
    end;
```

7.8. Phép toán chia lấy dư (mod) của một số lớn với một số nhỏ

```
function bigMod1(a:bigNum;b:longint):longint;
var i,hold:longint;
begin
    hold:=0;
    for i:=1 to length(a) do
        hold:=(ord(a[i])-48+hold*10) mod b;
    bigMod1:=hold;
end;
```

Chú ý: Ta có các công thức sau:

- 1) $(A + B) \bmod N = ((A \bmod N) + (B \bmod N)) \bmod N$
- 2) $(A \times B) \bmod N = ((A \bmod N) \times (B \bmod N)) \bmod N$

7.9. Phép toán chia lấy thương nguyên (div) của hai số lớn

```
function bigDiv2(a,b:bigNum):bigNum;
var c,hold :bigNum;
```

```

        kb      :array[0..10]of bigNum;
        i,k      :longint;
begin
    kb[0]:='0';
    for i:=1 to 10 do
        kb[i]:=add(kb[i-1],b);
    hold:='';
    c:='';
    for i:=1 to length(a) do
        begin
            hold:=hold+a[i];
            k:=1;
            while cmp(hold,kb[k])<>-1 do
                inc(k);
            c:=c+chr(k-1+48);
            hold:=sub(hold,kb[k-1]);
        end;
        while (length(c)>1)and(c[1]='0') do delete(c,1,1);
        bigDiv2:=c;
    end;
end;

```

7.10. Phép toán chia lấy dư (mod) của hai số lớn

```

function bigMod2(a,b:bigNum):bigNum;
var  hold      :bigNum;
     kb        :array[0..10]of bigNum;
     i,k       :longint;
begin
    kb[0]:='0';
    for i:=1 to 10 do
        kb[i]:=add(kb[i-1],b);
    hold:='';
    for i:=1 to length(a) do
        begin
            hold:=hold+a[i];
            k:=1;
            while cmp(hold,kb[k])<>-1 do

```

```

        inc(k);
        hold:=sub(hold, kb[k-1]);
    end;
    bigMod2:=hold;
end;

```

7.11. Ví dụ tính số Fibonacci thứ n ($n \leq 500$)

Số Fibonacci được xác định bởi công thức sau:

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \text{ với } n \geq 2 \end{cases}$$

Trước tiên ta xây dựng chương trình tính số Fibonacci bằng kiểu dữ liệu Extended như sau:

```

function Fibo(n : longint):extended;
var i :longint;
fi_1, fi_2, fi : extended;
begin
    if n<=1 then exit(n);
    fi_2:=0; fi_1:=1;
    for i:=2 to n do begin
        fi:=fi_1 + fi_2;
        fi_2:=fi_1;
        fi_1:=fi;
    end;
    exit(fi);
end;
var n : longint;
BEGIN
write('Nhap N:'); readln(n);
writeln(Fibo(n));
END.

```

Chạy chương trình với $n = 500$ ta nhận được kết quả:

1.3942322456169788E+0104, như vậy số Fibonacci thứ 500 có 105 chữ số (có thể sử dụng cách biểu diễn bằng xâu kí tự), ta xây dựng chương trình tính số Fibonacci lớn bằng cách sau:

- Thay kiểu extended bằng kiểu bigNum.
- Thay các phép toán bằng các hàm tính toán số lớn, xây dựng các hàm tính toán số lớn cần thiết.

```

type bigNum = string;
function      add(a,b : bigNum): bigNum;
var  sum, carry, i : integer;
      c              : bigNum;
begin
    carry:=0;c:='';
    while length(a)<length(b) do a:='0'+a;
    while length(b)<length(a) do b:='0'+b;
    for i:=length(a) downto 1 do
        begin
            sum:=ord(a[i])-48+ord(b[i])-48+carry;
            carry:=sum div 10;
            c:=chr(sum mod 10 +48)+c;
        end;
    if carry>0 then c:='1'+c;
    add:=c;
end;
function Fibo(n : longint):bigNum;
var i :longint;
fi_1, fi_2, fi : bigNum;
begin
    if n<=1 then exit(char(n+48));
    fi_2:='0'; fi_1:='1';
    for i:=2 to n do begin
        fi:=add(fi_1,fi_2); {fi:=fi_1 + fi_2;}
        fi_2:=fi_1;
        fi_1:=fi;
    end;
    exit(fi);
end;
var n : longint;
BEGIN

```

```

write('Nhap N:'); readln(n);
writeln(Fibo(n));
END.

```

7.12. Ví dụ tính số $Catalan_n$ ($n \leq 100$)

Số Catalan được xác định bởi công thức sau:

$$Catalan_n = \frac{1}{n+1} C_{2n}^n = \frac{(2n)!}{(n+1)!n!} \text{ với } n \geq 0$$

Rút gọn tử và mẫu cho $(n+1)!$ ta có:

$$Catalan_n = \frac{(n+2) \times (n+3) \times \dots \times (2n)}{1 \times 2 \times \dots \times n}$$

Ta sẽ tính tử số bằng cách sử dụng hàm nhân số lớn với số nhỏ, sau đó sử dụng hàm chia số lớn cho số nhỏ để được kết quả cần tính.

```

type bigNum      =string;
function         multiply1(a:bigNum;b:longint):bigNum;
var i            :integer;
    carry,s      :longint;
    c,tmp        :bigNum;
begin
    c:='';
    carry:=0;
    for i:=length(a) downto 1 do
        begin
            s:=(ord(a[i])-48) * b + carry;
            carry:= s div 10;
            c:=chr(s mod 10 + 48)+c;
        end;
    if carry>0 then str(carry,tmp) else tmp:='';
    multiply1:=tmp+c;
end;
function  bigDiv1(a:bigNum;b:longint):bigNum;
var s,i,hold:longint;
c:bigNum;
begin
    hold:=0;s:=0; c:='';

```



```

        for i:=1 to length(a) do
            begin
                hold:=hold*10 + ord(a[i])-48;
                s:=hold div b;
                hold:=hold mod b;
                c:=c+chr(s+48);
            end;
        while (length(c)>1) and(c[1]='0') do
delete(c,1,1);
        bigDiv1:=c;
        end;
var n,k    :longint;
    s      :bigNum;
BEGIN
    write('Nhap N:'); readln(n);
    s:='1';
    for k:=(n+2) to 2*n do s:=multiply1(s,k); {tính tử số}
    for k:=1 to n do s:=bigDiv(s,k); {chia cho mẫu số}
    writeln(s);
END.

```

Tuy nhiên, ta có thể rút gọn hoàn toàn mẫu số của phân số trên và chỉ cần sử dụng hàm nhân số lớn với số nhỏ, chương trình sẽ chạy nhanh hơn.

Bài tập

2.1. Cho s là một chuỗi chỉ gồm 2 ký tự '0' hoặc '1' mô tả một số nguyên không âm ở hệ cơ số 2, hãy chuyển số đó sang hệ cơ số 16 (độ dài chuỗi s không vượt quá 200).

Ví dụ: $10101100_2 = AC_{16}$

$101010111100000100100011_2 = ABC123_{16}$

2.2. Cho số nguyên dương N ($N \leq 10^9$)

a) Phân tích N thành thừa số nguyên tố

b) Đếm số ước của N

c) Tính tổng các ước của N

2.3. Đưa ra những số $\leq 10^6$ mà cách kiểm tra tính nguyên tố của Fermat bị sai.

2.4. Sử dụng sàng số nguyên tố liệt kê các số nguyên tố trong đoạn $[L, R]$

2.5. Người ta định nghĩa một số nguyên dương N được gọi là số đẹp nếu N thỏa mãn *một trong hai* điều kiện sau:

- N bằng 9
- Gọi $f(N)$ là tổng các chữ số của N thì $f(N)$ cũng là số đẹp

Cho số nguyên dương N ($N \leq 10^{100}$), hãy kiểm tra xem N có phải là số đẹp không?

2.6. Dùng cách biểu diễn số nguyên lớn bằng xâu và thêm thông tin dấu ($\text{sign}=1$ nếu số lớn là số không âm, $\text{sign}=-1$ nếu số lớn là số âm) để xử lý số nguyên lớn có dấu như sau:

```
type    bigNum    = record
                                sign : longint;
                                num  : string;
                                end;
```

Hãy xây dựng các hàm xử lý số nguyên lớn có dấu.

2.7. Dùng cách biểu diễn số nguyên lớn bằng mảng (mỗi phần tử của mảng là một nhóm các chữ số).

a) Hãy xây dựng các hàm xử lý số nguyên lớn.

b) Sử dụng hàm nhân số nguyên lớn với số nhỏ tính $N!$ với $N \leq 2000$.

2.8. Tìm K chữ số cuối cùng của M^N ($0 < K \leq 9, 0 \leq M, N \leq 10^6$)

Ví dụ: $K=2, M=2, N=10$, ta có $2^{10}=1024$, như vậy 2 chữ số cuối cùng của 2^{10} là 24

2.9. Cho N ($N \leq 10$) nguyên dương a_1, a_2, \dots, a_N ($a_i < 10^9$). Tìm ước số chung lớn nhất, bội số chung nhỏ nhất của N số trên (chú ý: BSCNN có thể rất lớn).

2.10. Cho hai số nguyên không âm A, B ($0 \leq A \leq B \leq 10^{200}$), tính số lượng số Fibonacci trong đoạn $[A, B]$.

2.11. Cho số nguyên dương N ($N \leq 10^{100}$), hãy tách N thành tổng các số Fibonacci đôi một khác nhau.

Ví dụ: $N=16=1+5+10$

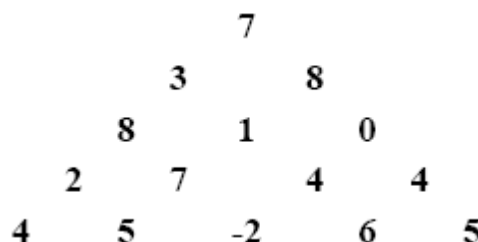
2.12. Cho N là một số nguyên dương không vượt quá 10^9 . Hãy tìm số chữ số 0 tận cùng của $N!$

- 2.13.** Cho s là một xâu mô tả số nguyên không âm ở hệ cơ số a , hãy chuyển số đó sang hệ cơ số b ($1 < a, b \leq 16$, độ dài xâu s không vượt quá 50).
- 2.14.** Xây dựng hàm kiểm tra số nguyên dương N có phải là số chính phương không? ($N < 10^{100}$)
- 2.15.** Tính C_n^k ($0 < k \leq n \leq 2000$)
- 2.16.** Tính Catalan $_n$ ($n \leq 2000$)
- 2.17.** Hãy đếm số cách đặt k quân xe lên bàn cờ $n \times n$ sao cho không có quân nào ăn được nhau. ($1 \leq k \leq n \leq 100$)
- 2.18.** Giả thiết N là số nguyên dương. Số nguyên M là tổng của N với các chữ số của nó. N được gọi là nguồn của M . Ví dụ, $N = 245$, khi đó $M = 245 + 2 + 4 + 5 = 256$. Như vậy, nguồn của 256 là 245. Có những số không có nguồn và có số lại có nhiều nguồn. Ví dụ, số 216 có 2 nguồn là 198 và 207.
Cho số nguyên M (M có không quá 100 chữ số) hãy tìm nguồn nhỏ nhất của nó. Nếu M không có nguồn thì đưa ra số 0.
- 2.19.** Tính số ước và tổng các ước của $N!$ ($N \leq 100$)
- 2.20.** Cho một chiếc cân hai đĩa và các quả cân có khối lượng $3^0, 3^1, 3^2, \dots$
Hãy chọn các quả cân để có thể cân được vật có khối lượng N ($N \leq 10^{100}$)
Ví dụ: cân cân vật có khối lượng $N=11$ ta cần sử dụng các quả cân sau:
- Cân bên trái: quả cân 3^1 và 3^2
- Cân bên phải: quả cân 3^0 và vật $N=11$
- 2.21.** Đếm số lượng dãy nhị phân khác nhau độ dài n mà không có 2 số 1 nào đứng cạnh nhau?
Ví dụ: $n = 3$, ta có 5 dãy 000, 001, 010, 100, 101
- 2.22.** Cho xâu s chỉ gồm kí tự từ 'a' đến 'z' (độ dài xâu s không vượt quá 100), hãy đếm số hoán vị khác nhau của xâu đó.
Ví dụ: $s='aba'$, ta có 3 hoán vị 'aab','aba','baa'
- 2.23.** John Smith quyết định đánh số trang cho quyển sách của anh ta từ 1 đến N . Hãy tính toán số lượng chữ số 0 cần dùng, số lượng chữ số 1 cần dùng, ..., số lượng chữ số 9 cần dùng.
Dữ liệu vào trong file: "digits.inp" gồm 1 dòng duy nhất chứa một số N ($N \leq 10^{100}$).

Kết quả ra file “digits.out” có dạng gồm 10 dòng, dòng thứ nhất là số lượng chữ số 0 cần dùng, dòng thứ hai là số lượng chữ số 1 cần dùng,..., dòng thứ 10 là số lượng chữ số 9 cần dùng.

2.24. TAM GIÁC SỐ (đề thi học sinh giỏi Hà Tây 2006)

Hình bên mô tả một tam giác số có số hàng $N=5$. Đi từ đỉnh (số 7) đến đáy tam giác bằng một đường gấp khúc, mỗi bước chỉ được đi từ số ở hàng trên xuống một trong hai số đứng kề bên phải hay bên trái ở hàng dưới, và tính tích các số trên đường đi lại ta được một tích.



Ví dụ: đường đi 7 8 1 4 6 có tích là $S=1344$, đường đi 7 3 1 7 5 có tích là $S=735$.

Yêu cầu: Cho tam giác số, tìm tích của đường đi có tích lớn nhất

Dữ liệu: Vào từ file văn bản TGS.INP:

- Dòng đầu tiên chứa số nguyên n , ($0 < n < 101$)
- N dòng tiếp theo, từ dòng thứ 2 đến dòng thứ $N+1$: dòng thứ i có $(i-1)$ số cách nhau bởi dấu cách (các số có giá trị tuyệt đối không vượt quá 100)

Kết quả: Đưa ra file văn bản TGS.OUT một số nguyên – là tích lớn nhất tìm được

TGS . INP	TGS . OUT
5	5880
7	
3 8	
8 1 0	
2 7 4 4	
4 5 -2 6 5	

2.25. HÁI NẤM (bài thi Olympic Sinh viên 2009, khối chuyên)

Một cháu gái hàng ngày được mẹ giao nhiệm vụ đến thăm bà nội. Từ nhà mình đến nhà bà nội cô bé phải đi qua một khu rừng có rất nhiều loại nấm. Trong số các loại nấm, có ba loại có thể ăn được. Cô bé đánh số ba loại nấm ăn được lần lượt là 1, 2 và 3. Là một người cháu hiếu thảo cho nên cô bé

quyết định mỗi lần đến thăm bà, cô sẽ hái ít nhất hai loại nấm ăn được để nấu súp cho bà. Khu rừng mà cô bé đi qua được chia thành lưới ô vuông gồm m hàng và n cột. Các hàng của lưới được đánh số từ trên xuống dưới bắt đầu từ 1, còn các cột – đánh số từ trái sang phải, bắt đầu từ 1. Ô nằm giao của hàng i và cột j có tọa độ (i, j) . Trên mỗi ô vuông, trừ ô $(1,1)$ và ô (m, n) các ô còn lại hoặc có nấm độc và cô bé không dám đi vào (đánh dấu là -1), hoặc là có đúng một loại nấm có thể ăn được (đánh dấu bằng số hiệu của loại nấm đó). Khi cô bé đi vào một ô vuông có nấm ăn được thì cô bé sẽ hái loại nấm mọc trên ô đó. Xuất phát từ ô $(1,1)$, để đến được nhà bà nội ở ô (m, n) một cách nhanh nhất cô bé luôn đi theo hướng sang phải hoặc xuống dưới.

Việc đi thăm bà và hái nấm trong rừng sâu gặp nguy hiểm bởi có một con chó sói luôn theo dõi và muốn ăn thịt cô bé. Để phòng tránh chó sói theo dõi và ăn thịt, cô bé quyết định mỗi ngày sẽ đi theo một con đường khác nhau (hai con đường khác nhau nếu chúng khác nhau ở ít nhất một ô).

Yêu cầu: Cho bảng $m \times n$ ô vuông mô tả trạng thái khu rừng. Hãy tính số con đường khác nhau để cô bé đến thăm bà nội theo cách chọn đường đi đã nêu ở trên.

Dữ liệu: Vào từ file văn bản MUSHROOM.INP:

- Dòng đầu chứa 2 số m, n ($1 < m, n < 101$),
- m dòng tiếp theo, mỗi dòng chứa n số nguyên cho biết thông tin về các ô của khu rừng. (riêng giá trị ở hai ô $(1,1)$ và ô (m, n) luôn luôn bằng 0 các ô còn lại có giá trị bằng -1, hoặc 1, hoặc 2, hoặc 3).

Hai số liên tiếp trên một dòng cách nhau một dấu cách.

Kết quả: Đưa ra file văn bản MUSHROOM.OUT chứa một dòng ghi một số nguyên là kết quả bài toán.

Ví dụ:

MUSHROOM.INP	MUSHROOM.OUT
3 4 0 3 -1 2 3 3 3 3 3 1 3 0	3

2.26. HỆ THỐNG ĐÈN MÀU (Tin học trẻ bảng B năm 2009)

Để trang trí cho lễ kỉ niệm 15 năm hội thi Tin học trẻ toàn quốc, ban tổ chức đã dùng một hệ thống đèn màu gồm n đèn đánh số từ 1 đến n . Mỗi đèn có

khả năng sáng màu xanh hoặc màu đỏ. Các đèn được điều khiển theo quy tắc sau:

- Ban đầu tất cả các đèn đều sáng màu xanh.
- Sau khi kết thúc chương trình thứ nhất của lễ kỉ niệm, tất cả các đèn có số thứ tự chia hết cho 2 sẽ đổi màu...Sau khi kết thúc chương trình thứ i , tất cả các đèn có số thứ tự chia hết cho $i + 1$ sẽ đổi màu (đèn xanh đổi thành màu đỏ còn đèn đỏ đổi thành màu xanh)

Minh, một thí sinh dự lễ kỉ niệm đã phát hiện được quy luật điều khiển đèn và rất thích thú với hệ thống đèn trang trí này. Vào lúc chương trình thứ k của buổi lễ vừa kết thúc, Minh đã nhầm tính được tại thời điểm đó có bao nhiêu đèn xanh và bao nhiêu đèn đỏ. Tuy nhiên vì không có máy tính nên Minh không chắc chắn kết quả của mình là đúng. Cho biết hai số n và k ($n, k \leq 10^6$), em hãy tính lại giúp Minh xem khi chương trình thứ k của buổi lễ vừa kết thúc, có bao nhiêu đèn màu đỏ.

Ví dụ với $n = 10; k = 3$.

Thời điểm	Trạng thái các đèn
Bắt đầu	Xanh: 1 2 3 4 5 6 7 8 9 10 Đỏ :
Sau chương trình 1	Xanh: 1 3 5 7 9 Đỏ : 2 4 6 8 10
Sau chương trình 2	Xanh: 1 5 6 7 Đỏ : 2 3 4 8 9 10
Sau chương trình 3	Xanh: 1 4 5 6 7 8 Đỏ : 2 3 9 10

Vậy có 4 đèn đỏ sau chương trình thứ 3.

SẮP XẾP

Sắp xếp là quá trình bố trí lại vị trí các đối tượng của một danh sách theo một trật tự nhất định. Sắp xếp đóng vai trò rất quan trọng trong cuộc sống nói chung và trong tin học nói riêng, thử hình dung xem, một cuốn từ điển, nếu các từ không được sắp xếp theo thứ tự, sẽ khó khăn như thế nào trong việc tra cứu các từ. Theo D.Knuth thì 40% thời gian tính toán của máy tính là dành cho việc sắp xếp. Không phải ngẫu nhiên thuật toán sắp xếp nhanh (Quick Sort) được bình chọn là một trong 10 thuật toán tiêu biểu của thế kỉ 20.

Do đặc điểm dữ liệu (kiểu số hay phi số, kích thước bé hay lớn, lưu trữ ở bộ nhớ trong hay bộ nhớ ngoài, truy cập tuần tự hay ngẫu nhiên...) mà người ta có các thuật toán sắp xếp khác nhau. Trong chuyên đề này, chúng ta chỉ quan tâm đến các thuật toán sắp xếp trong trường hợp dữ liệu được lưu trữ ở bộ nhớ trong (nghĩa là toàn bộ dữ liệu cần sắp xếp phải được đưa vào bộ nhớ chính của máy tính).

1. Phát biểu bài toán

Giả sử các đối tượng cần sắp xếp được biểu diễn bởi bản ghi gồm một số trường. Một trong các trường đó được gọi là *khoá sắp xếp*. Kiểu của khoá là kiểu có thứ tự (chẳng hạn, kiểu số nguyên, kiểu số thực,...)

```
const
MAX      =...;
type
object    = record
            key : keyType;
            [các trường khác]
        end;
TArray    = array[1..MAX] of object;
var
a          : TArray;
n          : longint;
```

Bài toán sắp xếp được phát biểu như sau: Cho mảng a các đối tượng, cần sắp xếp lại các thành phần (phần tử) của mảng a để nhận được mảng a mới với các thành phần có các giá trị khoá tăng dần:

$$a[1].key \leq a[2].key \leq \dots \leq a[n].key$$

2. Các thuật toán sắp xếp thông dụng

Hai thuật toán hay được sử dụng nhiều trong thực tế đó là thuật toán sắp xếp nổi bọt (BUBBLE SORT) và thuật toán sắp xếp nhanh (QUICK SORT).

2.1 Thuật toán sắp xếp nổi bọt (Bubble Sort)

Ý tưởng cơ bản của thuật toán là tìm và đổi chỗ các cặp phần tử kề nhau sai thứ tự (phần tử đứng trước có khoá lớn hơn khoá của phần tử đứng sau) cho đến khi không tồn tại cặp nào sai thứ tự (dãy được sắp xếp).

Cụ thể:

- Lượt 1: ta xét từ cuối dãy, nếu gặp 2 phần tử kề nhau mà sai thứ tự thì đổi chỗ chúng cho nhau. Sau lượt 1, phần tử có khoá nhỏ thứ nhất được đưa về vị trí 1.
- Lượt 2: ta xét từ cuối dãy (chỉ đến phần tử thứ 2), nếu gặp 2 phần tử kề nhau mà sai thứ tự thì đổi chỗ chúng cho nhau. Sau lượt 2, phần tử có khoá nhỏ thứ hai được đưa về vị trí 2.
- Lượt i : ta xét từ cuối dãy về (chỉ đến phần tử thứ i , vì phần đầu dãy từ 1 đến $i-1$ đã được xếp đúng thứ tự), nếu gặp 2 phần tử kề nhau mà sai thứ tự thì đổi chỗ chúng cho nhau. Sau lượt i , phần tử có khoá nhỏ thứ i được đưa về vị trí i .

Xong lượt thứ $n-1$ thì dãy được sắp xếp xong.

```
procedure BoubbleSort;
var i, j : integer;
    tmp : object;
begin
  for i := 1 to n-1 do
    for j := n downto i+1 do
      if a[j-1].key > a[j].key then
begin
      tmp := a[j];
```



```

        a[j] := a[j-1];
        a[j-1] := tmp;
    end;
end;

```

Đánh giá độ phức tạp

Số phép toán so sánh $a[j-1].key > a[j].key$ được dùng để đánh giá hiệu suất thuật toán về mặt thời gian cho thuật toán sắp xếp nổi bọt. Tại lượt thứ i ta cần $n-i$ phép so sánh. Như vậy tổng số phép so sánh cần thiết:

$$(n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2}$$

Thuật toán có độ phức tạp $O(N^2)$

Một thuật toán sắp xếp đơn giản, hay sử dụng khác cũng cho độ phức tạp $O(N^2)$

```

for i:=1 to n-1 do
  for j:=i+1 to n do
    if a[i].key>a[j].key then begin
      tmp:=a[i];
      a[i]:=a[j];
      a[j]:=tmp;
    end;
  end;
end;

```

2.2. Thuật toán sắp xếp nhanh (Quick Sort)

Ý tưởng của thuật toán như sau: Để sắp xếp dãy coi như là sắp xếp đoạn từ chỉ số 1 đến chỉ số n . Để sắp xếp một đoạn trong dãy, nếu đoạn chỉ có một phần tử thì dãy đã được sắp xếp, ngược lại ta chọn một phần tử x trong đoạn đó làm "chốt", mọi phần tử có khoá nhỏ hơn khoá của "chốt" được xếp vào vị trí đứng trước chốt, mọi phần tử có khoá lớn hơn khoá của "chốt" được xếp vào vị trí đứng sau chốt. Sau phép hoán chuyển như vậy thì đoạn đang xét được chia làm hai đoạn mà mọi phần tử trong đoạn đầu đều có khoá \leq khoá của "chốt" và mọi phần tử trong đoạn sau đều có khoá \geq khoá của "chốt". Tiếp tục sắp xếp kiểu như vậy với 2 đoạn con, ta sẽ được đoạn đã cho được sắp xếp theo chiều tăng dần của khoá.

Cụ thể:

Giả sử phải sắp xếp đoạn có chỉ số từ L đến H :

- chọn x là một phần tử ngẫu nhiên trong đoạn $L..H$ (có thể chọn x là phần tử ở giữa đoạn, nghĩa là $x = a[(L+H) \text{ div } 2]$)

- cho i chạy từ L sang phải, j chạy từ H sang trái; nếu phát hiện một cặp ngược thứ tự: $i \leq j$ và $a[i].key \geq x.key \geq a[j].key$ thì đổi chỗ 2 phần tử đó; cho đến khi $i > j$. Lúc đó dãy ở tình trạng: khoá các phần tử đoạn $L..i \leq$ khoá của x ; khoá của các phần tử đoạn $j..H \geq$ khoá của x . Tiếp tục sắp xếp như vậy với 2 đoạn $L..j$ và $i..H$.

Thủ tục QuickSort(L, H) sau, sắp xếp đoạn từ L tới H , để sắp xếp dãy số ta gọi QuickSort($1, n$)

```
procedure QuickSort(L,H:longint);
var i,j      :longint;
x,tmp       :object;
begin
    i:=L;
    j:=H;
    //x:=a[random(H-L+1)+L];
    x:=a[(L+H) div 2];
    repeat
        while a[i].key<x.key do inc(i);
        while a[j].key>x.key do dec(j);
        if i<=j then
            begin
                tmp:=a[i];
                a[i]:=a[j];
                a[j]:=tmp;
                inc(i);
                dec(j);
            end;
    until i>j;
    if L<j then QuickSort(L,j);
    if i<H then QuickSort(i,H);
end;
```

Đánh giá độ phức tạp

Việc chọn chốt để phân đoạn quyết định hiệu quả của thuật toán, nếu việc chọn chốt không tốt rất có thể việc phân đoạn bị suy biến thành trường hợp xấu (phân

thành hai đoạn mà số phần tử của hai đoạn chênh lệch nhiều) khiến Quick Sort hoạt động chậm. Các tính toán độ phức tạp chi tiết cho thấy thuật toán Quick sort:

- Có thời gian thực thi cỡ $O(n \log n)$ trong trường hợp trung bình.
- Có thời gian thực thi cỡ $O(n^2)$ trong trường hợp xấu nhất (2 đoạn được chia thành một đoạn $n-1$ và một đoạn 1 phần tử). Khả năng để xảy ra trường hợp này là rất ít, còn nếu chọn chốt ngẫu nhiên, hầu như sẽ không xảy ra.

2.3. Nhận xét

Nếu chương trình ít gọi tới thủ tục sắp xếp và chỉ trên tập dữ liệu nhỏ, thì việc sử dụng một thuật toán phức tạp (tuy có hiệu quả hơn) có thể không cần thiết, khi đó có thể sử dụng thuật toán đơn giản có độ phức tạp $O(N^2)$, dễ cài đặt. Tuy nhiên, vì độ phức tạp $O(n^2)$, nghĩa là thời gian thực hiện tăng lên gấp 4 khi số lượng phần tử tăng lên gấp đôi. Do đó, trong trường hợp sắp xếp trên tập dữ liệu lớn nên sử dụng thuật toán sắp xếp nhanh có độ phức tạp cỡ $O(n \log n)$.

3. Sắp xếp bằng đếm phân phối (Distribution Counting)

Trong trường hợp khoá các phần tử $a[1], a[2], \dots, a[n]$ là các số nguyên nằm trong khoảng từ 0 tới K ta có thuật toán *đơn giản* và *hiệu quả* như sau:

Xây dựng dãy $c[0], c[1], \dots, c[K]$, trong đó $c[V]$ là số lần xuất hiện khoá V trong dãy.

```
for V := 0 to K do c[V] := 0; {Khởi tạo dãy c}
for i := 1 to n do c[a[i].key] := c[a[i].key] + 1;
```

Như vậy, sau khi sắp xếp:

- Các phần tử có khoá bằng 0 đứng trong đoạn từ vị trí 1 tới vị trí $c[0]$.
- Các phần tử có khoá bằng 1 đứng trong đoạn từ vị trí $c[0] + 1$ tới vị trí $c[0] + c[1]$.
- Các phần tử có khoá bằng 2 đứng trong đoạn từ vị trí $c[0] + c[1] + 1$ tới vị trí $c[0] + c[1] + c[2]$.
- ...
- Các phần tử có khoá bằng V trong đoạn đứng từ vị trí $c[0] + c[1] + \dots + c[V - 1] + 1$ tới vị trí $c[0] + c[1] + \dots + c[V - 1] + c[V]$.
- ...
- Các phần tử có khoá bằng K trong đoạn đứng từ vị trí $c[0] + c[1] + \dots + c[K - 1] + 1$ tới vị trí $c[0] + c[1] + \dots + c[K]$.

Ví dụ: với dãy gồm 8 phần tử có dãy khoá bằng : 2, 0, 2, 5, 1, 2, 0, 3 ta có

$c[0]$	$c[1]$	$c[2]$	$c[3]$	$c[4]$	$c[5]$
2	1	3	1	0	1

Sau khi sắp xếp, các phần tử có khoá bằng 0 sẽ nằm từ vị trí 1 đến vị trí 2, phần tử có khoá bằng 1 nằm ở vị trí 3, các phần tử có khoá bằng 2 nằm từ vị trí 4 đến vị trí 6, phần tử có khoá bằng 3 nằm ở vị trí 7, các phần tử có khoá bằng 5 nằm ở vị trí 8.

Dãy khoá sau khi sắp xếp: 0, 0, 1, 2, 2, 2, 3, 5

Độ phức tạp của thuật toán là: $O(\max(N, K))$

4. Một số ví dụ ứng dụng thuật toán sắp xếp

Ví dụ 1: Giá trị nhỏ thứ k

Cho dãy a_1, a_2, \dots, a_n , các số đôi một khác nhau và số nguyên dương k ($1 \leq k \leq n$). Hãy đưa ra giá trị nhỏ thứ k trong dãy.

Ví dụ dãy gồm 5 phần tử: 5, 7, 1, 3, 4 và $k = 3$ thì giá trị nhỏ thứ k là 4.

Giải

Sắp xếp dãy theo giá trị tăng dần, số đứng thứ k của dãy là giá trị nhỏ thứ k . Nếu $n \leq 5000$ có thể sử dụng thuật toán sắp xếp nổi bọt, nhưng nếu $n > 5000$ thì nên sử dụng thuật toán sắp xếp nhanh.

Ta có thể tìm được giá trị nhỏ thứ k hiệu quả hơn (không cần phải sắp xếp lại cả dãy số) cụ thể:

+ Trong thuật toán sắp xếp nổi bọt ta chỉ cần sắp xếp đến phần tử thứ k , khi đó phần tử thứ k chính là phần tử có khoá nhỏ thứ k .

```
for i:=1 to k do // i chạy đến k
  for j:=n downto i+1 do
    if a[j-1]>a[j] then
      begin
        tmp:=a[j];
        a[j]:=a[j-1];
        a[j-1]:=tmp;
      end;
```

+ Trong thuật toán Quick Sort, ta thấy rằng:

- Nếu $k < L \leq H$ thì đoạn từ L đến H không cần sắp xếp vì đoạn này không ảnh hưởng đến vị trí thứ k .

- Nếu $L \leq H < k$ thì đoạn từ L đến H cũng không cần sắp xếp vì đoạn này không ảnh hưởng đến vị trí thứ k .
- Nếu $L \leq k \leq H$ thì ta sẽ xử lý tiếp trong đoạn này.

```

procedure QuickSort (L,H:longint);
var      i,j      :longint;
x,tmp    :longint;
begin
    if (L<=K) and (H>=K) then
        begin
            i:=L;
            j:=H;
            x:=a[(L+H) div 2];
            repeat
                while a[i]<x do inc(i);
                while a[j]>x do dec(j);
                if i<=j then
                    begin
                        tmp:=a[i];
                        a[i]:=a[j];
                        a[j]:=tmp;
                        inc(i);
                        dec(j);
                    end;
            until i>j;
            if L<j then QuickSort(L,j);
            if i<H then QuickSort(i,H);
        end;
    end;
end;

```

Sau khi gọi và thực hiện thủ tục QuickSort(1,n) thì số đứng thứ k chính là giá trị nhỏ thứ k .

Chú ý: khi X là số nhỏ thứ $(N \text{ div } 2 + 1)$ của dãy a_1, a_2, \dots, a_n thì hàm

$$F(X) = |a_1 - X| + |a_2 - X| + \dots + |a_n - X|$$

đạt giá trị nhỏ nhất

Ví dụ 2: Tìm kiếm

Cho dãy đã được sắp tăng dần $a_1 \leq a_2 \leq \dots \leq a_n$ và số X . Hãy đưa ra chỉ số i mà $a_i = X$ hoặc đưa ra $i=0$ nếu không có phần tử nào có giá trị bằng X .

Giải

Thuật toán tìm kiếm nhị phân có thể tìm phần tử có giá trị bằng X trên mảng đã được sắp xếp một cách hiệu quả trong thời gian $O(\log n)$. Thuật toán như sau:

Giả sử cần tìm trong đoạn $a[L], a[L+1], \dots, a[H]$ với giá trị cần tìm kiếm là X , trước hết ta xem xét với giá trị của phần tử nằm giữa dãy, $mid = (L + H) \div 2$

- Nếu $a[mid] < X$ thì có nghĩa là đoạn từ $a[L]$ tới $a[mid]$ chỉ chứa các phần tử có giá trị $< X$, ta tiến hành tìm kiếm tiếp với đoạn từ $a[mid+1]$ đến $a[H]$
- Nếu $a[mid] > X$ thì có nghĩa là đoạn từ $a[mid]$ tới $a[H]$ chỉ chứa các phần tử có giá trị $> X$, ta tiến hành tìm kiếm tiếp với đoạn từ $a[L]$ đến $a[mid-1]$
- Nếu $a[mid] = X$ thì việc tìm kiếm thành công (kết thúc quá trình tìm kiếm).

Quá trình tìm kiếm sẽ thất bại nếu đến một bước nào đó, đoạn tìm kiếm là rỗng ($L > H$)

```
function BinarySearch(X: longint): longint;  
var L, H, mid: longint;  
begin  
  L := 1; H := n;  
  while L ≤ H do  
    begin  
      mid := (L + H) div 2;  
      if a[mid] = X then exit(mid);  
  
      if a[mid] < X then L := mid + 1  
      else H := mid - 1;  
    end;  
  exit(0);  
end;
```

Ví dụ 3: Thống kê

Cho dãy a_1, a_2, \dots, a_n . Hãy đếm số lượng giá trị khác nhau có trong dãy và đưa ra số lần lặp của giá trị xuất hiện nhiều nhất.

Ví dụ: dãy gồm 8 số: 6, 7, 1, 7, 4, 6, 6, 8 thì dãy có 5 giá trị khác nhau và số lần lặp của giá trị xuất hiện nhiều nhất trong dãy là 3.

Giải

Các công việc trên sẽ được thực hiện đơn giản nếu mảng đã được sắp xếp, khi đó các phần tử có giá trị bằng nhau sẽ đứng cạnh nhau (liên tiếp nhau).

```
{ Hàm countValue trả về số giá trị khác nhau trong mảng a có n phần tử đã sắp xếp }
function countValue(a : TArray; n : longint) : longint;
var i, count : longint;
begin
    count:=1;
    for i:=2 to n do
        if a[i-1]<>a[i] then inc(count);
    countValue := count;
end;

{ Hàm highestFrequency trả về số lần lặp của giá trị xuất hiện nhiều nhất trong mảng a có n phần tử đã sắp xếp }
function highestFrequency(a:TArray; n:longint):longint;
var i,count,rslt :longint;
begin
    rslt:=1;
    count:=1;
    for i:=2 to n do begin
        if a[i] <> a[i-1] then count:=1
        else inc(count);
        if count>rslt then rslt:=count;
    end;
    highestFrequency := rslt;
end;
```

Ví dụ 4. Xét dãy F gồm n ($2 < n \leq 10^6$) số nguyên $F = (f_1, f_2, \dots, f_n)$ định nghĩa như sau:

$$f_i = \begin{cases} 1, & \text{nếu } 1 \leq i \leq 2 \\ (f_{i-1} + f_{i-2}) \bmod 128, & \text{nếu } 2 < i \leq n \end{cases}$$

Hãy cho biết nếu sắp xếp dãy F theo thứ tự không giảm thì số thứ k ($k \leq n$) có giá trị là bao nhiêu?

Giải

Ta nhận thấy f_i có giá trị nguyên và $0 \leq f_i \leq 127$, ta sẽ sử dụng thuật toán đếm phân phối như sau:

- Xây dựng dãy F và dãy $c[0], c[1], \dots, c[127]$, trong đó $c[V]$ là số lần xuất hiện giá trị V trong dãy F .
- Giá trị thứ k của dãy F sau khi sắp xếp là giá trị P nhỏ nhất thỏa mãn $c[0] + c[1] + \dots + c[P] \geq k$

Chú ý: Sử dụng $a \text{ and } (2^m - 1)$ thay cho $a \bmod (2^m)$, chương trình sẽ chạy nhanh hơn.

```
const    maxValue          =128 - 1;
var      fi_2, fi_1, fi    :longint;
         i, v, n, k, cv, P :longint;
         c                  :array[0..maxValue]of longint;
BEGIN
    write('Nhap n, k:');readln(n,k);
    fi_1:=1; fi_2:=1;
    fillchar(c,sizeof(c),0);
    c[fi_1]:=c[fi_1]+1;
    c[fi_2]:=c[fi_2]+1;
    for i:=3 to n do begin
        fi:=(fi_1+fi_2) and maxValue;
        {write(fi:4);}
        c[fi]:=c[fi]+1;
        fi_2:=fi_1;
        fi_1:=fi;
    end;
    cv:=0;
    for v:=0 to maxValue do begin
        cv:=cv+c[v];
        if cv >= k then begin
            P:=v;
```



```

        break;
    end;
end;
writeln(P)
END.

```

Ví dụ 5. Cho dãy gồm N ($N \leq 30000$) số tự nhiên không vượt quá 10^9 , tìm số tự nhiên nhỏ nhất không xuất hiện trong dãy.

Dữ liệu vào trong file *SN.INP* có dạng:

- Dòng đầu là số nguyên N
- Dòng thứ hai gồm N số

Kết quả ra file *SN.OUT* có dạng: số tự nhiên nhỏ nhất không xuất hiện trong dãy.

SN.INP	SN.OUT
5 5 0 3 1 4	2

Giải

Ta có nhận xét sau: số tự nhiên nhỏ nhất không xuất hiện trong dãy sẽ nằm trong đoạn $[0, n]$. Do đó, ta sử dụng mảng `c:array[0..30000] of longint;` với `c[x]` là số lần xuất hiện của x trong dãy, nếu `c[x]=0` tức là x không xuất hiện trong dãy.

```

const      Limit      =30000;
           fi          ='SN.INP';
           fo          ='SN.OUT';
var         c          :array[0..Limit] of longint;
           n           :longint;
           i,x         :longint;
           f           :text;

BEGIN
    fillchar(c,sizeof(c),0);
    assign(f,fi); reset(f);
    readln(f,n);
    for i:=1 to n do begin
        read(f,x);
        if x<=n then inc(c[x]);
    end;

```

```

close(f);
for i:=0 to n do
    if c[i]=0 then begin
        x:=i;
        break;
    end;
assign(f,fo); rewrite(f);
write(f,x);
close(f);
END.

```

Ví dụ 6. Cho chuỗi s (độ dài không vượt quá 10^6) chỉ gồm 2 ký tự 'A' và 'B'. Đếm số cách chọn cặp chỉ số (i,j) mà chuỗi con liên tiếp từ ký tự thứ i đến ký tự thứ j của chuỗi s có số lượng ký tự 'A' bằng số lượng ký tự 'B'.

Dữ liệu vào trong file “AB.INP” có dạng: gồm một dòng duy nhất chứa chuỗi s

Kết quả ra file “AB.OUT” có dạng: gồm một dòng duy nhất chứa một số là kết quả bài toán.

AB.INP	AB.OUT
ABAB	4

Giải

```

const
    MAX          =1000000;
    fi           ='AB.INP';
    fo           ='AB.OUT';
var
    s            :ansistring;
    c            :array[-MAX..MAX]of longint;
    f            :text;
    i, sum       :longint;
    count        :int64;
BEGIN
    assign(f,fi); reset(f);
    read(f,s);
    close(f);
    fillchar(c,sizeof(c),0);
    c[0]:=1;
    sum:=0;

```

```

count:=0;
for i:=1 to length(s) do begin
    if s[i]='A' then sum:=sum - 1
    else sum:=sum + 1;
    count:=count + c[sum];
    inc(c[sum]);
end;
assign(f,fo); rewrite(f);
write(f,count);
close(f);
END.

```

Bài tập

3.1. Cho một danh sách n học sinh ($1 \leq n \leq 200$), mỗi học sinh có thông tin sau:

- Họ và tên: Là một xâu kí tự độ dài không quá 30 (các từ cách nhau một dấu cách)
- Điểm: Là một số thực

A) Đưa ra danh sách họ và tên đã sắp xếp theo thứ tự abc (ưu tiên tên, họ, đệm)

B) Có bao nhiêu tên khác nhau trong danh sách, liệt kê các tên đó.

C) Chọn những học sinh có thứ hạng 1, 2, 3 điểm cao nhất trong danh sách để trao học bổng, hãy cho biết tên những học sinh đó.

Ví dụ

Dữ liệu vào	Kết quả câu A	Kết quả câu B	Kết quả câu C
6 Vu Anh Quan 8.9 Nguyen Van Chung 8.7 Hoang Trong Quynh 8.5	Nguyen Van Chung Cong Hoang Dinh Quang Hoang Dinh Quang Huy Vu Anh Quan Hoang Trong	5 Chung Hoang Huy Quan Quynh	Vu Anh Quan Dinh Quang Huy Dinh Quang Hoang Nguyen Van Chung

Dinh Hoang 8.7	Quang	Quynh		
Dinh Huy 8.8	Quang			
Cong Hoang 8.0				

3.2. Cho dãy số gồm N số nguyên $a_1 \leq a_2 \leq \dots \leq a_N$

A) Đưa ra thuật toán có độ phức tạp $O(N \log N)$ để tìm 2 chỉ số $i < j$ mà $a_i + a_j = 0$.

B) Đưa ra thuật toán có độ phức tạp $O(N^2 \log N)$ để tìm 3 chỉ số $i < j < k$ mà $a_i + a_j + a_k = 0$.

C) Đưa ra thuật toán có độ phức tạp $O(N)$ để tìm 2 chỉ số $i < j$ mà $a_i + a_j = 0$.

D) Đưa ra thuật toán có độ phức tạp $O(N^2)$ để tìm 3 chỉ số $i < j < k$ mà $a_i + a_j + a_k = 0$.

3.3. Cho một chuỗi s (độ dài không quá 200) chỉ gồm các kí tự 'a' đến 'z', đếm số lượng chuỗi con liên tiếp khác nhau nhận được từ chuỗi s .

Ví dụ: $s = 'abab'$, ta có các chuỗi con liên tiếp khác nhau là:

'a', 'b', 'ab', 'ba', 'aba', 'bab', 'abab', số lượng chuỗi con liên tiếp khác nhau là 7.

3.4. Viết liên tiếp các số tự nhiên từ 1 đến N ta được một số nguyên M . Ví dụ $N=15$ ta có $M=123456789101112131415$. Hãy tìm cách xoá đi K chữ số của số M để nhận được số M' là lớn nhất.

3.5. Xét tập $F(N)$ tất cả các số hữu tỷ trong đoạn $[0,1]$ với mẫu số không vượt quá N ($1 < N \leq 100$).

Ví dụ tập $F(5)$: $0/1 \quad 1/5 \quad 1/4 \quad 1/3 \quad 2/5 \quad 1/2 \quad 3/5 \quad 2/3 \quad 3/4 \quad 4/5 \quad 1/1$

Sắp xếp các phân số trong tập $F(N)$ theo thứ tự tăng dần, đưa ra phân số thứ K .

3.6. Cho chuỗi s (độ dài không vượt quá 10^6) chỉ gồm các kí tự 'a' đến 'z',

A) Có bao nhiêu loại kí tự xuất hiện trong s

- B) Đưa ra một kí tự xuất hiện nhiều nhất trong xâu s và số lần xuất hiện của kí tự đó.
- 3.7.** Cho 2 dãy $a_1 \leq a_2 \leq \dots \leq a_n$ và $b_1 \leq b_2 \leq \dots \leq b_m$, hãy đưa ra thuật toán có độ phức tạp $O(n + m)$ để có dãy $c_1 \leq c_2 \leq \dots \leq c_{n+m}$ là dãy trộn của hai dãy trên.
- 3.8.** Cho dãy số gồm n ($n \leq 10000$) số nguyên a_1, a_2, \dots, a_n ($|a_i| \leq 10^9$), tìm số nguyên X bất kì để $S = |a_1 - X| + |a_2 - X| + \dots + |a_n - X|$ đạt giá trị nhỏ nhất, có bao nhiêu giá trị nguyên khác nhau thoả mãn.
 Ví dụ 1: dãy gồm 5 số 3, 1, 5, 4, 5, ta có duy nhất một giá trị $X = 4$ để S đạt giá trị nhỏ nhất bằng 6.
 Ví dụ 2: dãy gồm 6 số 3, 1, 7, 2, 5, 7 ta có ba giá trị nguyên của X là 3, 4, 5 để S đạt giá trị nhỏ nhất bằng 13.
- 3.9.** Cho N ($N \leq 10000$) điểm trên mặt phẳng Oxy, điểm thứ i có tọa độ là (x_i, y_i) . Ta định nghĩa khoảng cách giữa 2 điểm $P(x_p, y_p)$ và $Q(x_q, y_q)$ bằng $|x_p - x_q| + |y_p - y_q|$. Hãy tìm điểm A có tọa độ nguyên mà tổng khoảng cách (theo cách định nghĩa trên) từ A tới N điểm đã cho là nhỏ nhất ($|x_i|, |y_i|$ nguyên không vượt quá 10^9)
- 3.10.** Cho N ($N \leq 10000$) đoạn thẳng trên trục số với các điểm đầu x_i và độ dài d_i ($|x_i|, d_i$ là những số nguyên và không vượt quá 10^9). Tính tổng độ dài trên trục số bị phủ bởi N đoạn trên.
 Ví dụ: có 3 đoạn $x_1 = -5, d_1 = 10; x_2 = 0, d_2 = 6; x_3 = -100, d_3 = 10$ thì tổng độ dài trên trục số bị phủ bởi 3 đoạn trên là: 21
- 3.11.** Cho N ($N \leq 300$) điểm trên mặt phẳng Oxy, điểm thứ i có tọa độ là (x_i, y_i) . Hãy đếm số cách chọn 4 điểm trong N điểm trên mà 4 điểm đó tạo thành 4 đỉnh của một hình chữ nhật. ($|x_i|, |y_i|$ nguyên không vượt quá 1000)
 Ví dụ: có 5 điểm $(0, 0), (0, 1), (1, 0), (-1, 0), (0, -1)$ có duy nhất 1 cách chọn 4 điểm mà 4 điểm đó tạo thành 4 đỉnh của một hình chữ nhật.
- 3.12.** Cho N ($N \leq 10000$) đoạn số nguyên $[a_i, b_i]$, hãy tìm một số mà số đó thuộc nhiều đoạn số nguyên nhất.
 Ví dụ: có 5 đoạn $[0,10], [2,3], [4,7], [3,5], [5,8]$, ta chọn số 5 thuộc 4 đoạn $[0,10], [4,7], [3,5], [5,8]$.
- 3.13.** Cho dãy gồm N ($N \leq 10000$) số a_1, a_2, \dots, a_N . Hãy tìm dãy con liên tiếp dài nhất có tổng bằng 0. ($|a_i| \leq 10^9$)

Ví dụ: dãy gồm 5 số 2, 1, -2, 3, -2 thì dãy con liên tiếp dài nhất có tổng bằng 0 là: 1, -2, 3, -2

3.14. ESEQ

Cho dãy số nguyên A gồm N phần tử A_1, A_2, \dots, A_N , tìm số cặp chỉ số i, j thỏa mãn:

$$\sum_{p=1}^i A_p = \sum_{q=j}^N A_q \text{ với } 1 \leq i < j \leq N$$

Dữ liệu vào trong file “ESEQ.INP” có dạng:

- Dòng đầu là số nguyên dương N ($2 \leq N \leq 10^5$)
- Dòng tiếp theo chứa N số nguyên A_1, A_2, \dots, A_N ($|A_i| < 10^9$), các số cách nhau một dấu cách.

Kết quả ra file “ESEQ.OUT” có dạng: gồm một số là số cặp tìm được.

ESEQ.INP	ESEQ.OUT
3	3
1 0 1	

3.15. GHÉP SỐ

Cho n số nguyên dương a_1, a_2, \dots, a_n ($1 < n \leq 100$), mỗi số không vượt quá 10^9 . Từ các số này người ta tạo ra một số nguyên mới bằng cách ghép tất cả các số đã cho, tức là viết liên tiếp các số đã cho với nhau. Ví dụ, với $n = 4$ và các số 123, 124, 56, 90 ta có thể tạo ra các số mới sau: 1231245690, 1241235690, 5612312490, 9012312456, 9056124123,... Có thể dễ dàng thấy rằng, với $n = 4$, ta có thể tạo ra 24 số mới. Trong trường hợp này, số lớn nhất có thể tạo ra là 9056124123.

Yêu cầu: Cho n và các số a_1, a_2, \dots, a_n . Hãy xác định số lớn nhất có thể tạo ra khi ghép các số đã cho thành một số mới.

Dữ liệu vào từ file văn bản NUMJOIN.INP có dạng:

- Dòng thứ nhất chứa số nguyên n ,
- Dòng thứ 2 chứa n số nguyên $a_1 a_2 \dots a_n$.

Kết quả ra file văn bản NUMJOIN.OUT gồm một dòng là số lớn nhất có thể tạo ra khi ghép các số đã cho thành một số mới.

3.16. **GIÁ TRỊ NHỎ NHẤT**

Cho bảng số A gồm MxN ô, mỗi ô chứa một số nguyên không âm (A_{ij}) có giá trị không vượt quá 10^9 . Xét hàng i và hàng j của bảng, ta cần xác định X_{ij} nguyên để:

$$S_{ij} = \left(\sum_{k=1}^N |A_{ik} - X_{ij}| \right) + \left(\sum_{k=1}^N |A_{jk} - X_{ij}| \right) \text{ đạt giá trị nhỏ nhất.}$$

Tính $W = \sum_{i=1}^{M-1} \sum_{j=i+1}^M S_{ij}$

Dữ liệu vào trong file “WMT.INP” có dạng:

- Dòng đầu là 2 số nguyên dương M, N ($1 < M, N < 1001$)
- M dòng sau, mỗi dòng N số

Kết quả ra file “WMT.OUT” có dạng: gồm một số W

WMT . INP	WMT . OUT
2 3	5
2 3 1	
2 3 4	

3.17. **DECIPHERING THE MAYAN WRITING** (IOI 2006)

Công việc giải mã chữ viết của người MAIA là khó khăn hơn người ta tưởng nhiều. Trải qua hơn 200 năm mà người ta vẫn hiểu rất ít về các chữ viết này. Chỉ trong 3 thập niên gần đây do công nghệ phát triển việc giải mã này mới có nhiều tiến bộ.

Chữ viết Maia dựa trên các kí hiệu nhỏ gọi là nét vẽ, mỗi nét vẽ tương ứng với một âm giọng nói. Mỗi từ trong chữ viết Maia sẽ bao gồm một tập hợp các nét vẽ như vậy kết hợp lại với nhiều kiểu dáng khác nhau. Mỗi nét vẽ có thể hiểu là một kí tự ta hiểu ngày nay.

Một trong những vấn đề lớn khi giải mã chữ Maia là thứ tự đọc các nét vẽ. Do người Maia trình bày các nét vẽ này không theo thứ tự phát âm, mà theo cách thể hiện của chúng. Do vậy nhiều khi đã biết hết các nét vẽ của một từ rồi nhưng vẫn không thể tìm ra được chính xác cách ghi và đọc của từ này.

Các nhà khảo cổ đang đi tìm kiếm một từ đặc biệt W. Họ đã biết rõ tất cả các nét vẽ của từ này nhưng vẫn chưa biết các cách viết ra của từ này. Vì họ biết có các thí sinh IOI'06 sẽ đến nên muốn sự trợ giúp của các sinh viên này. Họ sẽ đưa ra toàn bộ g nét vẽ của từ W và dãy S tất cả các nét vẽ có

trong hàng đá cổ. Bạn hãy giúp các nhà khảo cổ tính xem có bao nhiêu khả năng xuất hiện từ W trong hàng đá.

Yêu cầu: Hãy viết chương trình, cho trước các kí tự của từ W và dãy S các nét vẽ trong hàng đá, tính tổng số khả năng xuất hiện của từ W trong dãy S, nghĩa là số lần xuất hiện một hoán vị các kí tự của dãy g kí tự trong S.

Các ràng buộc

$1 \leq g \leq 3\,000$, số nét vẽ trong W

$g \leq |S| \leq 3\,000\,000$, |S| là số các nét vẽ của dãy S

Dữ liệu vào:

- Dòng 1: chứa 2 số g và |S| cách nhau bởi dấu cách.
- Dòng 2: chứa g kí tự liên nhau là các nét vẽ của từ W. Các kí tự hợp lệ là 'a'-'z' và 'A'-'Z'. Các chữ in hoa và in thường là khác nhau.
- Dòng 3: Chứa |S| kí tự là dãy các nét vẽ tìm thấy trong hàng. Các kí tự hợp lệ là 'a'-'z' và 'A'-'Z'. Các chữ in hoa và in thường là khác nhau.

Kết quả ra:

Chứa đúng 1 số là khả năng xuất hiện của từ W trong dãy S.

Dữ liệu vào	Kết quả ra
4 11 cAda AbrAcadAbRa	2

3.18. TRÒ CHƠI VỚI DÃY SỐ (Học sinh giỏi quốc gia, 2007-2008)

Hai bạn học sinh trong lúc nhàn rỗi nghĩ ra trò chơi sau đây. Mỗi bạn chọn trước một dãy số gồm n số nguyên. Giả sử dãy số mà bạn thứ nhất chọn là:

b_1, b_2, \dots, b_n

còn dãy số mà bạn thứ hai chọn là: c_1, c_2, \dots, c_n

Mỗi lượt chơi mỗi bạn đưa ra một số hạng trong dãy số của mình. Nếu bạn thứ nhất đưa ra số hạng b_i ($1 \leq i \leq n$), còn bạn thứ hai đưa ra số hạng c_j ($1 \leq j \leq n$) thì giá của lượt chơi đó sẽ là $|b_i + c_j|$.

Ví dụ: Giả sử dãy số bạn thứ nhất chọn là 1, -2; còn dãy số mà bạn thứ hai chọn là 2, 3. Khi đó các khả năng có thể của một lượt chơi là (1, 2), (1, 3), (-2, 2), (-2, 3). Như vậy, giá nhỏ nhất của một lượt chơi trong số các lượt chơi có thể là 0 tương ứng với giá của lượt chơi (-2, 2).

Yêu cầu: Hãy xác định giá nhỏ nhất của một lượt chơi trong số các lượt chơi có thể.

Dữ liệu vào:

Dòng đầu tiên chứa số nguyên dương n ($n \leq 10^5$)

Dòng thứ hai chứa dãy số nguyên b_1, b_2, \dots, b_n ($|b_i| \leq 10^9, i = 1, 2, \dots, n$)

Dòng thứ hai chứa dãy số nguyên c_1, c_2, \dots, c_n ($|c_i| \leq 10^9, i = 1, 2, \dots, n$)

Hai số liên tiếp trên một dòng được ghi cách nhau bởi dấu cách.

Kết quả ra:

Ghi ra giá nhỏ nhất tìm được.

Dữ liệu vào	Kết quả ra
2 1 -2 2 3	0

3.19. DÃY SỐ (Học sinh giỏi, Hà Nội 2008-2009)

Cho dãy số nguyên a_1, a_2, \dots, a_n . Số a_p ($1 \leq p \leq n$) được gọi là một số trung bình cộng trong dãy nếu tồn tại 3 chỉ số i, j, k ($1 \leq i, j, k \leq n$) đôi một khác nhau, sao cho $a_p = (a_i + a_j + a_k)/3$

Yêu cầu: Cho n và dãy số a_1, a_2, \dots, a_n . Hãy tìm số lượng các số trung bình cộng trong dãy.

Dữ liệu vào:

- Dòng đầu ghi số nguyên dương n ($3 \leq n \leq 1000$)
- Dòng thứ hai chứa n số nguyên a_i ($|a_i| < 10^8$)

Kết quả ra:

Số lượng các số trung bình cộng trong dãy.

Dữ liệu vào	Kết quả ra
5 4 3 6 3 5	2

3.20. ĐẾM SỐ TAM GIÁC (Tin học trẻ, bảng B, năm 2009)

Cho ba số nguyên dương a, b, m và n ($m, n \leq 10000$) đoạn thẳng đánh số từ 1 tới n . Đoạn thẳng thứ i có độ dài d_i ($\forall i: 1 \leq i \leq n$), ở đây các độ dài (d_1, d_2, \dots, d_n) được cho như sau:

$$d_i = \begin{cases} b, & \text{nếu } i = 1 \\ (a \times d_{i-1} + b) \bmod m + 1, & \text{nếu } 1 < i \leq n \end{cases} \quad (*)$$

Hãy cho biết có bao nhiêu tam giác khác nhau có thể được tạo ra bằng cách lấy đúng ba đoạn trong số n đoạn thẳng đã cho làm ba cạnh (hai tam giác bằng nhau nếu chúng có ba cặp cạnh tương ứng bằng nhau, nếu không chúng được coi là khác nhau).

Ví dụ với $a = 6; b = 3; m = 4; n = 5$. Ta có 5 đoạn thẳng với độ dài của chúng tính theo công thức (*) là $(3, 2, 4, 4, 4)$. Với 5 đoạn thẳng này có thể tạo ra được 4 tam giác với độ dài các cạnh được chỉ ra như sau:

Tam giác 1: $(2, 3, 4)$

Tam giác 2: $(2, 4, 4)$

Tam giác 3: $(3, 4, 4)$

Tam giác 4: $(4, 4, 4)$

Chuyên đề 4

THIẾT KẾ GIẢI THUẬT

Chuyên đề này trình bày các chiến lược thiết kế thuật giải như: Quay lui (Backtracking), Nhánh và cận (Branch and Bound), Tham ăn (Greedy Method), Chia để trị (Divide and Conquer) và 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.

1. 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 lý 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.

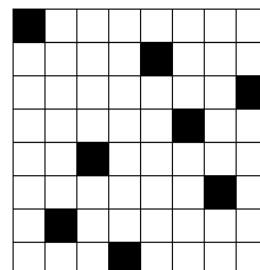
1.1. Phương pháp

Trong nhiều bài toán, việc tìm nghiệm có thể quy về việc tìm vector hữu hạn $(x_1, x_2, \dots, x_n, \dots)$, 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 x_i được chọn ra từ tập hữu hạn A_i .

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 8×8 , 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 ($i=1,2,\dots,8$). Gọi x_i là cột mà quân hậu i đứng. Như vậy nghiệm của bài toán là vector (x_1, x_2, \dots, x_8) , trong đó $1 \leq x_i \leq 8$, tức là x_i được chọn từ tập $A_i = \{1, 2, \dots, 8\}$. Vector (x_1, x_2, \dots, x_8) là nghiệm nếu $x_i \neq x_j$ và hai ô $(i, x_i), (j, x_j)$ 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 (\emptyset). Thành phần đầu tiên x_1 được chọn ra từ tập $S_1 = A_1$. Giả sử đã chọn được các thành phần x_1, x_2, \dots, x_{i-1} thì từ các điều kiện của bài toán ta xác định được tập S_i (các ứng cử viên có thể chọn làm thành phần x_i , S_i là tập con của A_i). Chọn một phần tử x_i từ S_i ta mở rộng nghiệm được x_1, x_2, \dots, x_i . Lặp lại quá trình trên để tiếp tục mở rộng nghiệm. Nếu không thể chọn được thành phần x_{i+1} (S_{i+1} rỗng) thì ta quay lại chọn một phần tử khác của S_i cho x_i . Nếu không còn một phần tử nào khác của S_i ta quay lại chọn một phần tử khác của S_{i-1} làm x_{i-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:

```

procedure Backtrack;
begin
  S1 := A1;
  k := 1;
  while k > 0 do begin

```

```

while  $S_k \neq \emptyset$  do begin
    <chọn  $x_k \in S_i$ >;
     $S_k := S_k - \{x_k\}$ ;
    if  $(x_1, x_2, \dots, x_k)$  là nghiệm then <Đưa ra nghiệm>;
     $k := k+1$ ;
    <Xác định  $S_k$ >;
end;
 $k := k-1$ ; // quay lui
end;
end;

```

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

```

procedure Backtrack(i) ;// xây dựng thành phần thứ i
begin
    <Xác định  $S_i$ >;
    for  $x_i \in S_i$  do begin
        <ghi nhận thành phần thứ i>;
        if (tìm thấy nghiệm) then <Đưa ra nghiệm>
        else Backtrack(i+1);
        <loại thành phần i>;
    end;
end;

```

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 $(x_1, x_2, \dots, x_i, \dots)$.
- Xác định tập S_i các ứng cử viên được chọn làm thành phần thứ i của nghiệm. Chọn cách thích hợp để biểu diễn S_i .
- Tìm các điều kiện để một vector đã chọn là nghiệm của bài toán.

1.2. Một số ví dụ áp dụng

1.2.1. 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 $X = (x_1, x_2, \dots, x_k)$
- x_i lấy giá trị trong tập $\{1, 2, \dots, n\}$
- Ràng buộc: $x_i < x_{i+1}$ 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: $1 \leq x_1 < x_2 < \dots < x_k \leq n$, do đó tập S_i (tập các ứng cử viên được chọn làm thành phần thứ i) là từ $x_{i-1} + 1$ đến $(n - k + i)$. Để điều này đúng cho cả trường hợp $i = 1$, ta thêm vào $x_0 = 0$.

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

```
program ToHop;
const   MAX      =20;
type    vector   =array[0..MAX]of longint;
var      x        :vector;
          n,k      :longint;
procedure GhiNghiem(x:vector);
var i    :longint;
begin
  for i:=1 to k do write(x[i], ' ');
  writeln;
end;
procedure ToHop(i:longint);
var j:longint;
begin
  for j := x[i-1]+1 to n-k+i do begin
    x[i] := j;
    if i=k then GhiNghiem(x)
    else ToHop(i+1);
  end;
end;
```

```

end;
end;
BEGIN
    write('Nhap n, k:'); readln(n,k);
    x[0]:=0;
    ToHop(1);
END.

```

Ví dụ về Input / Output của chương trình:

n = 4, k=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:

$$C_n^k = \frac{n(n-1) \dots (n-k+1)}{k!} = \frac{n!}{k!(n-k)!} = C_4^2 = 6$$

1.2.2. Chinh hợp lặp

Chinh 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 chinh hợp lặp là các **dãy nhị phân**. Một dãy nhị phân độ dài m là một chinh 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 chinh hợp lặp chập k của tập n phần tử yêu cầu tìm các nghiệm như sau:

- Là một vector $X = (x_1, x_2, \dots, x_k)$
- x_i lấy giá trị trong tập $\{1, 2, \dots, n\}$
- 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.

Thủ tục đệ quy sau sinh tất cả chỉnh hợp lặp chập k của n }

```
procedure ChinhHopLap(i:longint);
var j:longint;
begin
    for j := 1 to n do begin
        x[i] := j;
        if i=k then GhiNghiem(x)
        else ChinhHopLap(i+1);
    end;
end;
```

Ví dụ về Input/Output của chương trình:

n = 2, k=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:

$$\bar{A}_n^k = n^k = 2^3 = 8$$

1.2.3. 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 ($k \leq n$) 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: $X = (x_1, x_2, \dots, x_k)$
- x_i lấy giá trị trong tập $\{1, 2, \dots, n\}$
- Ràng buộc: các giá trị x_i đôi một khác nhau, tức là $x_i \neq x_j$ với mọi $i \neq j$.

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.

```
program ChinhHopKhongLap;
const   MAX      =20;
type    vector   =array[0..MAX]of longint;
var      x        :vector;
          d        :array[1..MAX]of longint; { mảng d để kiểm
soát ràng buộc các giá trị  $x_i$  đôi một khác nhau,  $x_i \neq x_j$  với mọi  $i \neq j$ }
          n,k      :longint;
procedure GhiNghiem(x:vector);
var i    :longint;
begin
  for i:=1 to k do write(x[i], ' ');
  writeln;
end;
procedure ChinhHopKhongLap(i:longint);
var j:longint;
begin
  for j := 1 to n do
    if d[j]=0 then begin
      x[i] := j;
      d[j] := 1;
      if i=k then GhiNghiem(x)
      else ChinhHopKhongLap(i+1);
      d[j] := 0;
    end;
end;
end;
BEGIN
  write('Nhap n, k(k<=n): '); readln(n,k);
```

```

fillchar(d, sizeof(d), 0);
ChinhHopKhongLap(1);
END.

```

Ví dụ về Input / Output của chương trình:

n = 3, k=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:

$$A_n^k = n(n-1) \dots (n-k+1) = \frac{n!}{(n-k)!} = 6$$

1.2.4. Bài toán xếp 8 quân hậu

Trong bài toán 8 quân hậu, nghiệm của bài toán có thể biểu diễn dưới dạng vector (x_1, x_2, \dots, x_8) thỏa mãn:

- 1) x_i là tọa độ cột của quân hậu đang đứng ở dòng thứ i , $x_i \in \{1, 2, \dots, 8\}$.
- 2) Các quân hậu không đứng cùng cột tức là $x_i \neq x_j$ với $i \neq j$.

3) Có thể dễ dàng nhận ra rằng hai ô (x_1, y_1) và (x_2, y_2) nằm trên cùng đường chéo chính (trên xuống dưới) nếu: $x_1 - y_1 = x_2 - y_2$, hai ô (x_1, y_1) và (x_2, y_2) nằm trên cùng đường chéo phụ (từ dưới lên trên) nếu: $x_1 + y_1 = x_2 + y_2$, nên điều kiện để hai quân hậu xếp ở hai ô $(i, x_i), (j, x_j)$ không nằm trên cùng một đường

chéo là:
$$\begin{cases} (i - x_i) \neq (j - x_j) \\ (i + x_i) \neq (j + x_j) \end{cases}$$

	1	2	3	4	5	6	7	8
1	■							
2		■						■
3			■				■	
4				■		■		
5					■			
6				■		■		
7			■				■	
8		■						■

Do đó, khi đã chọn được $(x_1, x_2, \dots, x_{k-1})$ thì x_k được chọn phải thỏa mãn các điều kiện:

$$\begin{cases} x_k \neq x_i \\ k - x_k \neq i - x_i \\ k + x_k \neq i + x_i \end{cases} \quad \text{với mọi } 1 \leq i < k$$

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

```

program XepHau;
type      vector  =array[1..8]of longint;
var        x      :vector;
procedure GhiNghiem(x:vector);
var i      :longint;
begin
    for i:=1 to 8 do write(x[i], ' ');
    writeln;
end;
procedure XepHau(k:longint);
var      Sk      :array[1..8]of longint;
        xk,i,nSk  :longint;
        ok       :boolean;
begin
    {Xác định tập  $S_k$  là tập các ứng cử viên có thể chọn làm thành phần  $x_k$ }
    nSk:=0; {lực lượng của tập  $S_k$ }
    for xk:=1 to 8 do {thử lần lượt từng giá trị 1, 2, ..., 8}
        begin
            ok:=true;
            {kiểm tra giá trị có thể chọn làm ứng cử viên cho  $x_k$  được hay không}
            for i:=1 to k-1 do
                if      not ((xk<>x[i]) and (k-xk<>i-x[i])
and (k+xk<>i+x[i])) then
                    begin
                        ok:=false;
                        break;
                    end;
            if ok then begin {có thể chọn làm ứng cử viên cho  $x_k$ , kết nạp
vào tập  $S_k$ }
                inc(nSk);
                Sk[nSk]:=xk;
            end;
        end;
    {chọn giá trị  $x_k$  từ tập  $S_k$ }
    for i:=1 to nSk do begin
        x[k]:=Sk[i];

```

```

        if k=8 then GhiNghiem(x)
        else XepHau(k+1);
        x[k]:=0;
    end;
end;
BEGIN
    XepHau(1);
END.

```

Việc xác định tập S_k 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 *chính hợp không lặp*), đánh dấu đường chéo chính $(i-x[i])$ và đánh dấu đường chéo phụ $(i+x[i])$.

```

const    n            =8;
type     vector        =array[1..n]of longint;
var       cot           :array[1..n]of longint;
          cheoChinh     :array[1-n..n-1]of longint;
          cheoPhu       :array[1+1..n+n]of longint;
          x             :vector;
procedure GhiNghiem(x:vector);
var i     :longint;
begin
    for i:=1 to n do write(x[i], ' ');
    writeln;
end;
procedure xepHau(k:longint);
var i :longint;
begin
    for i:=1 to n do
        if (cot[i]=0) and (cheoChinh[k-i]=0) and
(cheoPhu[k+i]=0) then
            begin
                x[k]:=i;
                cot[i]:=1;
                cheoChinh[k-i]:=1;
                CheoPhu[k+i]:=1;
                if k=n then GhiNghiem(x)
                else xepHau(k+1);
                cot[i]:=0;
            end;
    end;
end;

```

```

        cheoChinh[k-i]:=0;
        CheoPhu[k+i]:=0;
    end;
end;
BEGIN
    fillchar(cot,sizeof(cot),0);
    fillchar(cheoChinh,sizeof(cheoChinh),0);
    fillchar(cheoPhu,sizeof(cheoPhu),0);
    xepHau(1);
END.

```

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

```

1.2.5. Bài toán máy rút tiền tự động ATM

Một máy ATM hiện có n ($n \leq 20$) tờ tiền có giá t_1, t_2, \dots, t_n . Hãy đưa ra một cách trả với số tiền đúng bằng S .

Dữ liệu vào từ file “ATM.INP” có dạng:

- Dòng đầu là 2 số n và S
- Dòng thứ 2 gồm n số t_1, t_2, \dots, t_n

Kết quả ra file “ATM.OUT” có dạng: Nếu có thể trả đúng S thì đưa ra cách trả, nếu không ghi -1.

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

$X = (x_1, x_2, \dots, x_n)$ là nghiệm nếu: $x_1 \times t_1 + x_2 \times t_2 + \dots + x_n \times t_n = S$

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.

```
const      MAX      =20;
           fi        ='ATM.INP';
           fo        ='ATM.OUT';
type       vector    =array[1..MAX]of longint;
var        t         :array[1..MAX]of longint;
           x,xs      :vector;
           n,s,sum    :longint;
           ok         :boolean;

procedure input;
var  f    :text;
     i    :longint;
begin
  assign(f,fi); reset(f);
  readln(f,n, s);
  for i:=1 to n do read(f,t[i]);
  close(f);
end;
procedure check(x:vector);
var i    :longint;
     f    :text;
begin
  if sum = s then begin
    xs:=x;
    ok:=true;
  end;
end;
procedure printResult;
var i    :longint;
     f    :text;
begin
  assign(f,fo); rewrite(f);
  if ok then begin
    for i:=1 to n do
```

```

        if xs[i]=1 then write(f,t[i], ' ');
    end
    else write(f, '-1');
    close(f);
end;
procedure backTrack(i:longint);
var j :longint;
begin
    for j:=0 to 1 do begin
        x[i]:=j;
        sum:=sum + x[i]*t[i];
        if (i=n) then check(x)
        else if sum<=s then backTrack(i+1);
        if ok then exit; {nếu đã tìm được nghiệm thì không duyệt nữa}
        sum:=sum - x[i]*t[i];
    end;
end;
BEGIN
    input;
    ok:=false;
    sum:=0;
    backTrack(1);
    PrintResult;
END.

```

2. Nhánh và cận

2.1. 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 (x_1, x_2, \dots, x_n) , mỗi thành phần x_i ($i = 1, 2, \dots, n$) được chọn ra từ tập S_i . Mỗi nghiệm của bài toán

$X = (x_1, x_2, \dots, x_n)$, 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 (x_1, x_2, \dots, x_k) của nghiệm và khi mở rộng nghiệm $(x_1, x_2, \dots, x_{k+1})$, nếu biết rằng tất cả các nghiệm mở rộng của nó $(x_1, x_2, \dots, x_{k+1}, \dots)$ đều không tốt bằng nghiệm tốt nhất đã biết ở thời điểm đó, thì ta không cần mở rộng từ (x_1, x_2, \dots, x_k) 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:

```
procedure BranchBound(i) ;// xây dựng thành phần thứ i
begin
    <Đánh giá các nghiệm mở rộng>;
    if (các nghiệm mở rộng đều không tốt hơn
BestSolution) then exit;
    <Xác định  $S_i$ >;
    for  $x_i \in S_i$  do begin
        <ghi nhận thành phần thứ i>;
        if (tìm thấy nghiệm) then <Cập nhật BestSolution>
        else BranchBound(i+1);
        <loại thành phần i>;
    end;
end;
```

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

2.2. 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 $C_{ij} = C_{ji}$ 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 đồ 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)

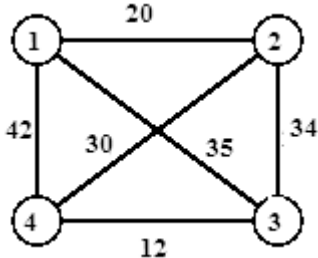
Dữ liệu vào trong file “TSP.INP” có dạng:

- Dòng đầu chứa số $n(1 < n \leq 20)$, là số thành phố.
- n dòng tiếp theo, mỗi dòng n số mô tả mảng C

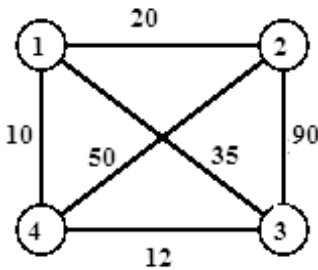
Kết quả ra file “TSP.OUT” có dạng:

- Dòng đầu là chi phí ít nhất
- Dòng thứ hai mô tả hành trình

Ví dụ 1:

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

Ví dụ 2:

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

Giải

- 1) Hành trình cần tìm có dạng $(x_1 = 1, x_2, \dots, x_n, x_{n+1} = 1)$, ở đây giữa x_i và x_{i+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 (x_1, x_2, \dots, x_n) lập thành một hoán vị của $(1, 2, \dots, n)$.
- 2) Duyệt quay lui: x_2 có thể chọn một trong các thành phố mà x_1 có đường đi trực tiếp tới, với mỗi cách thử chọn x_2 như vậy thì x_3 có thể chọn một trong các thành phố mà x_2 có đường đi tới (ngoài x_1). Tổng quát: x_i có thể chọn 1 trong các thành phố chưa đi qua mà từ x_{i-1} có đường đi trực tiếp tới. ($2 \leq i \leq n$).
- 3) Nhánh cận: Khởi tạo cấu hình BestSolution có chi phí $= +\infty$. Với mỗi bước thử chọn x_i 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 x_n ta kiểm tra xem x_n có đường đi trực tiếp về 1 không? Nếu có đánh giá chi phí đi từ thành phố 1 đến thành phố x_n cộng với chi phí từ x_n đi trực tiếp về 1, nếu nhỏ hơn chi phí của đường đi BestSolution thì cập nhật lại BestSolution bằng cách đi mới.

```
program TSP;
const      MAX           =20;
           oo            =1000000;
           fi            ='TSP.INP';
           fo            ='TSP.OUT';
var         c             :array[1..MAX,1..MAX]of longint;
           x,bestSolution :array[1..MAX]of longint;
           d             :array[1..MAX]of longint;
           n             :longint;
           sum,best      :longint;
procedure input;
var f       :text;
    i,j,k   :longint;
begin
  assign(f,fi); reset(f);
  read(f,n);
  for i:=1 to n do
    for j:=1 to n do read(f,C[i,j]);
  close(f);
```

```

end;
procedure update;
begin
    if sum+C[x[n],x[1]]<best then begin
        best:=sum+C[x[n],x[1]];
        bestSolution:=x;
    end;
end;
procedure branchBound(i:longint);
var j    :longint;
begin
    if sum>=best then exit;
    for j:=1 to n do
        if d[j]=0 then begin
            x[i]:=j;
            d[j]:=1;
            sum:=sum + C[x[i-1],j];
            if i=n then update
            else branchBound(i+1);
            sum:=sum - C[x[i-1],j];
            d[j]:=0;
        end;
    end;
end;
procedure init;
begin
    fillchar(d,sizeof(d),0);
    d[1]:=1;
    x[1]:=1;
    best:=oo;
end;
procedure output;
var f    :text;
    i    :longint;
begin
    assign(f,fo); rewrite(f);
    writeln(f,best);
    for i:=1 to n do write(f,bestSolution[i],'->');
    write(f,bestSolution[1]);
    close(f);
end;

```

```

end;
BEGIN
    input;
    init;
    branchBound(2);
    output;
END.

```

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.

2.3. Bài toán máy rút tiền tự động ATM

Bài toán

Một máy ATM hiện có n ($n \leq 20$) tờ tiền có giá t_1, t_2, \dots, t_n . Hãy tìm cách trả ít tờ nhất với số tiền đúng bằng S .

Dữ liệu vào từ file “ATM.INP” có dạng:

- Dòng đầu là 2 số n và S
- Dòng thứ 2 gồm n số t_1, t_2, \dots, t_n

Kết quả ra file “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	5
200 10 20 20 50 50 50 50 100 100	20 20 50 100 200

Giải

Như ta đã biết, nghiệm của bài toán là một dãy nhị phân độ dài n , giả sử đã xây dựng được k thành phần (x_1, x_2, \dots, x_k) , đã trả được sum và sử dụng c tờ. Để đánh giá được các nghiệm mở rộng của (x_1, x_2, \dots, x_k) , ta nhận thấy:

- Còn phải trả $S - sum$
- Gọi $tmax[k]$ là giá cao nhất trong các tờ tiền còn lại ($tmax[k] = \max\{t_{k+1}, \dots, t_n\}$) thì ít nhất cần sử dụng thêm $\frac{S-sum}{tmax[k]}$ tờ nữa.

Do đó, nếu $c + \frac{S-sum}{tmax[k]}$ mà lớn hơn hoặc bằng số tờ của cách trả tốt nhất hiện có thì không cần mở rộng các nghiệm của (x_1, x_2, \dots, x_k) nữa.

```

const      MAX      =20;
           fi        ='ATM.INP';
           fo        ='ATM.OUT';
type      vector    =array[1..MAX]of longint;
var        t,tmax    :array[1..MAX]of longint;
           x,xbest   :vector;
           c,cbest   :longint;
           n,s,sum   :longint;
procedure input;
var  f    :text;
     i    :longint;
begin
  assign(f,fi); reset(f);
  readln(f,n, s);
  for i:=1 to n do read(f,t[i]);
  close(f);
end;
procedure init;
var i :longint;
begin
  tmax[n]:=t[n];
  for i:=n-1 downto 1 do begin
    tmax[i]:=tmax[i+1];
    if tmax[i]<t[i] then tmax[i]:=t[i];
  end;
  sum:=0;
  c:=0;
  cbest:=n+1;
end;
procedure update;
var i    :longint;
     f    :text;
begin
  if (sum = s) and (c<cbest) then begin
    xbest:=x;
    cbest:=c;
  end;
end;
procedure printResult;

```

```

var i    :longint;
    f    :text;
begin
    assign(f,fo); rewrite(f);
    if cbest<n+1 then begin
        writeln(f,cbest);
        for i:=1 to n do
            if xbest[i]=1 then write(f,t[i],' ');
        end
        else write(f,'-1');
        close(f);
    end;
procedure branchBound(i:longint);
var j :longint;
begin
    if c + (s-sum)/tmax[i] >= cbest then exit;
    for j:=0 to 1 do begin
        x[i]:=j;
        sum:=sum + x[i]*t[i];
        c:=c + j;
        if (i=n) then update
        else if sum<=s then branchBound(i+1);
        sum:=sum - x[i]*t[i];
        c:=c - j;
    end;
end;
BEGIN
    input;
    init;
    branchBound(1);
    PrintResult;
END.

```

3. Tham ăn (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 ăn được sử dụng trong các trường hợp như vậy. Ưu điểm nổi bật của phương pháp tham ăn là độ phức tạp nhỏ, thường nhanh chóng tìm được lời giải.

3.1. Phương pháp

Giả sử nghiệm của bài toán có thể biểu diễn dưới dạng một vector (x_1, x_2, \dots, x_n) , mỗi thành phần x_i ($i = 1, 2, \dots, n$) được chọn ra từ tập S_i . Mỗi nghiệm của bài toán $X = (x_1, x_2, \dots, x_n)$, 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 ăn 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 $(x_1, x_2, \dots, x_{k-1})$ của nghiệm và khi mở rộng nghiệm ta sẽ chọn x_k “**tốt nhất**” trong các ứng cử viên trong tập S_k để được (x_1, x_2, \dots, x_k) . 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 ăn.

```
procedure Greedy;  
begin  
  X:= $\emptyset$ ;  
  i:=0;  
  while (chưa xây dựng xong hết thành phần của nghiệm) do  
    begin  
      i:=i+1;  
      <Xác định  $S_i$ >;  
      x←select( $S_i$ ) ; // chọn ứng cử viên tốt nhất trong tập  $S_i$   
    end;  
end;
```

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

Cần nhấn mạnh rằng, thuật toán tham ăn 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 ăn chỉ tìm được nghiệm gần đúng với nghiệm tối ưu.

3.2. Bài toán người du lịch

(Bài toán ở mục 2.2)

Có nhiều thuật toán tham ăn 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 $(x_1 = 1, x_2, \dots, x_n, x_{n+1} = 1)$, trong đó dãy (x_1, x_2, \dots, x_n) lập thành một hoán vị của $(1, 2, \dots, n)$.

+ Ta xây dựng nghiệm từng bước, bắt đầu từ $x_1=1$, chọn x_2 là thành phố gần x_1 nhất, sau đó chọn x_3 là thành phố gần x_2 nhất (x_3 khác x_1)... Tổng quát: chọn x_i là thành phố chưa đi qua mà gần x_{i-1} nhất. ($2 \leq i \leq n$).

```

program TSP;
const      MAX           =100;
           oo            =1000000;
           fi            ='TSP.INP';
           fo            ='TSP.OUT';
var         c             :array[1..MAX,1..MAX] of
longint;
           x             :array[1..MAX] of longint;
           d             :array[1..MAX] of longint;
           n             :longint;
           sum           :longint;
procedure input;
var f       :text;
    i,j,k   :longint;
begin
    assign(f,fi); reset(f);
    read(f,n);
    for i:=1 to n do
        for j:=1 to n do read(f,C[i,j]);
    close(f);
end;
procedure output;
var f       :text;
    i       :longint;
begin
    assign(f,fo); rewrite(f);
    writeln(f,sum);
    for i:=1 to n do write(f,x[i],'->');
    write(f,x[1]);

```

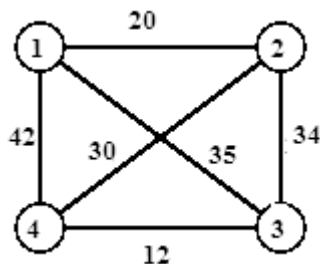


```

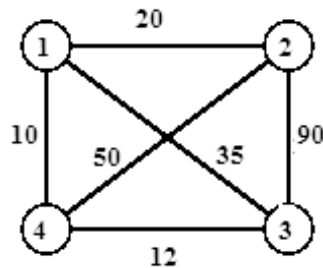
    close(f);
end;
procedure Greedy;
var i,j,xi :longint;
    best    :longint;
begin
    x[1]:=1;
    d[1]:=1;
    i:=1;
    while i<n do begin
        inc(i);
        // chọn ứng cử viên tốt nhất
        best:=oo;
        for j:=1 to n do
            if (d[j]=0) and (c[x[i-1],j]<best) then begin
                best:=c[x[i-1],j];
                xi:=j;
            end;
        x[i]:=xi; //ghi nhận thành phần nghiệm thứ i
        d[xi]:=1;
        sum:=sum+c[x[i-1],x[i]];
    end;
    sum:=sum+c[x[n],x[1]];
end;
BEGIN
    input;
    Greedy;
    output;
END.

```

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



3.3. Bài toán máy rút tiền tự động ATM

(bài toán ở mục 2.3)

Thuật toán với ý tưởng tham ăn 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.

```

const      MAX      =100;
           fi        ='ATM.INP';
           fo        ='ATM.OUT';

type       vector    =array[1..MAX]of longint;
var        t         :array[1..MAX]of longint;
           x         :vector;
           c         :longint;
           n,s       :longint;

procedure input;
var        f         :text;
           i         :longint;
begin
    assign(f,fi); reset(f);
    readln(f,n, s);
    for i:=1 to n do read(f,t[i]);
    close(f);
end;

procedure greedy;
var i,j       :longint;

```

```

    tmp    :longint;
begin
    fillchar(x, sizeof(x), 0);
    {sắp xếp các tờ theo giá trị giảm dần}
    for i:=1 to n-1 do
        for j:=i+1 to n do
            if t[i]<t[j] then begin
                tmp:=t[i];
                t[i]:=t[j];
                t[j]:=tmp;
            end;
        c:=0;
        for i:=1 to n do
            if s>=t[i] then
                begin
                    inc(c); {số lượng tờ lấy}
                    x[i]:=1; {tờ i được lấy}
                    s:=s-t[i];
                end;
        end;
procedure printResult;
var i    :longint;
    f    :text;
begin
    assign(f, fo); rewrite(f);
    if s=0 then begin
        writeln(f, c);
        for i:=1 to n do
            if x[i]=1 then write(f, t[i], ' ');
        end
        else write(f, '-1'); {nếu không lấy được đủ S, S>0}
        close(f);
    end;
BEGIN
    input;
    greedy;
    PrintResult;
END.

```

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

3.4. 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:

- p_i là thời gian cần thiết để hoàn thành công việc i .
- d_i 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 c_i là thời điểm hoàn thành công việc i . Khi đó, nếu $c_i > d_i$ ta nói công việc i bị hoàn thành trễ hạn, còn nếu $c_i \leq d_i$ 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).

Dữ liệu vào trong file “JS.INP” có dạng:

- Dòng đầu là số n ($n \leq 100$) 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

Kết quả file “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
p_i	6	3	5	7	2

d_i	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 J_s là tập gồm k công việc (mà cả k công việc này đều có thể thực hiện đúng hạn) và $\sigma = (i_1, i_2, \dots, i_k)$ là một hoán vị của các công việc trong J_s sao cho $d_{i_1} \leq d_{i_2} \leq \dots \leq d_{i_k}$ thì thứ tự σ là thứ tự để hoàn thành đúng hạn được cả k công việc.

Ví dụ: J_s 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 $\sigma = (5, 1, 3, 4)$ vì $d_5 = 3 \leq d_1 = 8 \leq d_3 = 15 \leq d_4 = 20$ để cả 4 công việc đều thực hiện đúng hạn.

Sử dụng chiến lược tham ăn, ta xây dựng tập công việc J_s theo từng bước, ban đầu $J_s = \emptyset$. Hàm chọn được xây dựng như sau: tại mỗi bước ta sẽ chọn công việc Job_i mà có thời gian thực hiện nhỏ nhất trong số các công việc còn lại cho vào tập J_s . Nếu sau khi kết nạp Job_i , các công việc trong tập J_s đều có thể thực hiện đúng hạn thì cố định việc kết nạp Job_i vào tập J_s , nếu không thì không kết nạp Job_i . Để đơn giản, ta giả sử rằng các công việc được đánh số theo thứ tự thời gian thực hiện tăng dần $p_1 \leq p_2 \leq \dots \leq p_n$. Ta có lược đồ thuật toán tham ăn như sau:

```

procedure JobScheduling;
begin
   $J_s := \emptyset$ ;
  for  $i := 1$  to  $n$  do

```

```

    if các công việc trong tập  $(Js \cup \{Job_i\})$  hoàn thành đúng
    hạn then
     $Js := Js \cup \{i\};$ 
    for  $i:=1$  to  $n$  do
        if  $Job_i \notin Js$  then  $Js \cup \{Job_i\};$ 
    end;

```

Sau đây là chương trình hoàn chỉnh:

```

const    MAX                =100;
         fi                 ='js.inp';
         fo                 ='js.out';
type     TJob               =record
                                p, d :longint;
                                name :longint;
                            end;
         TArrJobs           =array[1..MAX]of TJob;
var       jobs,Js           :TArrJobs;
         d                  :array[1..MAX]of longint;
         n,m                :longint;
procedure input;
var f     :text;
         i :longint;
begin
    assign(f,fi);
    reset(f);
    readln(f,n);
    for i:=1 to n do read(f,jobs[i].p);
    for i:=1 to n do read(f,jobs[i].d);
    close(f);
    for i:=1 to n do jobs[i].name:=i;
end;
procedure swap(var j1,j2:TJob);
var tmp :TJob;
begin
    tmp:=j1;
    j1:=j2;
    j2:=tmp;
end;
function check(var Js:TArrJobs; nJob:longint):boolean;

```

```

var i,j :longint;
    t    :longint;
begin
    for i:=1 to nJob-1 do
        for j:=i+1 to nJob do
            if Js[i].d>Js[j].d then swap(Js[i],Js[j]);
        t:=0;
        for i:=1 to nJob do begin
            if t+Js[i].p>Js[i].d then exit(false);
            t:=t+Js[i].p;
        end;
        exit(true);
    end;
procedure Greedy;
var i,j :longint;
    Js2 :TArrJobs;
begin
    for i:=1 to n-1 do
        for j:=i+1 to n do
            if      jobs[i].p      >      jobs[j].p      then
swap(jobs[i],jobs[j]);
        fillchar(d,sizeof(d),0);
        m:=0;
        for i:=1 to n do begin
            Js2:=Js;
            Js2[m+1]:=jobs[i];
            if check(Js2,m+1) then begin
                m:=m+1;
                Js:=Js2;
                d[i]:=1;
            end;
        end;
        //writeln(m);
        for i:=1 to n do
            if d[i]=0 then begin
                m:=m+1;
                Js[m]:=jobs[i];
            end;
    end;
end;

```

```

procedure printResult;
var f      :text;
    i      :longint;
begin
    assign(f,fo); rewrite(f);
    for i:=1 to n do write(f,Js[i].name,' ');
    close(f);
end;
BEGIN
    input;
    Greedy;
    printResult;
END.

```

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

4. Chia để trị (Divide and Conquer)

4.1. 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 dàng 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:

```

procedure DivideConquer(A,x); // tìm nghiệm x của bài toán A
begin
    if (A đủ nhỏ) then Solve(A)
    else begin
        Phân A thành các bài toán con  $A_1, A_2, \dots, A_m$ ;
        for i:=1 to m do DivideConquer( $A_i, x_i$ );
        Kết hợp các nghiệm  $x_i$  ( $i=1,2,\dots,m$ ) của các bài toán
        con  $A_i$  để nhận được nghiệm của bài toán A;
    end;
end;

```


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.

4.2. Bài toán tính a^n

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

Cách 1: Sử dụng thuật toán lặp, mất n phép nhân để tính a^n

```
procedure power(a,n:longint;var p:longint);  
  {giá trị  $a^n$  sẽ được lưu vào biến p}  
  var i : longint;  
  begin  
    p:=1;  
    for i:=1 to n do p:=p*a;  
  end;  
  var a, n, p : longint;  
  BEGIN  
    write('Nhap a, n:'); readln(a, n);  
    power(a,n,p);  
    write(p);  
  END.
```

Cách 2: Áp dụng kỹ thuật chia để trị, ta tính a^n dựa vào a^k (trong đó $k = n \text{ div } 2$) như sau:

- nếu n chẵn: $a^n = a^k \times a^k$
- nếu n lẻ: $a^n = a^k \times a^k \times a$

Để tính a^k ta lại dựa vào $a^{k \text{ div } 2}$, quá trình chia nhỏ cho đến khi nhận được bài toán tính a^1 thì dừng.

Ví dụ: tính 9^{13}

- bài toán được tính dựa trên bài toán con 9^6 , ta có $9^{13} = 9^6 \times 9^7$
- bài toán 9^6 được tính dựa trên bài toán con 9^3 , ta có $9^6 = 9^3 \times 9^3$
- bài toán 9^3 được tính dựa trên bài toán con 9^1 , ta có $9^{13} = 9^1 \times 9^1 \times 9^1$

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

```
procedure power(a,n:longint; var p:longint);  
  var tmp : longint;
```

```

begin
  if (n=1) then p:=a
  else begin
    power(a,n div 2,tmp);
    if (n mod 2=1) then p:=tmp*tmp*a
    else p:=tmp*tmp;
  end;
end;

```

hoặc viết dưới dạng hàm như sau:

```

function power(a,n:longint):longint;
var tmp : longint;
begin
  if (n=1) then exit(a)
  else begin
    tmp:=power(a,n div 2);
    if (n mod 2=1) then exit(tmp*tmp*a)
    else exit(tmp*tmp);
  end;
end;

```

Để đánh giá thời gian thực hiện thuật toán, ta tính số phép nhân phải sử dụng, gọi $T(n)$ là số phép nhân thực hiện, ta có:

$$\begin{cases} T(1) = 0 \\ T(n) \leq T\left(\frac{n}{2}\right) + 1 + 1 \text{ nếu } n > 1 \end{cases}$$

$$T(n) \leq T\left(\frac{n}{2}\right) + 2 \leq T\left(\frac{n}{2^2}\right) + 2 + 2 \leq \dots \leq 2\log n$$

Như vậy, thuật toán chia để trị mất không quá $2\log n$ phép nhân, nhỏ hơn rất nhiều so với n phép nhân.

4.3. Bài toán *Diff*

Bài toán: Cho mảng số nguyên $A[1..n]$, cần tìm $Diff(A[1..n]) = A[j] - A[i]$ đạt giá trị lớn nhất mà $1 \leq i \leq j \leq n$.

Ví dụ: mảng gồm 6 số 4, 2, 5, 8, 1, 7 thì độ lệch cần tìm là: 6

Cách 1: Thử tất cả các cặp chỉ số (i, j) , độ phức tạp $O(N^2)$

```

procedure find(var maxDiff:longint);
var i,j :longint;

```

```

begin
  maxDiff:=0;
  for i:=1 to n do
    for j:=i to n do
      if a[j]-a[i]>maxDiff then maxDiff:=a[j]-a[i];
    end;
  end;

```

Cách 2: Áp dụng kĩ thuật chia để trị, ta chia mảng $A[1..n]$ thành hai mảng con $A[1..k]$ và $A[(k+1)..n]$ trong đó $k = n \div 2$, ta có:

$$Diff(A[1..n]) = \begin{cases} Diff(A[1..k]) \\ Diff(A[(k+1)..n]) \\ MAX(A[(k+1)..n]) - MIN(A[1..k]) \end{cases}$$

Nếu tìm được độ lệch (*Diff*), giá trị lớn nhất (*MAX*) và giá trị nhỏ nhất (*MIN*) của hai mảng con $A[1..k]$ và $A[(k+1)..n]$, ta sẽ dễ dàng xác định được giá trị $Diff(A[1..n])$. Để tìm độ lệch, giá trị lớn nhất và giá trị nhỏ nhất của hai mảng con $A[1..k]$ và $A[(k+1)..n]$, ta lại tiếp tục chia đôi chúng. Quá trình phân nhỏ bài toán dừng lại khi ta nhận được bài toán mảng con chỉ có 1 phần tử. Từ phương pháp đã trình bày ở trên, ta xây dựng thủ tục đệ quy

`find2(l, r, maxDiff, maxValue, minValue)`

tìm giá trị độ lệch, giá trị lớn nhất, giá trị nhỏ nhất trên mảng $A[l..r]$ với $1 \leq l \leq r \leq n$.

```

const      MAXN      =100000;
           fi         =' ';
           fo         =' ';
var         a         :array[1..MAXN]of longint;
           n          :longint;
           maxdiff    :longint;
           tmp1,tmp2  :longint;

procedure      find2(l,r:longint;var
maxDiff,maxValue,minValue :longint);
var           mid          :longint;
           maxD1, maxV1, minV1 :longint;
           maxD2, maxV2, minV2 :longint;

begin
  if l=r then begin
    maxDiff:=0;
    maxValue:=a[r];

```

```

        minValue:=a[r];
    end
    else begin
        mid:=(l+r) div 2;
        find2(l, mid, maxD1, maxV1,minV1);
        find2(mid+1, r, maxD2, maxV2, minV2);
        maxDiff:=maxV2 - minV1;
        if maxDiff < maxD1 then maxDiff := maxD1;
        if maxDiff < maxD2 then maxDiff := maxD2;
        if maxV1 > maxV2 then
            maxValue:=maxV1 else maxValue:=maxV2;
            if minV1 < minV2 then
                minValue:=minV1 else minValue:=minV2;
            end;
        end;
    end;
end;
procedure input;
var f      :text;
    i      :longint;
begin
    assign(f,fi); reset(f);
    readln(f,n);
    for i:=1 to n do read(f,a[i]);
    close(f);
end;
BEGIN
    input;
    find2(1,n,maxDiff,tmp1,tmp2);
    writeln(maxDiff);
END.

```

Gọi $T(n)$ là số phép toán cần thực hiện trên mảng n phần tử $A[1..n]$, ta có:

$$T(n) = \begin{cases} 0 & \text{nếu } n = 1 \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \alpha & \text{nếu } n > 1 \end{cases}$$

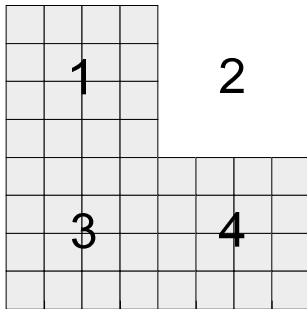
Giả sử, $n = 2^k$, bằng phương pháp thế ta có:

$$\begin{aligned}
 T(n) &= T(2^k) = 2T(2^{k-1}) + \alpha = 2(2T(2^{k-2}) + \alpha) + \alpha \\
 &= 2^2T(2^{k-2}) + 2\alpha + \alpha = \dots = 2^3T(2^{k-3}) + 2^2 + 2 + 1 \\
 &= 2^kT(1) + 2^{k-1}\alpha + \dots + 2\alpha + \alpha = (2^k - 1)\alpha = (n - 1)\alpha
 \end{aligned}$$

Độ phức tạp thuật toán là: $O(N)$

4.4. Lát nền

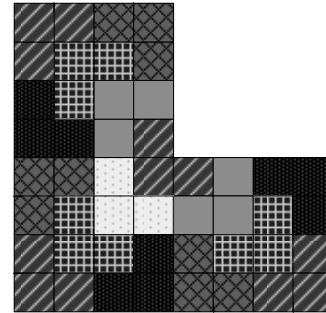
Hãy lát nền nhà hình vuông cạnh $n = 2^k$ ($2 \leq k \leq 10$) bị khuyết một phần tư tại góc trên phải (khuyết phần 2) bằng những viên gạch hình thước thợ tạo bởi 3 ô vuông đơn vị.



Nền nhà ($k = 3$)



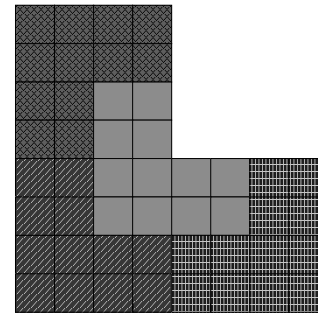
Gạch hình thước thợ



Một cách lát nền

Giải. Ta chia nền nhà thành 4 phần (hình bên phải), mỗi phần có hình dạng giống với hình ban đầu nhưng có cạnh giảm đi một nửa. Như vậy, nếu có thể lát nền với kích thước 2^k thì ta hoàn toàn có thể lát nền với kích thước 2^{k+1} .

Thủ tục đệ quy $\text{cover}(x, y, s, t)$ dưới đây sẽ lát nền có kích thước s , bị khuyết phần t ($t = 1, 2, 3, 4$) có tọa độ trái trên là (x, y) .



```
const    MAXSIZE =1 shl 10;
var      a        :array[1..MAXSIZE,1..MAXSIZE]of longint;
          count,k :longint;
procedure cover(x,y,s,t:longint);
begin
  if s = 2 then begin
    inc(count);
    if t<>1 then a[x,y]:=count;
    if t<>2 then a[x,y+1]:=count;
    if t<>3 then a[x+1,y]:=count;
    if t<>4 then a[x+1,y+1]:=count;
    exit;
  end;
  if t=1 then begin
    cover(x,y,s/2,t);
    cover(x,y,s/2,4);
    cover(x,y+s/2,t);
    cover(x,y+s/2,4);
  end;
  if t=2 then begin
    cover(x,y,s/2,4);
    cover(x,y,s/2,t);
    cover(x,y+s/2,4);
    cover(x,y+s/2,t);
  end;
  if t=3 then begin
    cover(x,y,s/2,t);
    cover(x,y,s/2,4);
    cover(x,y+s/2,t);
    cover(x,y+s/2,4);
  end;
  if t=4 then begin
    cover(x,y,s/2,t);
    cover(x,y,s/2,4);
    cover(x,y+s/2,t);
    cover(x,y+s/2,4);
  end;
end;
```

```

end;
if t=1 then begin
    cover(x,y+s div 2,s div 2,3);
    cover(x+s div 2,y,s div 2,2);
    cover(x+s div 2,y+s div 2,s div 2,1);
    cover(x+s div 4,y+s div 4,s div 2,1);
end;
if t=2 then begin
    cover(x,y,s div 2,4);
    cover(x+s div 2,y,s div 2,2);
    cover(x+s div 2,y+s div 2,s div 2,1);
    cover(x+s div 4,y+s div 4,s div 2,2);
end;
if t=3 then begin
    cover(x,y,s div 2,4);
    cover(x,y+s div 2,s div 2,3);
    cover(x+s div 2,y+s div 2,s div 2,1);
    cover(x+s div 4,y+s div 4,s div 2,3);
end;
if t=4 then begin
    cover(x,y,s div 2,4);
    cover(x+s div 2,y,s div 2,2);
    cover(x,y+s div 2,s div 2,3);
    cover(x+s div 4,y+s div 4,s div 2,4);
end;
end;
procedure output;
var i,j :longint;
    f    :text;
begin
    assign(f,'cover.out'); rewrite(f);
    for i:=1 to 1 shl k do
        begin
            for j:=1 to 1 shl k do write(f,a[i,j], ' ');
            writeln(f);
        end;
end;

```

```

close(f);
end;
BEGIN
  write('k=');readln(k);
  cover(1,1,1 shl k,2);
  output;
END.

```

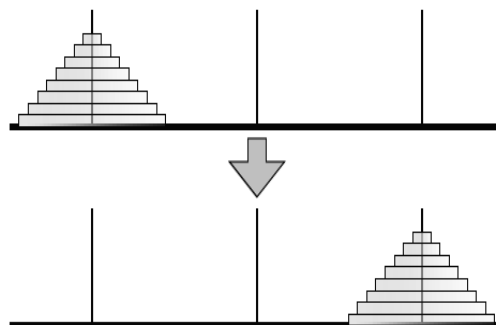
Ví dụ về Input / Output của chương trình:

k = 3	1 1 3 3 0 0 0 0
	1 4 4 3 0 0 0 0
	2 4 13 13 0 0 0 0
	2 2 13 16 0 0 0 0
	5 5 14 16 16 15 9 9
	5 8 14 14 15 15 12 9
	6 8 8 7 10 12 12 11
	6 6 7 7 10 10 11 11

4.5. Tháp Hà Nội

Cho 3 cái cọc và n đĩa có kích thước khác nhau. Ban đầu cả n đĩa đều ở cọc 1 và được xếp theo thứ tự đĩa to ở dưới, đĩa nhỏ ở trên. Hãy di chuyển cả n đĩa từ cọc 1 sang cọc 3 theo quy tắc sau:

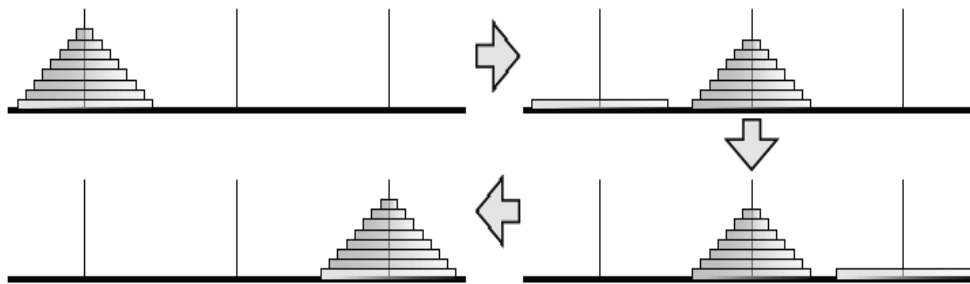
- Một lần chỉ được chuyển một đĩa
- Trong quá trình chuyển đĩa, có thể sử dụng cọc 2 làm cọc trung gian và một đĩa chỉ được đặt lên một đĩa lớn hơn.



Giải

Để chuyển n đĩa từ cọc 1 sang cọc 3 ta sẽ thực hiện như sau:

- chuyển $(n - 1)$ đĩa từ cọc 1 sang cọc 2, sử dụng cọc 3 làm cọc trung gian
- chuyển 1 đĩa từ cọc 1 sang cọc 3
- chuyển $(n - 1)$ đĩa từ cọc 2 sang cọc 3, sử dụng cọc 1 làm cọc trung gian



```

procedure move(n :longint;src,dst,tmp:longint);
begin
  if n = 1 then writeln('move ',src,' ',dst)
  else begin
    move(n-1,src,tmp,dst);
    move(1,src,dst,tmp);
    move(n-1,tmp,dst,src);
  end;
end;

```

4.6. Bài toán sắp xếp mảng bằng thuật toán trộn (Merge Sort)

Bài toán: Cho mảng số nguyên $A[1..n]$, cần sắp xếp các phần tử của mảng theo thứ tự tăng dần.

Giải: Ta chia mảng $A[1..n]$ thành hai mảng con $A[1..k]$ và $A[(k+1)..n]$ trong đó $k = n \text{ div } 2$. Giả sử, hai mảng con $A[1..k]$ và $A[(k+1)..n]$ đã được sắp xếp tăng dần, ta sẽ trộn hai mảng con để được mảng $A[1..n]$ cũng sắp xếp tăng dần. Để sắp xếp hai mảng con $A[1..k]$ và $A[(k+1)..n]$ ta lại tiếp tục chia đôi chúng. Thủ tục đệ quy MergeSort(i,j) sắp xếp tăng dần mảng con $A[i..j]$ với $1 \leq i \leq j \leq n$. Để sắp xếp cả mảng $A[1..n]$, ta chỉ cần gọi thủ tục này với $i = 1, j = n$.

```

procedure MergeSort(i,j:longint);
var k : longint;
begin
  if (i<j) then begin
    k:=(i+j) div 2;
    MergeSort(i,k);
    MergeSort(k+1,j);
    Merge(i,k,j);
    {thủ tục Merge(i,k,j) trộn hai mảng con A[i..k], A[(k+1)..j] đã được
    sắp xếp thành mảng A[i..j] cũng được sắp xếp}
  end;
end;

```


Việc trộn hai mảng con đã được sắp xếp $A[i..k]$ và $A[(k+1)..j]$ thành mảng $A[i..j]$ cũng được sắp xếp có thể thực hiện trong thời gian $O(j-i+1)$, là bài tập 3.8 ở chuyên đề Sắp xếp. Thuật toán MergeSort có độ phức tạp là $O(n \log n)$.

5. Quy hoạch động (Dynamic programming)

5.1. 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.

5.2. Số Fibonacci

Số Fibonacci được xác định bởi công thức:

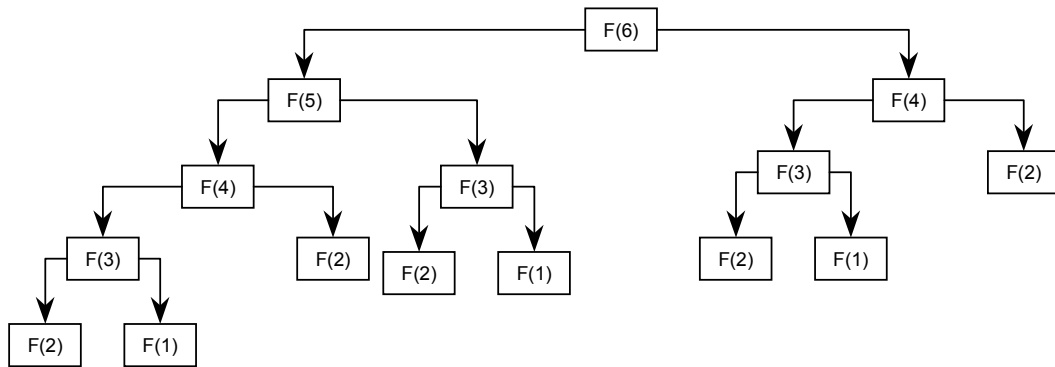
$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \text{ với } n \geq 2 \end{cases}$$

Hãy xác định số Fibonacci thứ n .

Cách 1: Áp dụng phương pháp chia để trị, ta tính F_n dựa vào F_{n-1} và F_{n-2} .

```
function F(n: longint): int64;
begin
    if n <= 1 then F := n
    else F := F(i - 1) + F(i - 2);
end;
BEGIN
    readln(n);
    Writeln(F(n));
END.
```

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 $F(1)$.



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 .

```
const           MaxN           =50;
var             S               :array[0..MaxN]of int64;
               n,k             :longint;

function F(n:longint):int64;
begin
    if S[n]=-1 then
begin
{bài toán chưa được giải thì sẽ tiến hành giải}
        if n<=1 then S[n]:=n
        else S[n]:=F(n-1) + F(n-2);
end;
{nếu bài toán đã được giải thì không cần giải nữa mà lấy luôn kết quả}
end;
```

```

    F:=S[n];
end;
BEGIN
    readln(n);
    for k:=0 to MaxN do S[k]:=-1;
    writeln(F(n));
END.

```

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:

```

const      maxN      =50;
var        S          : array[0..maxN] of Int64;
           i           : longint;

BEGIN
    readln(n);
    S[0] := 1; S[1] := 1;
    for i := 2 to n do
        S[i] := S[i - 1] + S[i - 2];
    Writeln(S[n]);
END.

```

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.

5.3. Dãy con đơn điệu tăng dài nhất

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

Yêu cầu: Tìm 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: $a_0 = -\infty$ và $a_{n+1} = +\infty$. Khi đó dãy con đơn điệu tăng dài nhất chắc chắn sẽ bắt đầu từ a_0 và kết thúc ở a_{n+1} .

Với $\forall i: 0 \leq i \leq n+1$. Ta sẽ tính $L[i]$ = độ dài dãy con đơn điệu tăng dài nhất bắt đầu tại a_i .

1. Bài toán nhỏ nhất

$L[n + 1]$ = Độ dài dãy con đơn điệu tăng dài nhất bắt đầu tại $a_{n+1} = +\infty$. Dãy con này chỉ gồm mỗi một phần tử $(+\infty)$ nên $L[n + 1] = 1$.

2. 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 a_i . $L[i]$ 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 a_i sẽ được thành lập bằng cách lấy a_i ghép vào đầu một trong số những dãy con đơn điệu tăng dài nhất bắt đầu tại vị trí a_j đứng sau a_i .

Ta sẽ chọn dãy nào để ghép a_i vào đầu? Tất nhiên là chỉ được ghép a_i vào đầu những dãy con bắt đầu tại a_j nào đó lớn hơn a_i (để đảm bảo tính tăng) và dĩ nhiên ta sẽ chọn dãy dài nhất để ghép a_i 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 $a_j > a_i$, chọn ra chỉ số j_{\max} có $L[j_{\max}]$ lớn nhất. Đặt $L[i] := L[j_{\max}] + 1$.

3. Truy vết

Tại bước xây dựng dãy L , mỗi khi tính $L[i] := L[j_{\max}] + 1$, ta đặt $T[i] = j_{\max}$.

Để lưu lại rằng: Dãy con dài nhất bắt đầu tại a_i sẽ có phần tử thứ hai kế tiếp là $a_{j_{\max}}$. 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 do
{Chúng ta chưa duyệt đến số  $a_{n+1}=+\infty$  ở cuối}
begin
  <Thông báo chọn  $a_i$ >
  i := T[i];
end;
```

Ví dụ: với $A = (5, 2, 3, 4, 9, 10, 5, 6, 7, 8)$.

Hai dãy Length và Trace sau khi tính sẽ là:

i	0	1	2	3	4	5	6	7	8	9	10	11
a_i	$-\infty$	5	2	3	4	9	10	5	6	7	8	$+\infty$
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 $\longrightarrow \longrightarrow \longrightarrow \longrightarrow \longrightarrow \longrightarrow \longrightarrow \longrightarrow \longrightarrow \longrightarrow \longrightarrow \longrightarrow$

```

Const    max = 1000;
var
  a, L, T: array[0..max + 1] of longint;
  n: longint;
procedure Enter;      {Nhập dữ liệu}
var
  i: longint;
begin
  Write('n = '); Readln(n);
  for i := 1 to n do
    begin
      Write('a[', i, '] = '); Readln(a[i]);
    end;
end;
procedure Optimize;   {Quy hoạch động}
var
  i, j, jmax: longint;
begin
  a[0] := -32768; a[n + 1] := 32767;  {Thêm hai phần tử cạnh hai đầu dãy a}
  L[n + 1] := 1;      {Điền cơ sở quy hoạch động vào bảng phương án}
  for i := n downto 0 do
    begin
      {Chọn trong các chỉ số j đứng sau i thoả mãn  $a_j > a_i$  ra chỉ số jmax có L[jmax] lớn nhất}
      jmax := n + 1;
      for j := i + 1 to n + 1 do
        if (a[j] > a[i]) and (L[j] > L[jmax]) then jmax := j;
      L[i] := L[jmax] + 1; {Lưu độ dài dãy con tăng dài nhất bắt đầu tại ai}
      T[i] := jmax;      {Lưu vết: phần tử đứng liền sau  $a_i$  trong dãy con tăng dài nhất đó là  $a_{jmax}$ }
    end;
end;

```

```

Writeln('Length of result : ', L[0] - 2);{Chiều dài dãy con
tăng dài nhất}
i := T[0];      {Bắt đầu truy vết tìm nghiệm}
while i <> n + 1 do
begin
    Writeln('a[', i, '] = ', a[i]);
    i := T[i];
end;
end;
begin
    Enter;
    Optimize;
end.

```

5.4. Dãy con chung dài nhất

Cho hai số nguyên dương M, N ($0 < M, N \leq 100$) và hai dãy số nguyên: A_1, A_2, \dots, A_M và B_1, B_2, \dots, B_N . 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.

Dữ liệu vào trong file *LCS.INP* có dạng:

- + Dòng thứ nhất chứa M số A_1, A_2, \dots, A_M
- + Dòng thứ hai chứa N số B_1, B_2, \dots, B_N .

Dữ liệu ra trong file *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 con 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 $L[0, j] = 0 \forall j, j = 1..N$, $L[i, 0] = 0 \forall i, i = 1..M$. Với $M \geq i > 0$ và $N \geq j > 0$ thì $L[i, j]$ được tính theo công thức truy hồi sau:

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

(với $x = 0$ nếu $A[i] \neq B[j]$, $x=1$ nếu $A[i]=B[j]$)

```

const  fi    = 'LCS.INP';
       fo    = 'LCS.OUT';
       MaxMN = 100;
var    f      : text;
       a,b    : array[0..MaxMN] of longint;
       l      : array[0..MaxMN,0..MaxMN] of longint;
       m,n    : longint;
       p      : array[0..MaxMN] of longint;
       count  : longint;
procedure Enter;
var f : text;
begin
    m := 0;
    n := 0;
    assign(f,fi);
    reset(f);
    while not eoln(f) do
    begin
        inc(m);
        read(f,a[m]);
    end;
    readln(f);
    while not eoln(f) do
    begin
        inc(n);
        read(f,b[n]);
    end;
    close(f);
end;
function max(x,y : longint) : longint;
begin
    if x>y then max := y
    else max := y;
end;
procedure Optimize;
var i,j : longint;
begin
    for i:=1 to m do l[i,0] := 0;
    for j:=1 to n do l[0,j] := 0;
    for i:=1 to m do
        for j:=1 to n do
            begin

```

```

        if a[i]=b[j] then l[i,j] := l[i-1,j-1] + 1
        else l[i,j] := max(l[i,j-1],l[i-1,j]);
    end;
end;
procedure Trace;
var    f    : text;
        i,j : longint;
begin
    assign(f,fo);
    rewrite(f);
    writeln(f,l[m,n]);
    i := m;
    j := n;
    fillchar(p,sizeof(p),0);
    count:= 0;
    while (i>0) and (j>0) do
    begin
        if a[i]=b[j] then
        begin
            inc(count);
            p[count] := a[i];
            dec(i);
            dec(j);
        end
        else if l[i,j]=l[i,j-1] then dec(j)
        else dec(i);
    end;
    for i:=count downto 1 do write(f,p[i],' ');
    close(f);
end;
BEGIN
    Enter;
    Optimize;
    Trace;
END.

```

5.5. Bài toán cái túi

Trong siêu thị có n gói hàng ($n \leq 100$), gói hàng thứ i có trọng lượng là $W_i \leq 100$ và trị giá $V_i \leq 100$. Một tên trộm đột nhập vào siêu thị, sức của tên trộm không thể mang được trọng lượng vượt quá M ($M \leq 100$). 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 $\{1, 2, \dots, i\}$ 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 $B[n, M]$.

1. Công thức tính $B[i, j]$.

Với giới hạn trọng lượng j , việc chọn tối ưu trong số các gói $\{1, 2, \dots, i-1, i\}$ 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 $\{1, 2, \dots, i-1\}$ 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 $W_i \leq j$) thì $B[i, j]$ bằng giá trị gói thứ i là V_i cộng với giá trị lớn nhất có thể có được bằng cách chọn trong số các gói $\{1, 2, \dots, i-1\}$ với giới hạn trọng lượng $j - W_i$. Tức là về mặt giá trị thu được: $B[i, j] = V_i + B[i-1, j - W_i]$

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.


2. Cơ sở quy hoạch động:

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

3. 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				



4. 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 $b[n, M] \neq b[n - 1, M]$ thì ta thông báo rằng phép chọn tối ưu có chọn gói thứ n và truy tiếp $b[n - 1, M - W_n]$. Cứ tiếp tục cho tới khi truy lên tới hàng 0 của bảng phương án.

```
const
    max          = 100;
var
    W, V         : array[1..max] of longint;
    B            : array[0..max, 0..max] of longint;
    n, M         : longint;
procedure Enter;
var
    i: longint;
begin
    Write('n = '); Readln(n);
    for i := 1 to n do
        begin
            Writeln('Pack ', i);
            Write('  + Weight : '); Readln(W[i]);
            Write('  + Value  : '); Readln(V[i]);
        end;
    Write('M = '); Readln(M);
end;
procedure Optimize;
var
    i, j: longint;
begin
    FillChar(B[0], SizeOf(B[0]), 0);
    for i := 1 to n do
        for j := 0 to M do
            begin
                B[i, j] := B[i - 1, j];
                if (j >= W[i]) and (B[i, j] < B[i-1, j-W[i]] + V[i])
then
                    B[i, j] := B[i - 1, j - W[i]] + V[i];
            end;
        end;
    end;
```

```

end;
procedure Trace;
begin
  Writeln('Max Value : ', B[n, M]);
  Writeln('Selected Packs: ');
  while n <> 0 do
    begin
      if B[n, M] <> B[n - 1, M] then
        begin
          Writeln('Pack ', n, ' W = ', W[n], ' Value = ', V[n]);
          M := M - W[n];
        end;
      Dec(n);
    end;
  end;
BEGIN
  Enter;
  Optimize;
  Trace;
END.

```

Bài tập

- 4.1.** Cho danh sách tên của n ($n \leq 10$) học sinh (các tên đôi một khác nhau) và một số nguyên dương k ($k \leq n$). Hãy liệt kê tất cả các cách chọn k học sinh trong n học sinh.

Ví dụ:

Dữ liệu vào	Kết quả ra
$n = 4, k = 2$, danh sách tên học sinh như sau: An Binh Hong Minh	Có 6 cách chọn 2 học sinh trong 4 học sinh: 1. An Binh 2. An Hong 3. An Minh 4. Binh Hong 5. Binh Minh 6. Hong Minh

- 4.2. Một dãy nhị phân độ dài n ($n \leq 10$) là một dãy $x = x_1x_2 \dots x_n$ trong đó $x_i \in \{0,1\}, i = 1,2,\dots,n$. Hãy liệt kê tất cả các dãy nhị phân độ dài n

Dữ liệu vào	Kết quả ra
$n = 3$	<p>Có 8 dãy nhị phân độ dài 3</p> <p>1. 000</p> <p>2. 001</p> <p>3. 010</p> <p>4. 011</p> <p>5. 100</p> <p>6. 101</p> <p>7. 110</p> <p>8. 111</p>

- 4.3. Cho xâu S (độ dài không vượt quá 10) chỉ gồm các kí tự 'A' đến 'Z' (các kí tự trong xâu S đôi một khác nhau). Hãy liệt kê tất cả các hoán vị khác nhau của xâu S .

Dữ liệu vào	Kết quả ra
$S = \text{'XYZ'}$	<p>Có 6 hoán vị khác nhau của 'XYZ'</p> <p>1. XYZ</p> <p>2. XZY</p> <p>3. YXZ</p> <p>4. YZX</p> <p>5. ZXY</p> <p>6. ZYX</p>

- 4.4. Cho số nguyên dương n ($n \leq 20$), hãy liệt kê tất cả các xâu độ dài n chỉ gồm 2 kí tự 'A' hoặc 'B' mà không có 2 kí tự 'B' nào đứng cạnh nhau.

Dữ liệu vào	Kết quả ra
$n = 4$	<p>Có 8 xâu độ dài 4</p> <p>1. AAAA</p> <p>2. AAAB</p> <p>3. AABA</p> <p>4. ABAA</p> <p>5. ABAB</p>

	6. BAAA
	7. BAAB
	8. BABA

- 4.5. Cho dãy số A gồm N ($N \leq 10$) số nguyên a_1, a_2, \dots, a_N và một số nguyên dương K ($1 < K < N$). Hãy đưa ra một cách chia dãy số thành K nhóm mà các nhóm có tổng bằng nhau.

Dữ liệu vào	Kết quả ra
$N=5, S=3$	nhóm 1: 4, 6
Dãy số a:	nhóm 2: 1, 9
1, 4, 6, 9, 10	nhóm 3: 10

- 4.6. Một chuỗi $X = x_1x_2..x_M$ được gọi là chuỗi con của chuỗi $Y = y_1y_2..y_N$ nếu ta có thể nhận được chuỗi X từ chuỗi Y bằng cách xoá đi một số ký tự, tức là tồn tại một dãy các chỉ số:

$$1 \leq i_1 < i_2 < \dots < i_M \leq N \quad \text{để} \quad x_1 = y_{i_1}, x_2 = y_{i_2}, \dots, x_M = y_{i_M}$$

Ví dụ: $X='adz'$ là chuỗi con của chuỗi $Y='baczdtz'$; $i_1 = 2 < i_2 = 5 < i_3 = 7$.

Nhập vào một chuỗi S (độ dài không quá 15, chỉ gồm các ký tự 'a' đến 'z'), hãy liệt kê tất cả các chuỗi con khác nhau của chuỗi S .

Dữ liệu vào	Kết quả ra
$S='aba'$	Có 6 chuỗi con khác nhau của 'aba' 1. a 2. b 3. aa 4. ab 5. ba 6. aba

- 4.7. Cho số nguyên dương n ($n \leq 10$), liệt kê tất cả các cách khác nhau đặt n dấu ngoặc mở và n dấu ngoặc đóng đúng đắn?

Dữ liệu vào	Kết quả ra
$n = 3$	Có 5 cách $((())), (()()), (())(), ()(()), ()(())$

- 4.8. Cho n ($n \leq 10$) số nguyên dương a_1, a_2, \dots, a_n ($a_i \leq 10^9$). Tìm số nguyên dương m nhỏ nhất sao cho m không phân tích được dưới dạng tổng của một số các số (mỗi số sử dụng không quá một lần) thuộc n số trên.

Dữ liệu vào	Kết quả ra
n=4 Dãy số a: 1, 2, 3, 6	13

- 4.9. Cho chuỗi S (độ dài không vượt quá 10) chỉ gồm các ký tự 'A' đến 'Z' (các ký tự trong chuỗi S không nhất thiết phải khác nhau). Hãy liệt kê tất cả các hoán vị khác nhau của chuỗi S .

Dữ liệu vào	Kết quả ra
S='ABA'	Có 3 hoán vị khác nhau của 'ABA' 1. AAB 2. ABA 3. BAA

4.10. Bài toán mã đi tuần

Cho bàn cờ $n \times n$ ô, tìm cách di chuyển một quân mã (mã di chuyển theo luật cờ vua) trên bàn cờ xuất phát từ ô (1,1) đi qua tất cả các ô, mỗi ô qua đúng một lần.

Ví dụ: N=5

1	24	13	18	7
14	19	8	23	12
9	2	25	6	17
20	15	4	11	22
3	10	21	16	5

- 4.11. Số siêu nguyên tố là số nguyên tố mà khi bỏ một số tùy ý các chữ số bên phải của nó thì phần còn lại vẫn tạo thành một số nguyên tố.

Ví dụ: 2333 là một số siêu nguyên tố có 4 chữ số vì 233, 23, 2 cũng là các số nguyên tố.

Cho số nguyên dương N ($0 < N < 10$), đưa ra các số siêu nguyên tố có N chữ số cùng số lượng của chúng.

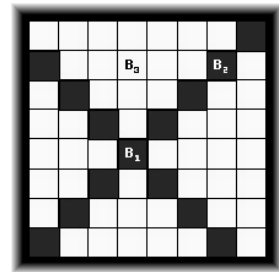
Ví dụ: Với $N=4$

Có 16 số: 2333 2339 2393 2399 2939 3119 3137 3733 3739 3793 3797 5939 7193 7331 7333 7393

- 4.12.** Cho một xâu S (chỉ gồm các kí tự '0' đến '9', độ dài nhỏ hơn 10) và số nguyên M , hãy đưa ra một cách chèn vào S các dấu '+' hoặc '-' để thu được số M cho trước (nếu có thể).

Ví dụ: $M = 8$, $S = '123456789'$ một cách chèn: $'-1+2-3+4+5-6+7'$;

- 4.13.** Trong cờ vua quân tượng chỉ có thể di chuyển theo đường chéo và hai quân tượng có thể chiếu nhau nếu chúng nằm trên đường di chuyển của nhau. Trong hình bên, hình vuông tô đậm thể hiện các vị trí mà quân tượng B_1 có thể đi tới được, quân tượng B_1 và B_2 chiếu nhau, quân B_1 và B_3 không chiếu nhau. Cho kích thước N của bàn cờ và K quân tượng, hỏi có bao nhiêu cách đặt các quân tượng vào bàn cờ mà các quân tượng không chiếu nhau.



Dữ liệu vào trong file: “bishops.inp” có dạng:

- Dòng đầu là số t là số test ($t \leq 10$)
- t dòng sau mỗi dòng chứa 2 số nguyên dương N, K ($2 \leq N \leq 10, 0 < K \leq N^2$)

Kết quả ra file: “bishops.out” gồm t dòng, mỗi chứa một số duy nhất là số cách đặt các quân tượng vào bàn cờ tương ứng với dữ liệu vào.

- 4.14.** N -mino là hình thu được từ N hình vuông 1×1 ghép lại (cạnh kề cạnh). Hai n -mino được gọi là đồng nhất nếu chúng có thể đặt chồng khít lên nhau. Cho số nguyên dương N ($1 < N < 8$), tính và vẽ ra tất cả các N -mino trên màn hình.

Ví dụ: Với $N=3$ chỉ có hai loại N -mino sau đây:



3-mino thẳng

3-mino hình thước thợ

- 4.15.** Trong mục 2.2, lời giải bài toán TSP là một giải pháp nhánh cận rất thô sơ. Hãy thử chạy chương trình với trường hợp như sau: số thành phố $n = 20$,

khoảng cách giữa các thành phố bằng 1 (nghĩa là $C[i, j] = 1$ với $i \neq j$). Hãy rút ra nhận xét và có thể đánh giá nhánh cận chặt hơn nữa làm tăng hiệu quả của chương trình.

4.16. Cho bàn cờ quốc tế 8×8 ô, mỗi ô ghi một số nguyên dương không vượt quá 32000.

Yêu cầu: Xếp 8 quân hậu lên bàn cờ sao cho không quân nào không chế được quân nào và tổng các số ghi trên các ô mà quân hậu đứng là lớn nhất.

Dữ liệu vào: gồm 8 dòng, mỗi dòng ghi 8 số nguyên dương, giữa các số cách nhau một dấu cách.

Kết quả ra: một số duy nhất là đáp số của bài toán.

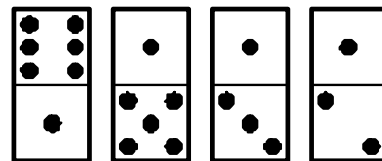
Dữ liệu vào	Kết quả ra
1 2 4 9 3 2 1 4	66
6 9 5 4 2 3 1 4	
3 6 2 3 4 1 8 3	
2 3 7 3 2 1 4 2	
1 2 3 2 3 9 2 1	
2 1 3 4 2 4 2 8	
2 1 3 2 8 4 2 1	
8 2 3 4 2 3 1 2	

4.17. Một chiếc ba lô có thể chứa được một khối lượng w . Có n ($n \leq 20$) đồ vật được đánh số $1, 2, \dots, n$. Đồ vật i có khối lượng a_i và có giá trị c_i . Cần chọn các đồ vật cho vào ba lô để tổng giá trị các đồ vật là lớn nhất.

4.18. Dominoes

Có N quân Domino xếp thành một hàng như hình vẽ

Mỗi quân Domino được chia làm hai phần, phần trên và phần dưới. Trên mặt mỗi phần có từ 1 đến 6 dấu chấm.



Ta nhận thấy rằng:

Tổng số dấu chấm ở phần trên của N quân Domino bằng: $6+1+1+1=9$, tổng số dấu chấm ở phần dưới của N quân Domino bằng $1+5+3+2=11$, độ chênh lệch giữa tổng trên và tổng dưới bằng $|9-11|=2$

Với mỗi quân, bạn có thể quay 180° để phần trên trở thành phần dưới, phần dưới trở thành phần trên, và khi đó độ chênh lệch có thể được thay đổi. Ví dụ như ta quay quân Domino cuối cùng của hình trên thì độ chênh lệch bằng 0

Bài toán đặt ra là: Cần quay ít nhất bao nhiêu quân Domino nhất để độ chênh lệch giữa phần trên và phần dưới là nhỏ nhất.

Dữ liệu vào trong file: “DOMINO.INP” có dạng:

- Dòng đầu là số nguyên dương N ($1 \leq N \leq 20$)
- N dòng sau, mỗi dòng hai số a_i, b_i là số dấu chấm ở phần trên, số dấu chấm ở phần dưới của quân Domino thứ i ($1 \leq a_i, b_i \leq 6$)

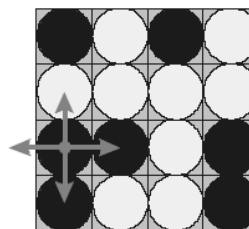
Kết quả ra file: “DOMINO.OUT” có dạng: Gồm 1 dòng duy nhất chứa 2 số nguyên cách nhau một dấu cách là độ chênh lệch nhỏ nhất và số quân Domino cần quay ít nhất để được độ chênh lệch đó.

4.19. Cho một lưới $M \times N$ ($M, N \leq 10$) ô, mỗi ô đặt một bóng đèn bật hoặc tắt. Trên mỗi dòng và mỗi cột có một công tắc. Nếu tác động vào công tắc dòng i ($i=1..M$) hoặc công tắc cột j ($j=1..N$) thì tất cả các bóng đèn trên dòng i hoặc cột j sẽ thay đổi trạng thái. Hãy tìm cách tác động vào các công tắc để được nhiều đèn sáng nhất.

4.20. Có 16 đồng xu xếp thành bảng 4×4 , mỗi đồng xu có thể úp hoặc ngửa như hình vẽ sau:

Màu đen thể hiện đồng xu úp, màu trắng thể hiện đồng xu ngửa.

Tại mỗi bước ta có phép biến đổi sau: Chọn một đồng xu và thay đổi trạng thái của đồng xu đó và tất cả các đồng xu nằm ở các ô chung cạnh (úp thành ngửa, ngửa thành úp). Cho trước một trạng thái các đồng xu, hãy lập trình tìm số phép biến đổi ít nhất để đưa về trạng thái tất cả các đồng xu hoặc đều úp hoặc đều ngửa.



Dữ liệu vào trong file “COIN.INP” có dạng: Gồm 4 dòng, mỗi dòng 4 ký tự 'w' - mô tả trạng thái ngửa hoặc 'b' - mô tả trạng thái úp.

Kết quả ra file “COIN.OUT” có dạng: Nếu có thể biến đổi được ghi số phép biến đổi ít nhất nếu không ghi “Impossible”

COIN.INP	COIN.OUT	COIN.INP	COIN.OUT
bwbw www	Impossible	bwwb bbwb	4

bbwb		bwwb	
bwwb		bwww	

4.21. Có N file chương trình với dung lượng S_1, S_2, \dots, S_n và loại đĩa CD có dung lượng D . Hỏi cần ít nhất bao nhiêu đĩa CD để có thể copy đủ tất cả các file chương trình (một file chương trình chỉ nằm trong một đĩa CD).

a) Giải bài toán bằng phương pháp nhánh cận với $N \leq 10$.

b) Giải bài toán bằng một thuật toán tham ăn với $N \leq 100$.

Dữ liệu vào	Kết quả ra
$N=5, D=700$ 320, 100, 300, 560, 50	Cần ít nhất 2 đĩa CD Đĩa 1: 320, 300, 50 Đĩa 2: 100, 560

4.22. Chương trình giải bài toán lập lịch giảm thiểu trễ hạn (ở mục 3.4) có độ phức tạp $O(N^3)$, hãy cải tiến hàm check để nhận được chương trình với độ phức tạp $O(N^2)$.

4.23. Cho một chuỗi S (độ dài không quá 200) chỉ gồm 3 loại ký tự ' A ', ' B ', ' C '. Ta có phép đổi chỗ hai ký tự bất kỳ trong chuỗi, hãy tìm cách biến đổi ít bước nhất để được chuỗi theo thứ tự tăng dần.

Dữ liệu vào	Kết quả ra
$S = \text{'CBABA'}$	Cần ít nhất 2 phép biến đổi CBABA \rightarrow ABABC \rightarrow AABBC

4.24. Cho N ($N \leq 1000$) đoạn số nguyên $[a_i, b_i]$, hãy chọn một tập gồm ít số nhất mà mỗi đoạn số nguyên trên đều có ít nhất 2 số thuộc tập.

$$(|a_i|, |b_i| \leq 10^9)$$

Ví dụ: có 5 đoạn $[0,10], [2,3], [4,7], [3,5], [5,8]$, ta chọn tập gồm 4 số

$$\{2, 3, 5, 7\}$$

4.25. Cho phân số M/N ($0 < M < N$, M, N nguyên). Hãy phân tích phân số này thành tổng các phân số có tử số bằng 1, càng ít số hạng càng tốt.

Dữ liệu vào từ file "PS.IN" chứa 2 số M, N

Kết quả ra file "PS.OUT"

- Dòng đầu là số lượng số tách

- Các dòng sau mỗi dòng chứa mẫu số của các số hạng

Dữ liệu vào	Kết quả ra
5 6	2
	2 3

4.26. Cho một số tự nhiên N . Hãy tìm cách phân tích số N thành các số nguyên dương p_1, p_2, \dots, p_k (với $k > 1$) sao cho:

- p_1, p_2, \dots, p_k đôi một khác nhau
- $p_1 + p_2 + \dots + p_k = N$
- $S = p_1 * p_2 * \dots * p_k$ đạt giá trị lớn nhất

Dữ liệu vào trong file: "PT.INP" có dạng: Gồm nhiều test, mỗi dòng là một test chứa một số N ($5 \leq N \leq 1000$)

Kết quả ra file: "PT.OUT" có dạng: Gồm nhiều dòng, mỗi dòng là tích lớn nhất đạt được (số S) cho test đó

Dữ liệu vào	Kết quả ra
5	6
7	12

4.27. Cho hai phép toán $*2$ (nhân với 2) và $/3$ (chia nguyên cho 3). Cho trước số 1, bằng cách sử dụng hai phép toán trên ta xây dựng được biểu thức có giá trị bằng N .

Ví dụ $N=6$ thì $1*2*2*2*2*2/3/3*2=6$ (thực hiện từ trái qua phải)

Dữ liệu vào từ file "BT.INP" chứa số N (N có không quá 100 chữ số)

Ghi kết quả ra file "BT.OUT" biểu thức ngắn nhất có thể

4.28. Cho số nguyên dương N ($N \leq 10^{100}$), hãy tách N thành tổng ít các số Fibonacci nhất.

Ví dụ: $N=16=1+5+13$

4.29. Cần phải tổ chức việc thực hiện N chương trình đánh số từ 1 đến N trên một máy tính. Mỗi chương trình thứ i đòi hỏi thời gian tính là 1 giờ, và nếu nó được hoàn thành trước thời điểm $d[i]$ (giả sử thời điểm bắt đầu thực hiện các chương trình là 0) thì người chủ máy tính sẽ được trả tiền công là $w[i]$ ($i = 1, 2, \dots, N$). Việc thực hiện mỗi chương trình phải được tiến hành liên tục từ lúc bắt đầu cho đến khi kết thúc không cho phép ngắt quãng, đồng thời tại mỗi thời điểm máy chỉ có thể thực hiện một chương trình).

Hãy tìm trình tự thực hiện các chương trình sao cho tổng tiền công nhận được là lớn nhất.

Dữ liệu vào được cho trong *JOB.INP*:

- Dòng đầu tiên chứa số N ($N \leq 5000$),
- Dòng thứ i trong N dòng tiếp theo chứa 2 số $d[i]$, $w[i]$ được ghi cách nhau bởi dấu cách.

Kết quả đưa ra file *JOB.OUT*:

- Dòng đầu tiên chứa tổng tiền công nhận được theo trình tự tìm được.
- Dòng tiếp theo ghi trình tự thực hiện các chương trình.

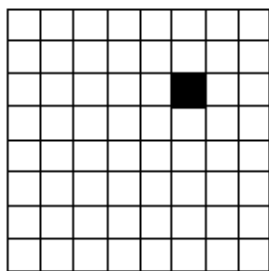
4.30. Tìm K chữ số cuối cùng của M^N ($0 < K \leq 9$, $0 \leq M$, $N \leq 10^9$)

Ví dụ: $K=2$, $M=2$, $N=10$, ta có $2^{10}=1024$, như vậy 2 chữ số cuối cùng của 2^{10} là 24

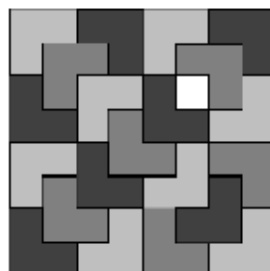
4.31. Viết hàm kiểm tra tính nguyên tố của số N ($N \leq 10^9$) theo Fermat.

4.32. Lát gạch

Cho một nền nhà hình vuông có kích thước 2^k bị khuyết một ô, hãy tìm cách lát nền nhà bằng loại gạch hình thước thợ (tạo bởi 3 hình vuông đơn vị).



nền nhà (ô màu đen là ô khuyết)



một cách lát nền

4.33. Cho dãy a_1, a_2, \dots, a_n , các số đôi một khác nhau và số nguyên dương k ($1 \leq k \leq n$). Hãy đưa ra giá trị nhỏ thứ k trong dãy.

Ví dụ: dãy gồm 5 phần tử: 5, 7, 1, 3, 4 và $k = 3$ thì giá trị nhỏ thứ k là 4.

4.34. Dãy con lồi

Dãy số nguyên A_1, A_2, \dots, A_N được gọi là lồi, nếu nó giảm dần từ A_1 đến một A_i nào đó, rồi tăng dần tới A_N .

Ví dụ dãy lồi: 10 5 4 2 -1 4 6 8 12

Yêu cầu: Cho một dãy số nguyên, bằng cách xóa bớt một số phần tử của dãy và giữ nguyên trình tự các phần tử còn lại, ta nhận được dãy con lồi dài nhất.

Dữ liệu vào trong file: *DS.INP*

- Dòng đầu là N ($N \leq 10000$)
- Các dòng sau là N số nguyên của dãy số (các số kiểu longint)

Kết quả ra file: *DS.OUT*

- Ghi số phần tử của dãy con tìm được
- Các dòng tiếp theo ghi các số thuộc dãy con

4.35. *Palindrome*

Một xâu được gọi là xâu đối xứng nếu đọc từ trái qua phải cũng giống như đọc từ phải qua trái. Ví dụ xâu “madam” là một xâu đối xứng. Bài toán đặt ra là cho một xâu S gồm các kí tự thuộc tập $['a'..'z']$, hãy tìm cách chèn vào xâu S ít nhất các kí tự để xâu S thành xâu đối xứng.

Ví dụ: xâu “adbhbca” ta sẽ chèn thêm 2 kí tự (c và d) để được xâu đối xứng “ad**cb**hb**cd**a”.

Dữ liệu vào trong file PALIN.INP có dạng: Gồm một dòng chứa xâu S. (độ dài mỗi xâu không vượt quá 200)

Kết quả ghi ra file PALIN.OUT có dạng: Gồm một dòng là một xâu đối xứng sau khi đã chèn thêm ít kí tự nhất vào xâu S.

Palin.inp	Palin.out
acbcd	ad cb cd a

4.36. *Stones*

Có N đồng sỏi xếp thành một hàng, đồng thứ i có A_i viên sỏi. Ta có thể ghép hai đồng sỏi kề nhau thành một đồng và mất một chi phí bằng tổng hai đồng sỏi đó.

Yêu cầu: Hãy tìm cách ghép N đồng sỏi này thành một đồng với chi phí là nhỏ nhất.

Ví dụ: Có 5 đồng sỏi

$$\begin{array}{ccccccc} 4 & & \underline{1} & \underline{2} & & 7 & 5 \\ 4 & & \underline{\quad} & \underline{3} & & 7 & 5 \\ & 7 & & \underline{7} & \underline{5} & & \\ & & 7 & & \underline{12} & & \\ & & & & & 19 & \end{array}$$

$$\text{Phạt} = 3 + 7 + 12 + 19 = 41$$

Dữ liệu vào trong file “*STONES.INP*” có dạng:

- Dòng đầu là số N ($N < 101$) là số đồng sỏi
 - Dòng thứ 2 gồm N số nguyên là số sỏi của N đồng sỏi. ($0 < A_i < 1001$)
- Kết quả ra file "STONES.OUT" có dạng: gồm một số là chi phí nhỏ nhất để ghép N đồng thành một đồng.

STONES . INP	STONES . OUT
5	41
4 1 2 7 5	

4.37. Cắt hình 1

Có một hình chữ nhật $M \times N$ ô, mỗi lần ta được phép cắt một hình chữ nhật thành hai hình chữ nhật con theo chiều ngang hoặc chiều dọc và lại tiếp tục cắt các hình chữ nhật con cho đến khi được hình vuông thì dừng.

Hỏi có thể cắt hình chữ nhật $M \times N$ thành ít nhất bao nhiêu hình vuông.

Dữ liệu vào trong file HCN.INP:

Gồm 1 số dòng, mỗi dòng là 1 test là một cặp số M, N ($1 \leq M, N \leq 100$)

Kết quả ra file HCN.INP:

Gồm 1 số dòng là kết quả tương ứng với dữ liệu vào

4.38. Cắt hình 2

Cho một bảng số A gồm M dòng, N cột, các giá trị của bảng A chỉ là 0 hoặc 1. Ta muốn cắt bảng A thành các hình chữ nhật con sao cho các hình chữ nhật con có giá trị toàn bằng 1 hay toàn bằng 0. Một lần cắt là một nhất cắt thẳng theo dòng hoặc theo cột của một hình chữ nhật thành hai hình chữ nhật riêng biệt. Cứ tiếp tục cắt cho đến khi hình chữ nhật toàn bằng 1 hay toàn bằng 0. Hãy tìm cách cắt để được ít hình chữ nhật nhất mà các hình chữ nhật con có giá trị toàn bằng 1 hay toàn bằng 0.

Ví dụ: Bảng số 5×5 sau được chia thành 8 hình chữ nhật con.

0	1	0	0	1
0	1	0	0	1
1	1	0	0	1
1	1	1	0	0
0	0	1	0	0

0	1	0	0	1
0	1	0	0	1
1	1	0	0	1
1	1	1	0	0
0	0	1	0	0

Dữ liệu vào trong file HCN2.INP

- Dòng đầu là 2 số nguyên dương M, N ($M, N \leq 30$)
 - M dòng tiếp theo, mỗi dòng N số chỉ gồm 0 hoặc 1 thể hiện bảng số A
- Kết quả ra file HCN2*
- Gồm 1 dòng duy nhất chứa một số duy nhất là số hình chữ nhật ít nhất

4.39. TKSEQ

Cho dãy số A gồm N số nguyên và số nguyên K. Tìm dãy chỉ số $1 \leq i_1 < i_2 < \dots < i_{3K} \leq N$ sao cho:

$$S = (a_{i_1} - a_{i_2} + a_{i_3}) + (a_{i_4} - a_{i_5} + a_{i_6}) + \dots + (a_{i_{3K-2}} - a_{i_{3K-1}} + a_{i_{3K}})$$

đạt giá trị lớn nhất.

Dữ liệu vào trong file “TKSEQ.INP” có dạng:

- Dòng đầu là gồm 2 số nguyên N, K ($0 < 3K \leq N \leq 500$)
- Dòng 2 gồm N số nguyên a_1, a_2, \dots, a_N ($|a_i| < 10^9$)

Kết quả ra file “TKSEQ.OUT” có dạng: gồm một số duy nhất S lớn nhất tìm được

TKSEQ.INP	TKSEQ.OUT
5 1 1 2 3 4 5	4

4.40. Least-Squares Segmentation

Ta định nghĩa trọng số của đoạn số từ số ở vị trí thứ i đến vị trí thứ j của dãy số nguyên $A[1], A[2], \dots, A[N]$ là:

$$\sum_{k=i}^j (A[k] - \text{mean})^2 \text{ trong đó } \text{mean} = (\sum_{k=i}^j A[k]) / (j - i + 1)$$

Yêu cầu: Cho dãy số nguyên A gồm N số $A[1], A[2], \dots, A[N]$ và số nguyên dương G ($1 < G^2 < N$). Hãy chia dãy A thành đúng G đoạn để tổng trọng số là nhỏ nhất.

Dữ liệu vào trong file văn bản “LSS.INP” có dạng:

- Dòng đầu gồm hai số N và G ($1 < G^2 < N < 1001$)
- N dòng tiếp theo, mỗi dòng một số nguyên mô tả dãy số A ($0 < A[i] < 10^6$)

Kết quả ra file văn bản “LSS.OUT” có dạng: gồm một dòng chứa một số thực duy nhất là đáp án của bài toán. (đưa ra theo quy cách :0:2)

LSS.INP	LSS.OUT
5 2 3	0.50

3	
3	
4	
5	

4.41. *Phân trang* (Đề thi chọn đội tuyển quốc gia 1999)

Văn bản là một dãy gồm N từ đánh số từ 1 đến N . Từ thứ i có độ dài là w_i ($i=1, 2, \dots, N$). Phân trang là một cách xếp lần lượt các từ của văn bản vào dãy các dòng, mỗi dòng có độ dài L , sao cho tổng độ dài của các từ trên cùng một dòng không vượt quá L . Ta gọi hệ số phạt của mỗi dòng trong cách phân trang là hiệu số $(L-S)$, trong đó S là tổng độ dài của các từ xếp trên dòng đó. Hệ số phạt của cách phân trang là giá trị lớn nhất trong số các hệ số phạt của các dòng.

Yêu cầu: Tìm cách phân trang với hệ số phạt nhỏ nhất.

Dữ liệu vào từ tệp văn bản PTRANG.INP

- Dòng 1 chứa 2 số nguyên dương N, L ($N \leq 4000, L \leq 70$)
- Dòng thứ i trong số N dòng tiếp theo chứa số nguyên dương w_i ($w_i \leq L$), $i=1, 2, \dots, N$

Kết quả ghi ra file văn bản PTRANG.OUT

- Dòng 1 ghi 2 số P, Q theo thứ tự là hệ số phạt và số dòng theo cách phân trang tìm được
- Dòng thứ i trong số Q dòng tiếp theo ghi chỉ số của các từ trong dòng thứ i của cách phân trang.

4.42. *Chọn số*

Cho mảng A có kích thước $N \times N$ gồm các số nguyên không âm. Hãy chọn ra K số sao cho mỗi dòng có nhiều nhất 1 số được chọn, mỗi cột có nhiều nhất 1 số được chọn để tổng K số là lớn nhất.

Dữ liệu vào từ tệp văn bản SELECT.INP

- Dòng thứ nhất gồm 2 số N và K ($K \leq N \leq 15$)
- N dòng sau, mỗi dòng N số nguyên không âm $A_{ij} < 10000$

Kết quả ghi ra file văn bản SELECT.OUT

Tổng lớn nhất chọn được và số cách chọn (cách nhau đúng một dấu cách)

Select.inp	Select.out
3 2	6 3
1 2 3	
2 3 1	
3 1 2	

4.43. *Puzzle of numbers*

Khi một số phần chữ số trong đẳng thức đúng của tổng hai số nguyên bị mất (được thay bởi các dấu sao “*”). Có một câu đố là: Hãy thay các dấu sao bởi các chữ số để cho đẳng thức vẫn đúng.

Ví dụ bắt đầu từ đẳng thức sau:

9334

789

10123 (9334+789=10123)

Các ví dụ các chữ số bị mất được thay bằng các dấu sao như sau:

*3*4	hay	****
78*		***
10123		*****

Nhiệm vụ của bạn là viết chương trình thay các dấu sao thành các chữ số để được một đẳng thức đúng. Nếu có nhiều lời giải thì đưa ra một trong số đó. Nếu không có thì đưa ra thông báo: “**No Solution**”.

Chú ý các chữ số ở đầu mỗi số phải khác 0.

Dữ liệu vào trong file “REBUSS.INP”: gồm 3 dòng, mỗi dòng là một xâu kí tự gồm các chữ số hoặc kí tự “*”. Độ dài mỗi xâu không quá 50 kí tự. Dòng 1, dòng 2 thể hiện là hai số được cộng, dòng 3 thể hiện là tổng hai số.

Kết quả ra file “REBUSS.OUT”: Nếu có lời giải thì file kết quả gồm 3 dòng tương ứng với file dữ liệu vào, nếu không thì thông báo “**No Solution**”

REBUSS . INP	REBUSS . OUT
*3*4	9334
78*	789
10123	10123

4.44. Xếp lịch giảng

Một giáo viên cần giảng n vấn đề được đánh số từ 1 đến n ($n \leq 10000$). Mỗi một vấn đề i cần có thời gian là t_i ($i = 1..n$). Để giảng n vấn đề đó thì giáo viên có các buổi đã được phân có độ dài là L ($L \leq 500$).

- Một vấn đề thì phải giải quyết trong một buổi.
- Vấn đề i phải được giảng trước vấn đề $i + 1$ với mọi $i = 1..(n - 1)$.

Học sinh có thể ra về sớm nếu như buổi giảng đã kết thúc, tuy nhiên nếu thời gian ra về đó quá sớm so với buổi giảng thì thật là phí. Chính vì thế người ta đánh giá buổi lên lớp bằng giá trị DI như sau :

$$DI = \begin{cases} 0 & \text{nu } t = 0 \\ -C & \text{nu } 1 \leq t \leq 10 \\ (t - 10)^2 & \text{nu } t > 10 \end{cases}$$

Trong đó t là thời gian thừa của buổi lên lớp đó, C là một hằng số.

Yêu cầu: Hãy xếp lịch dạy sao cho tổng số các buổi là cần ít nhất có thể được. Trong các lịch dạy ít nhất đó, hãy tìm lịch dạy sao cho tổng số DI là nhỏ nhất có thể được.

Dữ liệu vào từ file *SCHEDULING.INP*

- Dòng đầu là số n (số vấn đề cần giảng).
- Dòng tiếp theo là L và C
- Dòng cuối cùng là N số thể hiện cho t_1, t_2, \dots, t_n .

Kết quả ra file *SCHEDULING.OUT*

- Dòng đầu tiên là số buổi.
- Dòng tiếp theo là tổng DI nhỏ nhất đạt được.

SCHEDULING.INP	SCHEDULING.OUT
10	6
120 10	2700
80 80 10 50 30 20 40 30 120 100	

4.45. Khu vườn (IOI 2008)

Ramsesses II thắng trận trở về. Để ghi nhận chiến tích của mình ông quyết định xây một khu vườn tráng lệ. Khu vườn phải có một hàng cây chạy dài từ cung điện của ông tại Luxor tới thánh đường Karnak. Hàng cây này chỉ chứa hai loại cây là sen và cói giấy, bởi vì chúng tương ứng là biểu tượng của miền Thượng Ai Cập và Hạ Ai Cập.

Vườn phải có đúng N cây. Ngoài ra, phải có sự cân bằng: ở mọi đoạn cây liên tiếp của vườn, số lượng sen và số lượng cói giấy phải không lệch nhau quá 2.

Vườn cây được biểu diễn dưới dạng xâu các kí tự 'L' (lotus – sen) và 'P' (papyrus – cói giấy). Ví dụ, với $N = 5$ có tất cả 14 vườn đảm bảo cân bằng. Theo thứ tự từ điển, các vườn đó là: LLPLP, LLPPL, LPLLP, LPLPL, LPLPP, LPPLL, LPPLP, PLLPL, PLLPP, PLPLL, PLPLP, PLPPL, PLLLP và PPLPL.

Các vườn cân bằng với độ dài xác định cho trước được sắp xếp theo thứ tự từ điển và được đánh số từ 1 trở đi. Ví dụ, với $N=5$, vườn số 12 sẽ là vườn PLPPL.

NHIỆM VỤ

Cho số cây N và xâu biểu diễn một vườn cân bằng, hãy lập trình tính số thứ tự của vườn này theo môđun M , trong đó M là số nguyên cho trước.

Lưu ý rằng giá trị của M không đóng vai trò quan trọng trong việc giải bài toán, nó chỉ làm cho việc tính toán trở nên đơn giản.

HẠN CHẾ

$$1 \leq N \leq 1\,000\,000$$

$$7 \leq M \leq 10\,000\,000$$

CHẤM ĐIỂM

Có 40 điểm dành cho các dữ liệu vào với N không vượt quá 40.

INPUT

Chương trình của bạn phải đọc từ file “GARDEN.INP” các dữ liệu sau:

- Dòng 1 chứa số nguyên N , số cây trong vườn,
- Dòng 2 chứa số nguyên M ,
- Dòng 3 chứa xâu gồm N kí tự 'L' (sen) hoặc 'P' (cói giấy) biểu diễn vườn cân bằng.

OUTPUT

Chương trình của bạn phải ghi ra file “GARDEN.OUT” một dòng chứa một số nguyên trong phạm vi từ 0 đến $M-1$, là số thứ tự từ theo môđun M của vườn được mô tả trong đầu vào.

Input ví dụ 1	Output ví dụ 1	Giải thích

5 7 PLPPL	5	Số thứ tự của PLPPL là 12. Như vậy output là 12 theo môđun 7, tức là 5.
-----------------	---	---

Input ví dụ 2	Output ví dụ 2
12 10000 LPLLPLPPLPLL	39

4.46. Số rõ ràng

Bờm mới tìm được một tài liệu định nghĩa số rõ ràng như sau: Với số nguyên dương n , ta tạo số mới bằng cách lấy tổng bình phương các chữ số của nó, với số mới này ta lại lặp lại công việc trên. Nếu trong quá trình đó, ta nhận được số mới là 1, thì số n ban đầu được gọi là số rõ ràng. Ví dụ, với $n = 19$, ta có:

$$19 \rightarrow 82 (= 1^2 + 9^2) \rightarrow 68 \rightarrow 100 \rightarrow 1$$

Như vậy, 19 là số rõ ràng.

Không phải mọi số đều rõ ràng. Ví dụ, với $n = 12$, ta có:

$$12 \rightarrow 5 \rightarrow 25 \rightarrow 29 \rightarrow 85 \rightarrow 89 \rightarrow 145 \rightarrow 42 \rightarrow 20 \rightarrow 4 \rightarrow 16 \rightarrow 37 \rightarrow 58 \rightarrow 89 \rightarrow 145$$

Bờm rất thích thú với định nghĩa số rõ ràng này và thách đố phú ông: Cho một số nguyên dương n , tìm số $S(n)$ là số rõ ràng liền sau số n , tức là $S(n)$ là số rõ ràng nhỏ nhất lớn hơn n . Tuy nhiên, câu hỏi đó quá dễ với phú ông và phú ông đã đố lại Bờm: Cho hai số nguyên dương n và k ($1 \leq n, k \leq 10^{15}$), hãy tìm số $S^k(n) = S(S(\dots S(n)))$ là số rõ ràng liền sau thứ k của n .

Bạn hãy giúp Bờm giải câu đố này nhé!

Dữ liệu vào từ file văn bản *CLEAR.INP* có dạng:

- Dòng đầu là số t ($0 < t \leq 20$)
- t dòng sau, mỗi dòng chứa 2 số nguyên n và k .

Kết quả ra file văn bản *CLEAR.OUT* gồm t dòng, mỗi dòng là kết quả tương ứng với dữ liệu vào.

CLEAR.INP	CLEAR.OUT
2	19
18 1	1000000000
1 145674807	

4.47. Hái nấm

Bé Bông đi hái nấm trong N khu rừng đánh số từ 1 đến N, nhưng chỉ có M khu rừng có nấm. Việc di chuyển từ khu rừng thứ i sang khu rừng thứ j tốn t_{ij} đơn vị thời gian. Đến khu rừng i có nấm, cô bé có thể dừng lại để hái nấm. Nếu tổng số đơn vị thời gian cô bé dừng lại ở khu rừng thứ i là d_i ($d_i > 0$), thì cô bé hái được: $\left\lfloor \frac{S_i}{2} \right\rfloor + \left\lfloor \frac{S_i}{4} \right\rfloor + \dots + \left\lfloor \frac{S_i}{2^{d_i}} \right\rfloor$ cây nấm tại khu rừng đó (trong đó S_i là số lượng nấm có tại khu rừng i, $\lfloor x \rfloor$ là phần nguyên của x). Giả thiết rằng ban đầu cô bé ở khu rừng thứ nhất và đi hái nấm trong thời gian không quá P đơn vị.

Yêu cầu: Hãy tính số lượng cây nấm nhiều nhất mà cô bé có thể hái được.

Dữ liệu vào từ file văn bản MUSHROOM.INP:

- Dòng đầu tiên chứa ba số nguyên dương M ($M \leq 10$), N ($0 < M \leq N \leq 100$) và P ($P \leq 10000$);
- M dòng tiếp theo, mỗi dòng chứa 2 số nguyên dương r và S_r nghĩa là khu rừng r có S_r nấm ($S_r \leq 10^9$);
- Dòng thứ i trong N dòng cuối cùng chứa N số nguyên dương t_{ij} ($t_{ij} \leq 10000$), ($i, j=1, \dots, N$).

Kết quả ghi ra file văn bản MUSHROOM.OUT: số lượng cây nấm nhiều nhất bé Bông có thể hái được.

MUSHROOM.INP	MUSHROOM.OUT
2 2 2	3
1 5	
2 10	
0 3	
3 0	

CÁC THUẬT TOÁN TRÊN ĐỒ THỊ

Trên thực tế có nhiều bài toán liên quan tới một tập các đối tượng và những mối liên hệ giữa chúng, đòi hỏi toán học phải đặt ra một mô hình biểu diễn một cách chặt chẽ và tổng quát bằng ngôn ngữ kí hiệu, đó là đồ thị: một mô hình toán học gồm các đỉnh biểu diễn các đối tượng và các cạnh biểu diễn mối quan hệ giữa các đối tượng.

Những ý tưởng cơ bản của đồ thị được đưa ra từ thế kỉ thứ XVIII bởi nhà toán học Thụy Sĩ Leonhard Euler, năm 1736, ông đã dùng mô hình đồ thị để giải bài toán về bảy cây cầu Königsberg (*Seven Bridges of Königsberg*). Bài toán này cùng với bài toán mã đi tuần (*Knight Tour*) được coi là những bài toán đầu tiên của lí thuyết đồ thị.

Rất nhiều bài toán của lí thuyết đồ thị đã trở thành nổi tiếng và thu hút được sự quan tâm lớn của cộng đồng nghiên cứu. Ví dụ bài toán bốn màu, bài toán đẳng cấu đồ thị, bài toán người du lịch, bài toán người đưa thư Trung Hoa, bài toán đường đi ngắn nhất, luồng cực đại trên mạng v.v... Trong phạm vi một chuyên đề, không thể trình bày tất cả những gì đã phát triển trong suốt gần 300 năm, chúng ta sẽ xem xét lí thuyết đồ thị dưới góc độ người lập trình, tức là khảo sát những thuật toán cơ bản nhất có thể dễ dàng cài đặt trên máy tính một số ứng dụng của nó. Công việc của người lập trình là đọc hiểu được ý tưởng cơ bản của thuật toán và cài đặt được chương trình trong bài toán tổng quát cũng như trong trường hợp cụ thể.



Leonhard Euler
1707-1783

1. Các khái niệm cơ bản

1.1. Đồ thị

Đồ thị là mô hình biểu diễn một tập các đối tượng và mối quan hệ hai ngôi giữa các đối tượng:

$$\text{Graph} = \text{Objects} + \text{Connections}$$

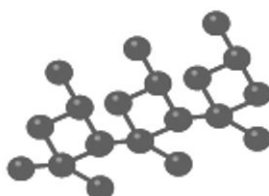
$$G = (V, E)$$

Có thể định nghĩa đồ thị G là một cặp (V, E) : $G = (V, E)$. Trong đó V là tập các đỉnh (vertices) biểu diễn các đối tượng và E gọi là tập các cạnh (edges) biểu diễn mối quan hệ giữa các đối tượng. Chúng ta quan tâm tới mối quan hệ hai ngôi (pairwise relations) giữa các đối tượng nên có thể coi E là tập các cặp (u, v) với u và v là hai đỉnh của V biểu diễn hai đối tượng có quan hệ với nhau.

Một số hình ảnh của đồ thị:



Sơ đồ giao thông



Cấu trúc phân tử



Mạng máy tính

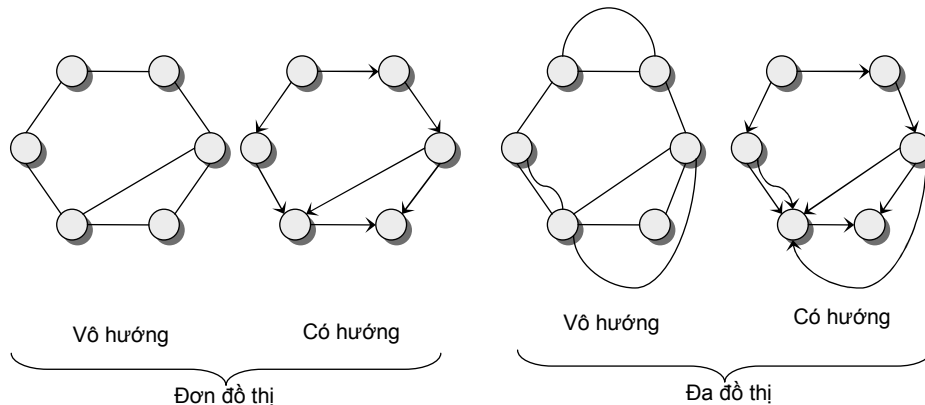
Hình 5-1. Ví dụ về mô hình đồ thị

Có thể phân loại đồ thị $G = (V, E)$ theo đặc tính và số lượng của tập các cạnh E :

- G được gọi là *đơn đồ thị* (hay gọi tắt là đồ thị) nếu giữa hai đỉnh $u, v \in V$ có nhiều nhất là 1 cạnh trong E nối từ u tới v .
- G được gọi là *đa đồ thị* (*multigraph*) nếu giữa hai đỉnh $u, v \in V$ có thể có nhiều hơn 1 cạnh trong E nối u và v (Hiển nhiên đơn đồ thị cũng là đa đồ thị). Nếu có nhiều cạnh nối giữa hai đỉnh $u, v \in V$ thì những cạnh đó được gọi là *cạnh song song* (parallel edges)
- G được gọi là *đồ thị vô hướng* (*undirected graph*) nếu các cạnh trong E là không định hướng, tức là cạnh nối hai đỉnh $u, v \in V$ bất kì cũng là cạnh nối hai đỉnh v, u . Hay nói cách khác, tập E gồm các cặp (u, v) không tính thứ tự: $(u, v) = (v, u)$.
- G được gọi là *đồ thị có hướng* (*directed graph*) nếu các cạnh trong E là có định hướng, tức là có thể có cạnh nối từ đỉnh u tới đỉnh v nhưng chưa chắc đã có cạnh nối từ đỉnh v tới đỉnh u . Hay nói cách khác, tập E gồm các cặp (u, v)

có tính thứ tự: $(u, v) \neq (v, u)$. Trong đồ thị có hướng, các cạnh còn được gọi là các *cung* (*arcs*). Đồ thị vô hướng cũng có thể coi là đồ thị có hướng nếu như ta coi cạnh nối hai đỉnh u, v bất kì tương đương với hai cung (u, v) và (v, u) .

Hình 5-2 là ví dụ về đơn đồ thị/đa đồ thị có hướng/vô hướng.



Hình 5-2. Phân loại đồ thị

1.2. Các khái niệm

Như trên định nghĩa đồ thị $G = (V, E)$ là một cấu trúc rời rạc, tức là các tập V và E là tập không quá đếm được, vì vậy ta có thể đánh số thứ tự 1, 2, 3... cho các phần tử của tập V và E và đồng nhất các phần tử của tập V và E với số thứ tự của chúng. Hơn nữa, đứng trên phương diện người lập trình cho máy tính thì ta chỉ quan tâm đến các đồ thị hữu hạn (V và E là tập hữu hạn) mà thôi, chính vì vậy từ đây về sau, nếu không chú thích gì thêm thì khi nói tới đồ thị, ta hiểu rằng đó là đồ thị hữu hạn.

a) Cạnh liên thuộc, đỉnh kề, bậc

Đối với đồ thị vô hướng $G = (V, E)$. Xét một cạnh $e \in E$, nếu $e = (u, v)$ thì ta nói hai đỉnh u và v là *kề nhau* (*adjacent*) và cạnh e này *liên thuộc* (*incident*) với đỉnh u và đỉnh v .

Với một đỉnh v trong đồ thị vô hướng, ta định nghĩa *bậc* (*degree*) của v , kí hiệu $\deg(v)$ là số cạnh liên thuộc với v . Trên đơn đồ thị thì số cạnh liên thuộc với v cũng là số đỉnh kề với v .

Định lý 5-1

|| Giả sử $G = (V, E)$ là đồ thị vô hướng, khi đó tổng tất cả các bậc đỉnh trong V sẽ bằng hai lần số cạnh:

$$\sum_{v \in V} \deg(v) = 2|E| \quad (1)$$

Chứng minh

Khi lấy tổng tất cả các bậc đỉnh tức là mỗi cạnh $e = (u, v)$ sẽ được tính một lần trong $\deg(u)$ và một lần trong $\deg(v)$. Từ đó suy ra kết quả.

Hệ quả

|| Trong đồ thị vô hướng, số đỉnh bậc lẻ là số chẵn.

Đối với đồ thị có hướng $G = (V, E)$. Xét một cung $e \in E$, nếu $e = (u, v)$ thì ta nói u nối tới v và v nối từ u , cung e là đi ra khỏi đỉnh u và đi vào đỉnh v . Đỉnh u khi đó được gọi là đỉnh đầu, đỉnh v được gọi là đỉnh cuối của cung e .

Với mỗi đỉnh v trong đồ thị có hướng, ta định nghĩa: *Bán bậc ra (out-degree)* của v kí hiệu $\deg^+(v)$ là số cung đi ra khỏi nó; *bán bậc vào (in-degree)* kí hiệu $\deg^-(v)$ là số cung đi vào đỉnh đó.

Định lý 5-2

|| Giả sử $G = (V, E)$ là đồ thị có hướng, khi đó tổng tất cả các bán bậc ra của các đỉnh bằng tổng tất cả các bán bậc vào và bằng số cung của đồ thị

$$\sum_{v \in V} \deg^+(v) = \sum_{v \in V} \deg^-(v) = |E| \quad (2)$$

Chứng minh

Khi lấy tổng tất cả các bán bậc ra hay bán bậc vào, mỗi cung (u, v) sẽ được tính đúng một lần trong $\deg^+(u)$ và cũng được tính đúng một lần trong $\deg^-(v)$. Từ đó suy ra kết quả.

b) Đường đi và chu trình

Một dãy các đỉnh:

$$P = \langle p_0, p_1, \dots, p_k \rangle$$

sao cho $(p_{i-1}, p_i) \in E, \forall i: 1 \leq i \leq k$ được gọi là một *đường đi (path)*, đường đi này gồm $k + 1$ đỉnh p_0, p_1, \dots, p_k và k cạnh $(p_0, p_1), (p_1, p_2), \dots, (p_{k-1}, p_k)$. Nếu có một đường đi như trên thì ta nói p_k *đến được (reachable)* từ p_0 hay p_0 đến được p_k , kí hiệu $p_0 \rightsquigarrow p_k$. Đỉnh p_0 được gọi là đỉnh đầu và đỉnh p_k gọi là đỉnh cuối của đường đi P . Các đỉnh p_1, p_2, \dots, p_{k-1} được gọi là *đỉnh trong* của đường đi P

Một đường đi gọi là *đơn giản (simple)* hay *đường đi đơn* nếu tất cả các đỉnh trên đường đi là hoàn toàn phân biệt (dĩ nhiên khi đó các cạnh trên đường đi cũng hoàn toàn phân biệt). Đường đi $P = \langle p_0, p_1, \dots, p_k \rangle$ trở thành *chu trình (circuit)* nếu $p_0 = p_k$. Trên đồ thị có hướng, chu trình P được gọi là *chu trình đơn* nếu nó có ít nhất một cung và các đỉnh p_1, p_2, \dots, p_k hoàn toàn phân biệt. Trên đồ thị vô hướng, chu trình P được gọi là chu trình đơn nếu $k \geq 3$ và các đỉnh p_1, p_2, \dots, p_k hoàn toàn phân biệt.

c) Một số khái niệm khác

Đồng cấu

Hai đồ thị $G = (V, E)$ và $G' = (V', E')$ được gọi là *đồng cấu (isomorphic)* nếu tồn tại một song ánh $f: V \rightarrow V'$ sao cho số cung nối u với v trên E bằng số cung nối $f(u)$ với $f(v)$ trên E' .

Đồ thị con

Đồ thị $G' = (V', E')$ là *đồ thị con (subgraph)* của đồ thị $G = (V, E)$ nếu $V' \subseteq V$ và $E' \subseteq E$.

Đồ thị con $G_U = (U, E_U)$ được gọi là *đồ thị con cảm ứng (induced graph)* từ đồ thị G bởi tập $U \subseteq V$ nếu $E_U = \{(u, v) \in E: u, v \in U\}$ trong trường hợp này chúng ta còn nói G_U là đồ thị G hạn chế trên U .

Phiên bản có hướng/vô hướng

Với một đồ thị vô hướng $G = (V, E)$, ta gọi *phiên bản có hướng (directed version)* của G là một đồ thị có hướng $G' = (V, E')$ tạo thành từ G bằng cách thay mỗi cạnh (u, v) bằng hai cung có hướng ngược chiều nhau: (u, v) và (v, u) .

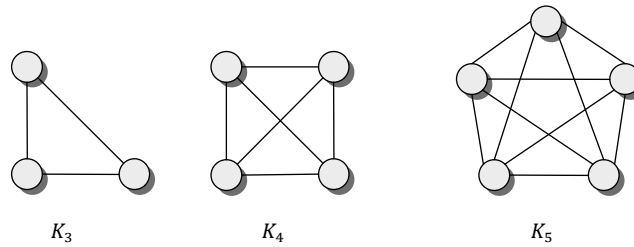
Với một đồ thị có hướng $G = (V, E)$, ta gọi *phiên bản vô hướng (undirected version)* của G là một đồ thị vô hướng $G' = (V, E')$ tạo thành bằng cách thay mỗi cung (u, v) bằng cạnh vô hướng (u, v) . Nói cách khác, G' tạo thành từ G bằng cách bỏ đi chiều của cung.

Tính liên thông

Một đồ thị vô hướng gọi là *liên thông (connected)* nếu giữa hai đỉnh bất kì của đồ thị có tồn tại đường đi. Đối với đồ thị có hướng, có hai khái niệm liên thông tùy theo chúng ta có quan tâm tới hướng của các cung hay không. Đồ thị có hướng gọi là *liên thông mạnh (strongly connected)* nếu giữa hai đỉnh bất kì của đồ thị có tồn tại đường đi. Đồ thị có hướng gọi là *liên thông yếu (weakly connected)* nếu phiên bản vô hướng của nó là đồ thị liên thông.

Đồ thị đầy đủ

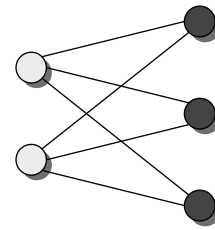
Một đồ thị vô hướng được gọi là *đầy đủ* (*complete*) nếu mọi cặp đỉnh đều là kề nhau, đồ thị đầy đủ gồm n đỉnh kí hiệu là K_n . Hình 5-3 là ví dụ về các đồ thị đầy đủ K_3 , K_4 và K_5 .



Hình 5-3. Đồ thị đầy đủ

Đồ thị hai phía

Một đồ thị vô hướng gọi là *hai phía* (*bipartite*) nếu tập đỉnh của nó có thể chia làm hai tập rời nhau X , Y sao cho không tồn tại cạnh nối hai đỉnh thuộc X cũng như không tồn tại cạnh nối hai đỉnh thuộc Y . Nếu $|X| = m$ và $|Y| = n$ và giữa mọi cặp đỉnh (x, y) trong đó $x \in X, y \in Y$ đều có cạnh nối thì đồ thị hai phía đó được gọi là đồ thị hai phía đầy đủ, kí hiệu $K_{m,n}$. Hình 5-4 là ví dụ về đồ thị hai phía đầy đủ $K_{2,3}$.



Hình 5-4. Đồ thị hai phía đầy đủ

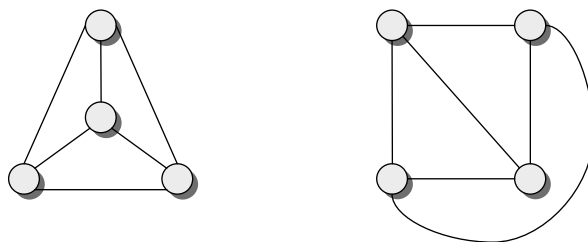
Đồ thị phẳng

Một đồ thị được gọi là *đồ thị phẳng* (*planar graph*) nếu chúng ta có thể vẽ đồ thị ra trên mặt phẳng sao cho:

- Mỗi đỉnh tương ứng với một điểm trên mặt phẳng, không có hai đỉnh cùng tọa độ.
- Mỗi cạnh tương ứng với một đoạn đường liên tục nối hai đỉnh, các điểm nằm trên hai cạnh bất kì là không giao nhau ngoại trừ các điểm đầu mút (tương ứng với các đỉnh)

Phép vẽ đồ thị phẳng như vậy gọi là biểu diễn phẳng của đồ thị

Ví dụ như đồ thị đầy đủ K_4 là đồ thị phẳng bởi nó có thể vẽ ra trên mặt phẳng như Hình 5-5



Hình 5-5. Hai cách vẽ đồ thị phẳng của K_4

Định lí 5-3 (Định lí Kuratowski)

Một đồ thị vô hướng là đồ thị phẳng nếu và chỉ nếu nó không chứa đồ thị con đẳng cấu với $K_{3,3}$ hoặc K_5 .

Định lí 5-4 (Công thức Euler)

Nếu một đồ thị vô hướng liên thông là đồ thị phẳng và biểu diễn phẳng của đồ thị đó gồm v đỉnh và e cạnh chia mặt phẳng thành f phần thì $v - e + f = 2$.

Định lí 5-5

Nếu đơn đồ thị vô hướng $G = (V, E)$ là đồ thị phẳng có ít nhất 3 đỉnh thì $|E| \leq 3|V| - 6$. Ngoài ra nếu G không có chu trình độ dài 3 thì $|E| \leq 2|V| - 4$.

Định lí 5-5 chỉ ra rằng số cạnh của đơn đồ thị phẳng là một đại lượng $|E| = O(|V|)$ điều này rất hữu ích đối với nhiều thuật toán trên đồ thị thưa (có ít cạnh).

Đồ thị đường

Từ đồ thị vô hướng G , ta xây dựng đồ thị vô hướng G' như sau: Mỗi đỉnh của G' tương ứng với một cạnh của G , giữa hai đỉnh x, y của G' có cạnh nối nếu và chỉ nếu tồn tại đỉnh liên thuộc với cả hai cạnh x, y trên G . Đồ thị G' như vậy được gọi là đồ thị đường của đồ thị G . Đồ thị đường được nghiên cứu trong các bài toán kiểm tra tính liên thông, tập độc lập cực đại, tô màu cạnh đồ thị, chu trình Euler và chu trình Hamilton v.v...

2. Biểu diễn đồ thị

Khi lập trình giải các bài toán được mô hình hoá bằng đồ thị, việc đầu tiên cần làm tìm cấu trúc dữ liệu để biểu diễn đồ thị sao cho việc giải quyết bài toán được thuận tiện nhất.

Có rất nhiều phương pháp biểu diễn đồ thị, trong bài này chúng ta sẽ khảo sát một số phương pháp phổ biến nhất. Tính hiệu quả của từng phương pháp biểu diễn sẽ được chỉ rõ hơn trong từng thuật toán cụ thể.

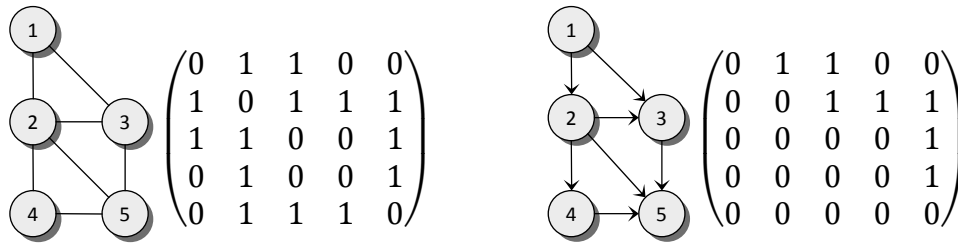
2.1. Ma trận kề

Với $G = (V, E)$ là một đơn đồ thị có hướng trong đó $|V| = n$, ta có thể đánh số các đỉnh từ 1 tới n và đồng nhất mỗi đỉnh với số thứ tự của nó. Bằng cách đánh số như vậy, đồ thị G có thể biểu diễn bằng ma trận vuông $A = \{a_{ij}\}_{n \times n}$. Trong đó:

$$a_{ij} = \begin{cases} 1, & \text{nếu } (i, j) \in E \\ 0, & \text{nếu } (i, j) \notin E \end{cases}$$

Với $\forall i$, giá trị của các phần tử trên đường chéo chính ma trận $A: \{a_{ii}\}$ có thể đặt tùy theo mục đích cụ thể, chẳng hạn đặt bằng 0. Ma trận A xây dựng như vậy được gọi là *ma trận kề* (*adjacency matrix*) của đồ thị G . Việc biểu diễn đồ thị vô hướng được quy về việc biểu diễn phiên bản có hướng tương ứng: thay mỗi cạnh (i, j) bởi hai cung ngược hướng nhau: (i, j) và (j, i) .

Đối với đa đồ thị thì việc biểu diễn cũng tương tự trên, chỉ có điều nếu như (i, j) là cung thì a_{ij} là số cạnh nối giữa đỉnh i và đỉnh j .



Hình 5-6. Ma trận kề biểu diễn đồ thị

Trong trường hợp G là đơn đồ thị, ta có thể biểu diễn ma trận kề A tương ứng là các phân tử logic:

$$a_{ij} = \begin{cases} \text{True}, & \text{nếu } (i, j) \in E \\ \text{False}, & \text{nếu } (i, j) \notin E \end{cases}$$

Có một cách khác biểu diễn đồ thị vô hướng G bằng ma trận $A = \{a_{ij}\}_{n \times n}$ như sau:

$$a_{ij} = \begin{cases} \deg(i), & \text{nếu } i = j \\ -1, & \text{nếu } (i, j) \in E \\ 0, & \text{TH khác} \end{cases}$$

Cách biểu diễn này có ứng dụng trong một số bài toán đồ thị, gọi là biểu diễn bằng ma trận Laplace (*Laplacian matrix* hay *Kirchhoff matrix*)

Ma trận kề có một số tính chất:

- Đối với đồ thị vô hướng G , thì ma trận kề tương ứng là ma trận đối xứng $a_{ij} = a_{ji}$, điều này không đúng với đồ thị có hướng.
- Nếu G là đồ thị vô hướng và A là ma trận kề tương ứng thì trên ma trận A , tổng các số trên hàng i bằng tổng các số trên cột i và bằng bậc của đỉnh i : $\deg(i)$
- Nếu G là đồ thị có hướng và A là ma trận kề tương ứng thì trên ma trận A , tổng các số trên hàng i bằng bán bậc ra của đỉnh i : $\deg^+(i)$, tổng các số trên cột i bằng bán bậc vào của đỉnh i : $\deg^-(i)$

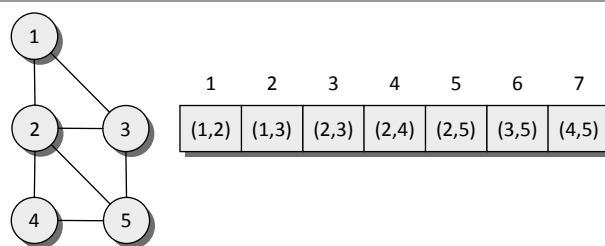
Ưu điểm của ma trận kề:

- Đơn giản, trực quan, dễ cài đặt trên máy tính
- Để kiểm tra xem hai đỉnh (u, v) của đồ thị có kề nhau hay không, ta chỉ việc kiểm tra bằng một phép so sánh: $a_{uv} \neq 0$

Nhược điểm của ma trận kề

- Bất kể số cạnh của đồ thị là nhiều hay ít, ma trận kề luôn luôn đòi hỏi n^2 ô nhớ để lưu các phần tử ma trận, điều đó gây lãng phí bộ nhớ.
- Một số bài toán yêu cầu thao tác liệt kê tất cả các đỉnh v kề với một đỉnh u cho trước. Trên ma trận kề việc này được thực hiện bằng cách xét tất cả các đỉnh v và kiểm tra điều kiện $a_{uv} \neq 0$. Như vậy, ngay cả khi đỉnh u là *đỉnh cô lập* (không kề với đỉnh nào) hoặc *đỉnh treo* (chỉ kề với 1 đỉnh) ta cũng buộc phải xét tất cả các đỉnh v và kiểm tra giá trị tương ứng a_{uv} .

2.2. Danh sách cạnh



Hình 5-7. Danh sách cạnh

Với đồ thị $G = (V, E)$ có n đỉnh, m cạnh, ta có thể liệt kê tất cả các cạnh của đồ thị trong một danh sách, mỗi phần tử của danh sách là một cặp (x, y) tương ứng với một cạnh của E , trong trường hợp đồ thị có hướng thì mỗi cặp (x, y) tương

ứng với một cung, x là đỉnh đầu và y là đỉnh cuối của cung. Cách biểu diễn này gọi là *danh sách cạnh* (*edge list*).

Có nhiều cách xây dựng cấu trúc dữ liệu để biểu diễn danh sách, nhưng phổ biến nhất là dùng mảng hoặc danh sách móc nối.

Ưu điểm của danh sách cạnh:

- Trong trường hợp đồ thị thưa (có số cạnh tương đối nhỏ), cách biểu diễn bằng danh sách cạnh sẽ tiết kiệm được không gian lưu trữ, bởi nó chỉ cần $O(m)$ ô nhớ để lưu danh sách cạnh.
- Trong một số trường hợp, ta phải xét tất cả các cạnh của đồ thị thì cài đặt trên danh sách cạnh làm cho việc duyệt các cạnh dễ dàng hơn. (Thuật toán Kruskal chẳng hạn)

Nhược điểm của danh sách cạnh:

- Nhược điểm cơ bản của danh sách cạnh là khi ta cần duyệt tất cả các đỉnh kề với đỉnh v nào đó của đồ thị, thì chẳng có cách nào khác là phải duyệt tất cả các cạnh, lọc ra những cạnh có chứa đỉnh v và xét đỉnh còn lại.
- Việc kiểm tra hai đỉnh u, v có kề nhau hay không cũng bắt buộc phải duyệt danh sách cạnh, điều đó khá tốn thời gian trong trường hợp đồ thị dày (nhiều cạnh).

2.3. Danh sách kề

Để khắc phục nhược điểm của các phương pháp ma trận kề và danh sách cạnh, người ta đề xuất phương pháp biểu diễn đồ thị bằng *danh sách kề* (*adjacency list*). Trong cách biểu diễn này, với mỗi đỉnh v của đồ thị, ta cho tương ứng với nó một danh sách các đỉnh kề với v .

Với đồ thị có hướng $G = (V, E)$. V gồm n đỉnh và E gồm m cung. Có hai cách cài đặt danh sách kề phổ biến:

- Forward Star: Với mỗi đỉnh u , lưu trữ một danh sách $adj[u]$ chứa các đỉnh nối từ u : $adj[u] = \{v: (u, v) \in E\}$.
- Reverse Star: Với mỗi đỉnh v , lưu trữ một danh sách $adj[v]$ chứa các đỉnh nối tới v : $adj[v] = \{u: (u, v) \in E\}$

Tùy theo từng bài toán, chúng ta sẽ chọn cấu trúc Forward Star hoặc Reverse Star để biểu diễn đồ thị. Có những bài toán yêu cầu phải biểu diễn đồ thị bằng cả hai cấu trúc Forward Star và Reverse Star.

Việc biểu diễn đồ thị vô hướng được quy về việc biểu diễn phiên bản có hướng tương ứng: thay mỗi cạnh (u, v) bởi hai cung có hướng ngược nhau: (u, v) và (v, u) .

Bất cứ cấu trúc dữ liệu nào có khả năng biểu diễn danh sách (mảng, danh sách móc nối, cây...) đều có thể sử dụng để biểu diễn danh sách kề, nhưng mảng và danh sách móc nối được sử dụng phổ biến nhất.

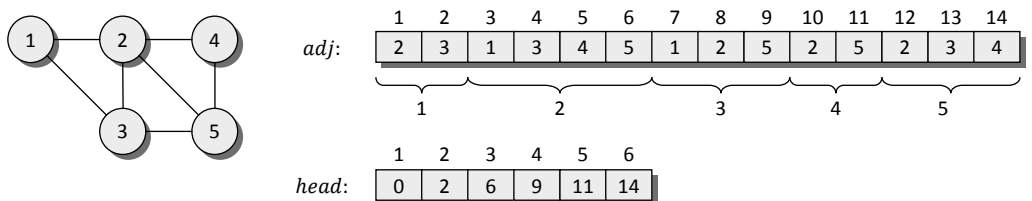
a) Biểu diễn danh sách kề bằng mảng

Dùng một mảng $adj[1 \dots m]$ chứa các đỉnh, mảng được chia làm n đoạn, đoạn thứ u trong mảng lưu danh sách các đỉnh kề với đỉnh u . Để biết một đoạn nằm từ chỉ số nào đến chỉ số nào, ta có một mảng $head[1 \dots n + 1]$ đánh dấu vị trí phân đoạn: $head[u]$ sẽ bằng chỉ số đứng liền trước đoạn thứ u , quy ước $head[n + 1] = m$. Khi đó các phần tử trong đoạn:

$$adj[head[u] + 1 \dots head[u + 1]]$$

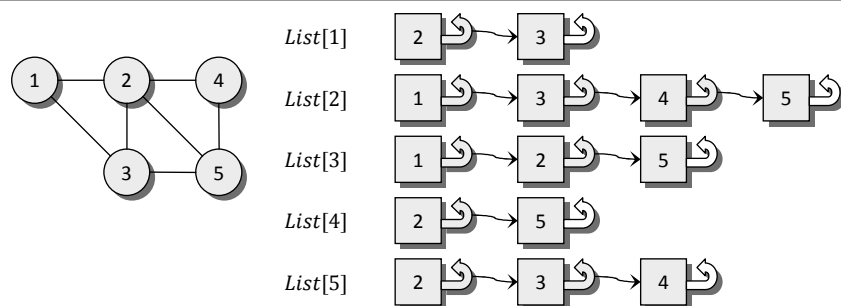
là các đỉnh kề với đỉnh u .

Nhắc lại rằng khi sử dụng danh sách kề để biểu diễn đồ thị vô hướng, ta quy nó về đồ thị có hướng và số cung m được nhân đôi (Hình 5-8).



Hình 5-8. Dùng mảng biểu diễn danh sách kề

b) Biểu diễn danh sách kề bằng các danh sách móc nối



Hình 5-9. Biểu danh sách kề bởi các danh sách móc nối

Trong cách biểu diễn này, ta cho tương ứng mỗi đỉnh u của đồ thị với $List[u]$ là chốt của một danh sách móc nối gồm các đỉnh kề với u .

Ưu điểm của danh sách kề

- Đối với danh sách kề, việc duyệt tất cả các đỉnh kề với một đỉnh v cho trước là hết sức dễ dàng, cái tên “danh sách kề” đã cho thấy rõ điều này.
- Việc duyệt tất cả các cạnh cũng đơn giản vì một cạnh thực ra là nối một đỉnh với một đỉnh khác kề nó.

Nhược điểm của danh sách kề

- Danh sách kề yếu hơn ma trận kề ở việc kiểm tra (u, v) có phải là cạnh hay không, bởi trong cách biểu diễn này ta sẽ phải duyệt toàn bộ danh sách kề của u hay danh sách kề của v .

2.4. Danh sách liên thuộc

Danh sách liên thuộc (incidence lists) là một mở rộng của danh sách kề. Nếu như trong biểu diễn danh sách kề, mỗi đỉnh được cho tương ứng với một danh sách các đỉnh kề thì trong biểu diễn danh sách liên thuộc, mỗi đỉnh được cho tương ứng với một danh sách các cạnh liên thuộc. Chính vì vậy, những kĩ thuật cài đặt danh sách kề có thể sửa đổi một chút để cài đặt danh sách liên thuộc.

Đặc biệt trong trường hợp đồ thị có hướng, ta có thể xây dựng danh sách liên thuộc từ danh sách cạnh tương đối dễ dàng bằng cách bổ sung các con trỏ liên kết. Giả sử đồ thị có hướng $G = (V, E)$ có n đỉnh và m cung được biểu diễn bởi danh sách cạnh $e[1 \dots m]$. Vì đồ thị có hướng, nếu ta cho tương ứng mỗi đỉnh u một danh sách các cung đi ra khỏi u (forward star) thì sẽ có tổng cộng n danh sách liên thuộc và mỗi cung chỉ xuất hiện trong đúng một danh sách liên thuộc. Vì vậy ta có thể bổ sung hai mảng $head[1 \dots n]$ và $link[1 \dots m]$ trong đó:

- $head[u]$ là chỉ số cung đầu tiên trong danh sách liên thuộc của đỉnh u . Nếu danh sách liên thuộc đỉnh u là \emptyset , $head[u]$ được gán bằng 0.
- $link[i]$ là chỉ số cung kế tiếp cung $e[i]$ trong danh sách liên thuộc chứa cung $e[i]$. Trường hợp $e[i]$ là cung cuối cùng của một danh sách liên thuộc, $link[i]$ được gán bằng 0

Để duyệt tất cả những cung đi ra khỏi một đỉnh u nào đó, ta có thể thực hiện dễ dàng bằng thuật toán sau:

```
i := head[u];  
while i ≠ 0 do  
begin
```

```

«Xử lí cung e[i]»;
i := link[i];
end;

```

2.5. Chuyển đổi giữa các cách biểu diễn đồ thị

Có một số thuật toán mà tính hiệu quả của nó phụ thuộc rất nhiều vào cách thức biểu diễn đồ thị, do đó khi bắt tay vào giải quyết một bài toán đồ thị, chúng ta phải tìm cấu trúc dữ liệu phù hợp để biểu diễn đồ thị sao cho hợp lý nhất. Nếu đồ thị đầu vào được cho bởi một cách biểu diễn bất hợp lý, chúng ta cần chuyển đổi cách biểu diễn khác để thuận tiện trong việc triển khai thuật toán.

Ta xét bài toán chuyển đổi các cách biểu diễn đơn đồ thị có hướng $G = (V, E)$ có n đỉnh và m cạnh. Có thể biểu diễn đồ thị này bởi:

Ma trận kề:

```

var
  a: array[1..n, 1..n] of Boolean;

```

Danh sách cạnh:

```

type
  TEdge = record
    x, y: Integer;
  end;
var
  e: array[1..m] of TEdge;

```

Danh sách kề (forward star) (biểu diễn bằng mảng):

```

var
  adj: array[1..2 * m] of Integer;
  head: array[1..n + 1] of Integer;

```

Danh sách liên thuộc (forward star) (biểu diễn bằng cấu trúc liên kết)

```

type
  TEdge = record
    x, y: Integer;
  end;
var
  e: array[1..m] of TEdge; //Danh sách cạnh
  link: array[1..m] of Integer; //link[i]: chỉ số cạnh kế tiếp trong danh sách
  //liên thuộc
  head: array[1..n] of Integer; //head[i]: chỉ số cạnh đầu
  //tiên trong danh sách liên thuộc

```

a) Chuyển đổi giữa ma trận kề và danh sách cạnh

Nếu đồ thị G được cho bởi ma trận kề $A = \{a_{ij}\}_{n \times n}$ trong đó $a_{ij} = \text{True} \Leftrightarrow (i, j) \in E$, ta có thể xây dựng danh sách cạnh tương ứng bằng cách duyệt tất cả các cặp (i, j) , nếu $a_{ij} = \text{True}$ thì đưa cặp này vào danh sách cạnh e .

```
k := 0;
for i := 1 to n do
  for j := 1 to n do
    if a[i, j] then
      begin
        k := k + 1;
        e[k].x := i; e[k].y := j;
      end;
```

Ngược lại, nếu đồ thị G cho bởi danh sách cạnh e , ta có thể xây dựng ma trận kề A bằng cách khởi tạo các phần tử của A là False rồi duyệt danh sách cạnh, mỗi khi duyệt qua cung (x, y) , ta đặt $a_{xy} := \text{True}$.

```
for i := 1 to n do
  for j := 1 to n do a[i, j] := False;
for k := 1 to m do
  with e[k] do
    a[x, y] := True;
```

b) Chuyển đổi giữa ma trận kề và danh sách kề

Từ ma trận kề $A = \{a_{ij}\}_{n \times n}$, ta có thể xây dựng hai mảng $adj[1 \dots m]$ và $head[1 \dots n + 1]$ sao cho các phần tử trong mảng adj từ chỉ số $head[u] + 1$ tới chỉ số $head[u + 1]$ chứa danh sách kề của đỉnh u . Hai mảng này là danh sách kề dạng forward star của đồ thị.

```
head[n + 1] := m;
for i := n downto 1 do
  begin
    head[i] := head[i + 1];
    for j := n downto 1 do
      if a[i, j] then
        begin
          adj[head[i]] := j;
          head[i] := head[i] - 1;
        end;
    end;
```

Ngược lại, chúng ta có thể xây dựng ma trận kề từ danh sách kề theo cách: Đặt các phần tử của ma trận kề A bằng $False$, sau đó với mỗi đỉnh u , duyệt các đỉnh v thuộc danh sách kề của nó và đặt $a_{uv} := True$.

```
for i := 1 to n do
  for j := 1 to n do a[i, j] := False;
for u := 1 to n do
  for k := head[u] + 1 to head[u + 1] do
    a[u, adj[k]] := True;
```

c) Chuyển đổi giữa danh sách cạnh và danh sách kề

Từ danh sách cạnh e , ta có thể xây dựng hai mảng adj và $head$ tương ứng với danh sách kề dạng forward star bằng thuật toán đếm phân phối.

Trước hết, ta tính các $head[u]$ là bậc của đỉnh u ($\forall u \in V$):

```
for u := 1 to n do head[u] := 0;
for i := 1 to m do
  with e[i] do
    head[x] := head[x] + 1;
```

Sau đó, ta chia mảng adj thành n đoạn, đoạn thứ u sẽ chứa các đỉnh kề với đỉnh u . Để xác định vị trí các đoạn này, ta đặt mỗi $head[u]$ trở tới vị trí cuối đoạn thứ u :

```
for u := 2 to n do
  head[u] := head[u - 1] + head[u];
```

Tiếp theo là duyệt lại danh sách cạnh, mỗi khi duyệt tới cạnh (x, y) ta đưa y vào mảng adj tại vị trí $head[x]$, đưa x vào mảng adj tại vị trí $head[y]$ đồng thời giảm hai con trỏ $head[x]$ và $head[y]$ đi 1.

```
for i := m downto 1 do
  with e[i] do
    begin
      adj[head[x]] := y; head[x] := head[x] - 1;
    end;
```

Đến đây, chúng ta có mảng adj phân làm n đoạn, trong đó $head[u]$ là vị trí đứng liền trước đoạn thứ u . Việc cuối cùng là đặt:

```
head[n + 1] := m;
```

Việc chuyển đổi từ danh sách kề sang danh sách cạnh được thực hiện đơn giản hơn: Với mỗi đỉnh u , ta xét các đỉnh v thuộc danh sách kề của nó và đưa (u, v) vào danh sách cạnh e .

```

i := 0;
for u := 1 to n do
  for k := head[u] + 1 to head[u + 1] do
    begin
      v := adj[k];
      i := i + 1;
      e[i].x := u; e[i].y := v;
    end;
  end;
end;

```

d) Chuyển đổi giữa danh sách cạnh và danh sách liên thuộc

Bởi danh sách liên thuộc được đặc tả đã bao gồm danh sách cạnh, ta chỉ quan tâm tới vấn đề chuyển đổi từ danh sách cạnh thành danh sách liên thuộc.

Trước hết với mọi đỉnh u ta đặt $head[u] := 0$ để khởi tạo danh sách liên thuộc của u bằng \emptyset .

```

for u := 1 to n do head[u] := 0;

```

Tiếp theo ta duyệt danh sách cạnh, mỗi khi duyệt qua một cung (x, y) ta móc nối cung này vào danh sách liên thuộc các cung đi ra khỏi x :

```

for i := m downto 1 do
  with e[i] do
    begin
      link[i] := head[x];
      head[x] := i;
    end;
  end;
end;

```

Với các bài toán mà chúng ta sẽ khảo sát, cũng có một số thuật toán không phụ thuộc nhiều và cách biểu diễn đồ thị, trong trường hợp này tôi sẽ chọn cấu trúc dữ liệu dễ cài đặt và trình bày nhất để việc đọc hiểu thuật toán/chương trình được thuận tiện hơn.

Bài tập

- 5.1. Cho một đồ thị có hướng n đỉnh, m cạnh được biểu diễn bằng danh sách kề, trong đó mỗi đỉnh u sẽ được cho tương ứng với một danh sách các đỉnh nối từ u . Cho một đỉnh v , hãy tìm thuật toán tính bán bậc ra và bán bậc vào của v . Xác định độ phức tạp tính toán của thuật toán
- 5.2. Đồ thị chuyển vị của đồ thị có hướng $G = (V, E)$ là đồ thị $G^T = (V, E^T)$, trong đó:

$$E^T = \{(u, v) : (v, u) \in E\}$$

Hãy tìm thuật toán xây dựng G^T từ G trong hai trường hợp: G và G^T được biểu diễn bằng ma trận kề; G và G^T được biểu diễn bằng danh sách kề.

- 5.3.** Cho đa đồ thị vô hướng $G = (V, E)$ được biểu diễn bằng danh sách kề, hãy tìm thuật toán $O(|V| + |E|)$ để xây dựng đơn đồ thị $G' = (V, E')$ và biểu diễn G' bằng danh sách kề, biết rằng đồ thị G' gồm tất cả các đỉnh của đồ thị G và các cạnh song song trên G được thay thế bằng duy nhất một cạnh trong G' .
- 5.4.** Cho đa đồ thị G được biểu diễn bằng ma trận kề $A = \{a_{ij}\}$ trong đó a_{ij} là số cạnh nối từ đỉnh i tới đỉnh j . Hãy chứng minh rằng A^k là ma trận $B = \{b_{ij}\}$ trong đó b_{ij} là số đường đi từ đỉnh i tới đỉnh j qua đúng k cạnh. Gợi ý: Sử dụng chứng minh quy nạp.
- 5.5.** Cho đơn đồ thị $G = (V, E)$, ta gọi bình phương của một đồ thị G là đơn đồ thị

$$G^2 = (V, E^2)$$

sao cho $(u, v) \in E^2$ nếu và chỉ nếu tồn tại một đỉnh $w \in V$ sao cho (u, w) và (w, v) đều thuộc E . Hãy tìm thuật toán $O(|V|^3)$ để xây dựng G^2 từ G trong trường hợp cả G và G^2 được biểu diễn bằng ma trận kề, tìm thuật toán $O(|E| + |V|^2)$ để xây dựng G^2 từ G trong trường hợp cả G và G^2 được biểu diễn bằng danh sách kề.

- 5.6.** Xây dựng cấu trúc dữ liệu để biểu diễn đồ thị vô hướng và các thao tác:
- Liệt kê các đỉnh kề với một đỉnh cho trước trong thời gian $O(|E|)$
 - Kiểm tra hai đỉnh có kề nhau hay không trong thời gian $O(1)$
 - Loại bỏ một cạnh trong thời gian $O(1)$
- 5.7.** Với đồ thị $G = (V, E)$ được biểu diễn bằng ma trận kề, đa số các thuật toán trên đồ thị sẽ có độ phức tạp tính toán $\Omega(|V|^2)$, tuy nhiên không phải không có ngoại lệ. Chẳng hạn bài toán tìm “bồn chứa” (*universal sink*) trong đồ thị: bồn chứa trong đồ thị có hướng là một đỉnh nối từ tất cả các đỉnh khác và không có cung đi ra. Hãy tìm thuật toán $O(|V|)$ để xác định sự tồn tại và chỉ ra bồn chứa trong đồ thị có hướng.
- 5.8.** Người ta còn có thể biểu diễn đồ thị bằng *ma trận liên thuộc* (*incidence matrix*): Với đồ thị có hướng $G = (V, E)$ có n đỉnh và m cung, ma trận liên thuộc $B = \{b_{ij}\}$ của G kích thước $m \times n$, trong đó:

$$b_{ij} = \begin{cases} -1, & \text{nếu cung thứ } j \text{ đi ra khỏi đỉnh } i \\ 1, & \text{nếu cung thứ } j \text{ đi vào đỉnh } i \\ 0, & \text{nếu cung thứ } j \text{ không liên thuộc với đỉnh } i \end{cases}$$

Xét B^T là ma trận chuyển vị của ma trận B , hãy cho biết ý nghĩa của ma trận tích BB^T

3. Các thuật toán tìm kiếm trên đồ thị

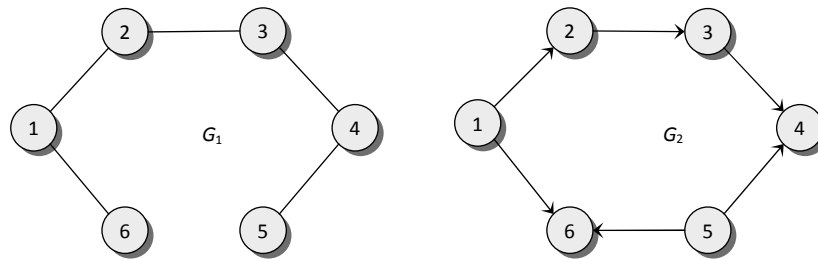
3.1. Bài toán tìm đường

Cho đồ thị $G = (V, E)$ và hai đỉnh $s, t \in V$.

Nhắc lại định nghĩa đường đi: Một dãy các đỉnh:

$$P = \langle s = p_0, p_1, \dots, p_k = t \rangle, (\forall i: (p_{i-1}, p_i) \in E)$$

được gọi là một đường đi từ s tới t , đường đi này gồm $k + 1$ đỉnh p_0, p_1, \dots, p_k và k cạnh $(p_0, p_1), (p_1, p_2), \dots, (p_{k-1}, p_k)$. Đỉnh s được gọi là đỉnh đầu và đỉnh t được gọi là đỉnh cuối của đường đi. Nếu tồn tại một đường đi từ s tới t , ta nói s đến được t và t đến được từ s : $s \rightsquigarrow t$.



Hình 5-10: Đồ thị và đường đi

Trên cả hai đồ thị ở Hình 5-10, $\langle 1,2,3,4 \rangle$ là đường đi từ đỉnh 1 tới đỉnh 4. $\langle 1,6,5,4 \rangle$ không phải đường đi vì không có cạnh (cung) $(6,5)$.

Một bài toán quan trọng trong lý thuyết đồ thị là bài toán duyệt tất cả các đỉnh có thể đến được từ một đỉnh xuất phát nào đó. Vấn đề này đưa về một bài toán liệt kê mà yêu cầu của nó là không được bỏ sót hay lặp lại bất kì đỉnh nào. Chính vì vậy mà ta phải xây dựng những thuật toán cho phép duyệt một cách hệ thống các đỉnh, những thuật toán như vậy gọi là những thuật toán tìm kiếm trên đồ thị (*graph traversal*). Ta quan tâm đến hai thuật toán cơ bản nhất: thuật toán tìm kiếm theo chiều sâu và thuật toán tìm kiếm theo chiều rộng.

Trong những chương trình cài đặt dưới đây, ta giả thiết rằng đồ thị được cho là đồ thị có hướng, số đỉnh không quá 10^5 , số cung không quá 10^6 , các đỉnh được đánh

số từ 1 tới n và đồng nhất với số hiệu của chúng. Khuôn dạng Input/Output quy định cụ thể như sau:

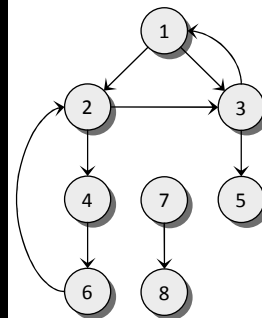
Input

- Dòng 1 chứa số đỉnh n , đỉnh xuất phát s và đỉnh cần đến t .
- n dòng tiếp theo, dòng thứ i chứa một danh sách các đỉnh, mỗi đỉnh j trong danh sách tương ứng với một cung (i, j) của đồ thị, ngoài ra có thêm một số 0 ở cuối dòng để báo hiệu kết thúc.

Output

- Danh sách các đỉnh có thể đến được từ s
- Đường đi từ s tới t nếu có

Sample Input	Sample Output
8 1 5	Reachable vertices from 1:
2 3 0	1, 2, 3, 5, 4, 6,
3 4 0	The path from 1 to 5:
1 5 0	5<-3<-2<-1
6 0	
0	
2 0	
8 0	
0	



3.2. Biểu diễn đồ thị

Đồ thị được biểu diễn bằng danh sách kề dạng forward star, mỗi đỉnh u sẽ được cho tương ứng với một danh sách các đỉnh nối từ u . Nếu đồ thị có n đỉnh thì có tổng cộng n danh sách kề, gọi m là tổng số phần tử trên tất cả các danh sách kề. Khi đó $m = |E|$, như đã quy ước, $m \leq 10^6$.

Cấu trúc dữ liệu được cài đặt bằng mảng $adj[1 \dots m]$ mảng này được chia làm n đoạn liên tiếp, đoạn thứ u chứa danh sách các đỉnh nối từ u . Vị trí của các đoạn được xác định bởi mảng $head[0 \dots n]$ trong đó $head[u]$ là vị trí cuối đoạn thứ u , quy ước $head[0] = 0$. Như vậy các đỉnh nối từ u sẽ nằm liên tiếp trong mảng adj từ chỉ số $head[u - 1] + 1$ tới chỉ số $head[u]$.

3.3. Thuật toán tìm kiếm theo chiều sâu

a) Ý tưởng

Tư tưởng của thuật toán tìm kiếm theo chiều sâu (Depth-First Search – DFS) có thể trình bày như sau: Trước hết, dĩ nhiên đỉnh s đến được từ s , tiếp theo, với mọi cung (s, x) của đồ thị thì x cũng sẽ đến được từ s . Với mỗi đỉnh x đó thì tất nhiên những đỉnh y nối từ x cũng đến được từ s ... Điều đó gợi ý cho ta viết một thủ tục đệ quy $DFSVisit(u)$ mô tả việc duyệt từ đỉnh u bằng cách thăm đỉnh u và tiếp tục quá trình duyệt $DFSVisit(v)$ với v là một đỉnh chưa thăm nối từ u .

Kĩ thuật đánh dấu được sử dụng để tránh việc liệt kê lặp các đỉnh: Khởi tạo $avail[v] := \text{True}, \forall v \in V$, mỗi lần thăm một đỉnh, ta đánh dấu đỉnh đó lại ($avail[v] := \text{False}$) để các bước duyệt đệ quy kế tiếp không duyệt lại đỉnh đó nữa. Để lưu lại đường đi từ đỉnh xuất phát s , trong thủ tục $DFSVisit(u)$, trước khi gọi đệ quy $DFSVisit(v)$ với v là một đỉnh chưa thăm nối từ u (chưa đánh dấu), ta lưu lại vết đường đi từ u tới v bằng cách đặt $trace[v] := u$, tức là $trace[v]$ lưu lại đỉnh liền trước v trong đường đi từ s tới v . Khi thuật toán DFS kết thúc, đường đi từ s tới t sẽ là:

$$\langle p_1 = t \leftarrow p_2 = trace[p_1] \leftarrow p_3 = trace[p_2] \leftarrow \dots \leftarrow s \rangle$$

```
procedure DFSVisit(u ∈ V); //Thuật toán tìm kiếm theo chiều sâu từ đỉnh u
begin
    avail[u] := False; //avail[u] = False ⇔ u đã thăm
    Output ← u; //Liệt kê u
    for ∀v ∈ V: (u, v) ∈ E do //Duyệt mọi đỉnh v chưa thăm nối từ u
        if avail[v] then
            begin
                trace[v] := u; //Lưu vết đường đi, đỉnh liền trước v trên đường đi
                //từ s tới v là u
                DFSVisit(v); //Gọi đệ quy để tìm kiếm theo chiều sâu từ đỉnh v
            end;
    end;
begin //Chương trình chính
    Input → Đồ thị G, đỉnh xuất phát s, đỉnh đích t;
    for ∀v ∈ V do avail[v] := True; //Đánh dấu mọi đỉnh đều chưa thăm
    DFSVisit(s);
    if avail[t] then //s đi tới được t
        «Truy theo vết từ t để tìm đường đi từ s tới t»;
    end.
```

b) Cài đặt



DFS.PAS ✓ Tìm đường bằng DFS

```
{ $MODE OBJFPC }
{ $M 4000000 }
program DepthFirstSearch;
const
  maxN = 100000;
  maxM = 1000000;
var
  adj: array[1..maxM] of Integer; //Các danh sách kề
  head: array[0..maxN] of Integer; //Mảng đánh dấu vị trí cắt đoạn trong
adj
  avail: array[1..maxN] of Boolean;
  trace: array[1..maxN] of Integer;
  n, s, t: Integer;
procedure Enter; //Nhập dữ liệu
var
  u, v, i: Integer;
begin
  ReadLn(n, s, t);
  i := 0;
  for u := 1 to n do
    begin //Đọc danh sách kề của u
      repeat
        read(v);
        if v <> 0 then //Thêm v vào mảng adj
          begin
            Inc(i); adj[i] := v;
          end;
      until v = 0;
      head[u] := i; //Đọc hết một dòng, đánh dấu vị trí cắt đoạn thứ u
      ReadLn;
    end;
  head[0] := 0; //Cắm canh
end;
procedure DFSVisit(u: Integer); //Thuật toán tìm kiếm theo chiều sâu bắt
đầu từ u
var
  i: Integer;
begin
  avail[u] := False;
```

```

Write(u, ' ', ' '); //Liệt kê u
for i := head[u - 1] + 1 to head[u] do //Duyệt các đỉnh adj[i] nối từ u
  if avail[adj[i]] then
    begin
      trace[adj[i]] := u;
      DFSVisit(adj[i]);
    end;
end;
procedure PrintPath; //In đường đi từ s tới t
begin
  if avail[t] then //Từ s không có đường tới t
    WriteLn(' There is no path from ', s, ' to ', t)
  else
    begin
      WriteLn('The path from ', s, ' to ', t, ':');
      while t <> s do //Truy vết ngược từ t về s
        begin
          Write(t, '<-');
          t := trace[t];
        end;
      WriteLn(s);
    end;
end;
begin
  Enter;
  FillChar(avail[1], n * SizeOf(avail[1]), True);
  WriteLn('Reachable vertices from ', s, ': ');
  DFSVisit(s);
  WriteLn;
  PrintPath;
end.

```

Có thể không cần mảng đánh dấu $avail[1 \dots n]$ mà dùng luôn mảng $trace[1 \dots n]$ để đánh dấu. Khởi tạo các phần tử mảng $trace[1 \dots n]$ là:

$$\begin{cases} trace[s] \neq 0 \\ trace[v] = 0, \forall v \neq s \end{cases}$$

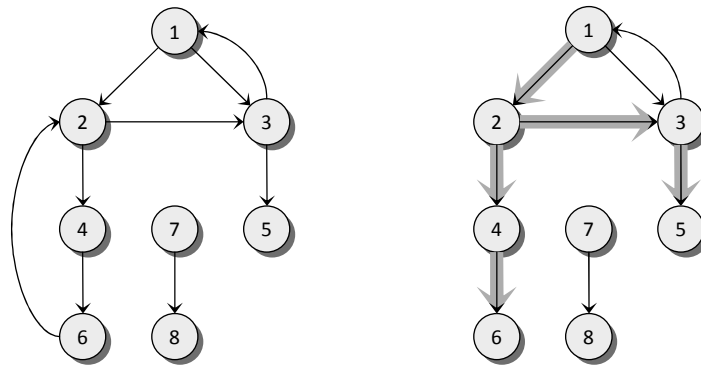
Khi đó điều kiện để một đỉnh v chưa thăm là $trace[v] = 0$, mỗi khi từ đỉnh u thăm đỉnh v , phép gán $trace[v] := u$ sẽ kiêm luôn công việc đánh dấu v đã thăm ($trace[v] \neq 0$).

Một vài tính chất của DFS

Cây DFS

Nếu ta sắp xếp danh sách kề của mỗi đỉnh theo thứ tự tăng dần thì thuật toán DFS luôn trả về đường đi có thứ tự từ điển nhỏ nhất trong số tất cả các đường đi từ s tới t .

Quá trình tìm kiếm theo chiều sâu cho ta một cây DFS gốc s . Quan hệ cha-con trên cây được định nghĩa là: nếu từ đỉnh u tới thăm đỉnh v ($DFSVisit(u)$ gọi $DFSVisit(v)$) thì u là nút cha của nút v . Hình 5-11 là đồ thị và cây DFS tương ứng với đỉnh xuất phát $s = 1$.



Hình 5-11: Đồ thị và cây DFS

Mô hình duyệt đồ thị theo DFS

Cài đặt trên chỉ là một ứng dụng của thuật toán DFS để liệt kê các đỉnh đến được từ một đỉnh. Thuật toán DFS dùng để duyệt qua các đỉnh và các cạnh của đồ thị được viết theo mô hình sau:

```
procedure DFSVisit( $u \in V$ ) ; //Thuật toán tìm kiếm theo chiều sâu từ đỉnh  $u$ 
begin
    Time := Time + 1;
    d[u] := Time;
    Output  $\leftarrow u$  ; //Liệt kê  $u$ 
    for  $\forall v \in V: (u, v) \in E$  do //Duyệt mọi đỉnh  $v$  nối từ  $u$ 
        if d[v] = 0 then DFSVisit(v) ; //Nếu  $v$  chưa thăm, gọi đệ quy để tìm
        kiểm theo chiều sâu từ đỉnh  $v$ 
    Time := Time + 1;
    f[u] := Time;
end;
begin //Chương trình chính
    Input  $\rightarrow$  Đồ thị  $G$ 
```

```

for  $\forall v \in V$  do  $d[v] := 0$ ; //Mọi đỉnh đều chưa được duyệt đến
Time := 0;
for  $\forall v \in V$  do
    if  $d[v] = 0$  then DFSVisit( $v$ );
end.

```

Thuật toán này sẽ thăm tất cả các đỉnh và các cạnh của đồ thị và thứ tự thăm được gọi là thứ tự duyệt DFS. Như ví dụ ở đồ thị trong bài, thứ tự thăm DFS với các đỉnh là:

1, 2, 3, 5, 4, 6, 7, 8

Thứ tự thăm DFS với các cạnh là:

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

Thời gian thực hiện giải thuật của DFS có thể đánh giá bằng số lần gọi thủ tục *DFSVisit* ($|V|$ lần) cộng với số lần thực hiện của vòng lặp for bên trong thủ tục *DFSVisit*. Chính vì vậy:

- Nếu đồ thị được biểu diễn bằng danh sách kề hoặc danh sách liên thuộc, vòng lặp for bên trong thủ tục *DFSVisit* (xét tổng thể cả chương trình) sẽ duyệt qua tất cả các cạnh của đồ thị (mỗi cạnh hai lần nếu là đồ thị vô hướng, mỗi cạnh một lần nếu là đồ thị có hướng). Trong trường hợp này, thời gian thực hiện giải thuật DFS là $\Theta(|V| + |E|)$
- Nếu đồ thị được biểu diễn bằng ma trận kề, vòng lặp for bên trong mỗi thủ tục *DFSVisit* sẽ phải duyệt qua tất cả các đỉnh $1 \dots n$. Trong trường hợp này thời gian thực hiện giải thuật DFS là $\Theta(|V| + |V|^2) = \Theta(|V|^2)$.
- Nếu đồ thị được biểu diễn bằng danh sách cạnh, vòng lặp for bên trong thủ tục *DFSVisit* sẽ phải duyệt qua tất cả danh sách cạnh mỗi lần thực hiện thủ tục. Trong trường hợp này thời gian thực hiện giải thuật DFS là $\Theta(|V||E|)$.

Thứ tự duyệt đến và duyệt xong

Hãy để ý thủ tục *DFSVisit*(u):

- Khi bắt đầu vào thủ tục ta nói đỉnh u được *duyet đến* hay được *thăm* (*discover*), có nghĩa là tại thời điểm đó, quá trình tìm kiếm theo chiều sâu bắt đầu từ u sẽ xây dựng nhánh cây DFS gốc u .
- Khi chuẩn bị thoát khỏi thủ tục để lùi về, ta nói đỉnh u được *duyet xong* (*finish*), có nghĩa là tại thời điểm đó, quá trình tìm kiếm theo chiều sâu từ u kết thúc.

Trong mô hình duyệt DFS ở trên, chúng ta sử dụng một biến đếm *Time* để xác định thời điểm duyệt đến d_u và thời điểm duyệt xong f_u của mỗi đỉnh u . Thứ tự

duyet đến và duyệt xong này có ý nghĩa rất quan trọng trong nhiều thuật toán có áp dụng DFS, chẳng hạn như các thuật toán tìm thành phần liên thông mạnh, thuật toán sắp xếp tô pô...

Định lí 5-6

Với hai đỉnh phân biệt u, v :

- Đỉnh v được duyệt đến trong thời gian từ d_u đến f_u : $d[v] \in [d_u, f_u]$ nếu và chỉ nếu v là hậu duệ của u trên cây DFS.
- Đỉnh v được duyệt xong trong thời gian từ d_u đến f_u :
 $f[v] \in [d_u, f_u]$ nếu và chỉ nếu v là hậu duệ của u trên cây DFS.

Chứng minh

Bản chất của việc đỉnh v được duyệt đến (hay duyệt xong) trong thời gian từ d_u đến f_u chính là thủ tục $DFSVisit(v)$ được gọi (hay thoát) khi mà thủ tục $DFSVisit(u)$ đã bắt đầu nhưng chưa kết thúc, nghĩa là thủ tục $DFSVisit(v)$ được dây chuyền đệ quy từ $DFSVisit(u)$ gọi tới. Điều này chỉ ra rằng v nằm trong nhánh DFS gốc u , hay nói cách khác, v là hậu duệ của u .

Hệ quả

Với hai đỉnh phân biệt (u, v) thì hai đoạn $[d_u, f_u]$ và $[d_v, f_v]$ hoặc rời nhau hoặc chứa nhau. Hai đoạn $[d_u, f_u]$ và $[d_v, f_v]$ chứa nhau nếu và chỉ nếu u và v có quan hệ tiền bối–hậu duệ.

Chứng minh

Dễ thấy rằng nếu hai đoạn $[d_u, f_u]$ và $[d_v, f_v]$ không rời nhau thì hoặc $d_u \in [d_v, f_v]$ hoặc $d_v \in [d_u, f_u]$, tức là hai đỉnh (u, v) có quan hệ tiền bối–hậu duệ, áp dụng Định lí 5-6, ta có ĐPCM.

Định lí 5-7

Với hai đỉnh phân biệt $u \neq v$ mà $(u, v) \in E$ thì v phải được duyệt đến trước khi u được duyệt xong:

$$(u, v) \in E \Rightarrow d_v < f_u \quad (0.1)$$

Chứng minh

Đây là một tính chất quan trọng của thuật toán DFS. Hãy để ý thủ tục $DFSVisit(u)$, trước khi thoát (duyet xong u), nó sẽ quét tất cả các đỉnh chưa thăm nối từ u và gọi đệ quy để thăm những đỉnh đó, tức là v phải được duyệt đến trước khi u được duyệt xong: $d_v < f_u$.

Định lí 5-8 (định lí đường đi trắng)

Đỉnh v là hậu duệ thực sự của đỉnh u trong một cây DFS nếu và chỉ nếu tại thời điểm d_u mà thuật toán thăm tới đỉnh u , tồn tại một đường đi từ u

tới v mà ngoại trừ đỉnh u , tất cả các đỉnh khác trên đường đi đều chưa được thăm.

Chứng minh

“ \Rightarrow ”

Nếu v là hậu duệ của u , ta xét đường đi từ u tới v dọc trên các cung trên cây DFS. Tất cả các đỉnh w nằm sau u trên đường đi này đều là hậu duệ của u , nên theo Định lí 5-6, ta có $d_u < d_w$, tức là vào thời điểm d_u , tất cả các đỉnh w đó đều chưa được thăm

“ \Leftarrow ”

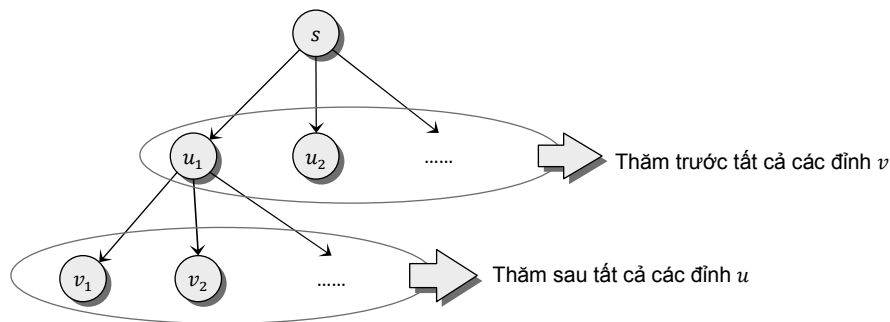
Nếu tại thời điểm d_u , tồn tại một đường đi từ u tới v mà ngoại trừ đỉnh u , tất cả các đỉnh khác trên đường đi đều chưa được thăm, ta sẽ chứng minh rằng mọi đỉnh trên đường đi này đều là hậu duệ của u . Thật vậy, giả sử phản chứng rằng v^* là đỉnh đầu tiên trên đường đi này mà không phải hậu duệ của u , tức là tồn tại đỉnh w liền trước v^* trên đường đi là hậu duệ của u . Theo Định lí 5-7, v^* phải được thăm trước khi duyệt xong w : $d_{v^*} < f_w$; w lại là hậu duệ của u nên theo Định lí 5-6, ta có $f_w \leq f_u$, vậy $d_{v^*} < f_u$. Mặt khác theo giả thiết rằng tại thời điểm d_u thì v^* chưa được thăm, tức là $d_u < d_{v^*}$, kết hợp lại ta có $d_{v^*} \in [d_w, f_u]$, vậy thì v^* là hậu duệ của u theo Định lí 5-6, trái với giả thiết phản chứng.

Tên gọi “định lí đường đi trắng: white-path theorem” xuất phát từ cách trình bày thuật toán DFS bằng cơ chế tô màu đồ thị: Ban đầu các đỉnh được tô màu trắng, mỗi khi duyệt đến một đỉnh thì đỉnh đó được tô màu xám và mỗi khi duyệt xong một đỉnh thì đỉnh đó được tô màu đen: Định lí khi đó có thể phát biểu: Điều kiện cần và đủ để đỉnh v là hậu duệ thực sự của đỉnh u trong một cây DFS là tại thời điểm đỉnh u được tô màu xám, tồn tại một đường đi từ u tới v mà ngoại trừ đỉnh u , tất cả các đỉnh khác trên đường đi đều có màu trắng.

3.4. Thuật toán tìm kiếm theo chiều rộng

a) Ý tưởng

Tư tưởng của thuật toán tìm kiếm theo chiều rộng (Breadth-First Search – BFS) là “lập lịch” duyệt các đỉnh. Việc thăm một đỉnh sẽ lên lịch duyệt các đỉnh nối từ nó sao cho thứ tự duyệt là ưu tiên chiều rộng (đỉnh nào gần đỉnh xuất phát s hơn sẽ được duyệt trước). Đầu tiên ta thăm đỉnh s . Việc thăm đỉnh s sẽ phát sinh thứ tự thăm những đỉnh u_1, u_2, \dots nối từ s (những đỉnh gần s nhất). Tiếp theo ta thăm đỉnh u_1 , khi thăm đỉnh u_1 sẽ lại phát sinh yêu cầu thăm những đỉnh v_1, v_2, \dots nối từ u_1 . Nhưng rõ ràng các đỉnh v này “xa” s hơn những đỉnh u nên chúng chỉ được thăm khi tất cả những đỉnh u đã thăm. Tức là thứ tự duyệt đỉnh sẽ là: $s, u_1, u_2, \dots, v_1, v_2, \dots$



Hình 5-12: Thứ tự thăm đỉnh của BFS

Thuật toán tìm kiếm theo chiều rộng sử dụng một danh sách để chứa những đỉnh đang “chờ” thăm. Tại mỗi bước, ta thăm một đỉnh đầu danh sách, loại nó ra khỏi danh sách và cho những đỉnh chưa “xếp hàng” kề với nó xếp hàng thêm vào cuối danh sách. Thuật toán sẽ kết thúc khi danh sách rỗng.

Vì nguyên tắc vào trước ra trước, danh sách chứa những đỉnh đang chờ thăm được tổ chức dưới dạng hàng đợi (Queue): Nếu ta có *Queue* là một hàng đợi với thủ tục *Push(v)* để đẩy một đỉnh *v* vào hàng đợi và hàm *Pop* trả về một đỉnh lấy ra từ hàng đợi thì mô hình của giải thuật BFS có thể viết như sau:

```

Queue := (s); //Khởi tạo hàng đợi chỉ gồm một đỉnh s
for  $\forall v \in V$  do
    avail[v] := True;
avail[s] := False; //Đánh dấu chỉ có đỉnh s được xếp hàng
repeat //Lặp tới khi hàng đợi rỗng
    u := Pop; //Lấy từ hàng đợi ra một đỉnh u
    Output  $\leftarrow$  u; //Liệt kê u
    for  $\forall v \in V$ : avail[v] and (u, v)  $\in E$  do //Xét những đỉnh v kề u
        chưa được đẩy vào hàng đợi
        begin
            trace[v] := u; //Lưu vết đường đi
            Push(v); //Đẩy v vào hàng đợi
            avail[v] := False; //Đánh dấu v đã xếp hàng
        end;
until Queue =  $\emptyset$ ;
if avail[t] then //s đi tới được t
    «Truy theo vết từ t để tìm đường đi từ s tới t»;

```


b) Cài đặt



BFS.PAS ✓ Tìm đường bằng BFS

```
{ $MODE OBJFPC }
program Breadth_First_Search;
const
  maxN = 1000000;
  maxM = 10000000;
var
  adj: array[1..maxM] of Integer; //Các danh sách kề
  head: array[0..maxN] of Integer; //Mảng đánh dấu vị trí cắt đoạn trong adj
  avail: array[1..maxN] of Boolean;
  trace: array[1..maxN] of Integer;
  n, s, t: Integer;
  Queue: array[1..maxN] of Integer;
  front, rear: Integer;
procedure Enter; //Nhập dữ liệu
var
  u, v, i: Integer;
begin
  ReadLn(n, s, t);
  i := 0;
  for u := 1 to n do
    begin //Đọc danh sách kề của u
      repeat
        read(v);
        if v <> 0 then //Thêm v vào mảng adj
          begin
            Inc(i); adj[i] := v;
          end;
      until v = 0;
      head[u] := i; //Đọc hết một dòng, đánh dấu vị trí cắt đoạn thứ u
      ReadLn;
    end;
  head[0] := 0; //Cắm canh
end;
procedure BFS; //Thuật toán tìm kiếm theo chiều rộng
var
  u, i: Integer;
begin
  front := 1; rear := 1; //front: chỉ số đầu hàng đợi; rear: chỉ số cuối hàng đợi
  Queue[1] := s; //Khởi tạo hàng đợi ban đầu chỉ có mỗi một đỉnh s
```

```

    FillChar(avail[1], n * SizeOf(avail[1]), True); //Các đỉnh đều
chưa xếp hàng
    avail[s] := False; //ngoại trừ đỉnh s đã xếp hàng
    repeat
        u := Queue[front]; Inc(front); //Lấy từ hàng đợi ra một đỉnh u
        Write(u, ' '); //Liệt kê u
        for i := head[u - 1] + 1 to head[u] do //Duyệt những đỉnh adj[i]
nối từ u
            if avail[adj[i]] then //Nếu đỉnh đó chưa thăm
                begin
                    Inc(rear); Queue[rear] := adj[i]; //Đẩy vào hàng đợi
                    avail[adj[i]] := False;
                    trace[adj[i]] := u; //Lưu vết đường đi
                end;
        until front > rear;
    end;
    procedure PrintPath; //In đường đi từ s tới t
    begin
        if avail[t] then //Từ s không có đường tới t
            WriteLn(' There is no path from ', s, ' to ', t)
        else
            begin
                WriteLn('The path from ', s, ' to ', t, ':');
                while t <> s do //Truy vết ngược từ t về s
                    begin
                        Write(t, '<-');
                        t := trace[t];
                    end;
                WriteLn(s);
            end;
    end;
    begin
        Enter;
        WriteLn('Reachable vertices from ', s, ': ');
        BFS;
        WriteLn;
        PrintPath;
    end.

```

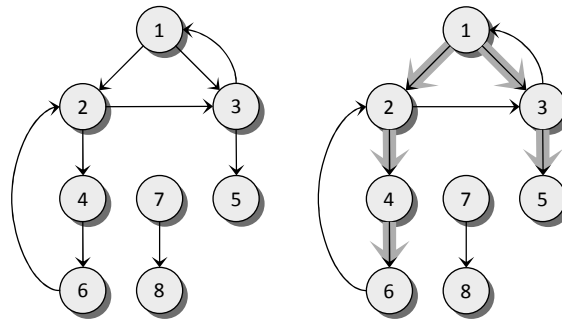
Tương tự như thuật toán tìm kiếm theo chiều sâu, ta có thể dùng mảng *Trace*[1 ... *n*] kiểm tra luôn chức năng đánh dấu.

c) Một vài tính chất của BFS

Cây BFS

Nếu ta sắp xếp các danh sách kề của mỗi đỉnh theo thứ tự tăng dần thì thuật toán BFS luôn trả về đường đi qua ít cạnh nhất trong số tất cả các đường đi từ s tới t . Nếu có nhiều đường đi từ s tới t đều qua ít cạnh nhất thì thuật toán BFS sẽ trả về đường đi có thứ tự từ điển nhỏ nhất trong số những đường đi đó.

Quá trình tìm kiếm theo chiều rộng cho ta một cây BFS gốc s . Khi thuật toán kết thúc $Trace[v]$ chính là nút cha của nút v trên cây. Hình 5-13 là đồ thị và cây BFS tương ứng với đỉnh xuất phát $s = 1$.



Hình 5-13: Đồ thị và cây BFS

Mô hình duyệt đồ thị theo BFS

Tương tự như thuật toán DFS, trên thực tế, thuật toán BFS cũng dùng để xác định một thứ tự trên các đỉnh của đồ thị và được viết theo mô hình sau:

```
procedure BFSVisit(sEV);
begin
  Queue := (s); //Khởi tạo hàng đợi chỉ gồm một đỉnh s
  Time := Time + 1;
  d[s] := Time; //Duyệt đến đỉnh s
  repeat //Lặp tới khi hàng đợi rỗng
    u := Pop; //Lấy từ hàng đợi ra một đỉnh u
    Time := Time + 1;
    f[u] := Time; //Ghi nhận thời điểm duyệt xong đỉnh u
    Output ← u; //Liệt kê u
    for ∀vEV: (u, v) ∈ E do //Xét những đỉnh v kề u
      if d[v] = 0 then //Nếu v chưa duyệt đến
        begin
          Push(v); //Đẩy v vào hàng đợi
```

```

        Time := Time + 1;
        d[v] := Time; //Ghi nhận thời điểm duyệt đến đỉnh v
    end;
until Queue =  $\emptyset$ ;
end;
begin //Chương trình chính
    Input  $\rightarrow$  Đồ thị G;
    for  $\forall v \in V$  do d[v] := 0; //Mọi đỉnh đều chưa được duyệt đến
    Time := 0;
    for  $\forall v \in V$  do
        if avail[v] then BFSVisit(v);
    end.
end.

```

Thời gian thực hiện giải thuật của BFS tương tự như đối với DFS, bằng $\Theta(|V| + |E|)$ nếu đồ thị được biểu diễn bằng danh sách kề hoặc danh sách liên thuộc, bằng $\Theta(|V|^2)$ nếu đồ thị được biểu diễn bằng ma trận kề, và bằng $\Theta(|V||E|)$ nếu đồ thị được biểu diễn bằng danh sách cạnh.

Thứ tự duyệt đến và duyệt xong

Tương tự như thuật toán DFS, đối với thuật toán BFS người ta cũng quan tâm tới thứ tự duyệt đến và duyệt xong: Khi một đỉnh được đẩy vào hàng đợi, ta nói đỉnh đó được duyệt đến (được thăm) và khi một đỉnh được lấy ra khỏi hàng đợi, ta nói đỉnh đó được duyệt xong. Trong mô hình cài đặt trên, mỗi đỉnh u sẽ tương ứng với thời điểm duyệt đến d_u và thời điểm duyệt xong d_v .

Vì cách hoạt động của hàng đợi: đỉnh nào duyệt đến trước sẽ phải duyệt xong trước, chính vì vậy, việc liệt kê các đỉnh có thể thực hiện khi chúng được duyệt đến hay duyệt xong mà không ảnh hưởng tới thứ tự. Như cách cài đặt ở trên, mỗi đỉnh được đánh dấu mỗi khi đỉnh đó được duyệt đến và được liệt kê mỗi khi nó được duyệt xong.

Có thể sửa đổi một chút mô hình cài đặt bằng cách thay cơ chế đánh dấu duyệt đến/chưa duyệt đến bằng duyệt xong/chưa duyệt xong:

```

Input  $\rightarrow$  Đồ thị G;
for  $\forall v \in V$  do avail[v] := True; //Đánh dấu mọi đỉnh đều chưa duyệt xong
Queue :=  $\emptyset$ ;
for  $\forall v \in V$  do Push(v); //Khởi tạo hàng đợi chứa tất cả các đỉnh
repeat //Lặp tới khi hàng đợi rỗng
    u := Pop; //Lấy từ hàng đợi ra một đỉnh u
    if avail[u] then //Nếu u chưa duyệt xong
        begin

```

```

Output  $\leftarrow u$ ; //Liệt kê  $u$ 
avail[ $u$ ] := False; //Đánh dấu  $u$  đã duyệt xong
for  $\forall v \in V$ : avail[ $v$ ] and  $((u, v) \in E)$  do //Xét những đỉnh  $v$  kề  $u$  chưa
    duyệt xong
    begin
        trace[ $v$ ] :=  $u$ ; //Lưu vết đường đi
        Push( $v$ ); //Đẩy  $v$  vào hàng đợi
    end;
until Queue =  $\emptyset$ ;

```

Kết quả của hai cách cài đặt không khác nhau, sự khác biệt chỉ nằm ở lượng bộ nhớ cần sử dụng cho hàng đợi *Queue*: Ở cách cài đặt thứ nhất, do cơ chế đánh dấu duyệt đến/chưa duyệt đến, mỗi đỉnh sẽ được đưa vào *Queue* đúng một lần và lấy ra khỏi *Queue* đúng một lần nên chúng ta cần không quá n ô nhớ để chứa các phần tử của *Queue*. Ở cách cài đặt thứ hai, có thể có nhiều hơn n đỉnh đứng xếp hàng trong *Queue* vì một đỉnh v có thể được đẩy vào *Queue* tới $1 + \deg(v)$ lần (tính cả bước khởi tạo hàng đợi chứa tất cả các đỉnh), có nghĩa là khi tổ chức dữ liệu, chúng ta phải dự trù $\sum_{v \in V} (1 + \deg(v)) = 2m + n$ ô nhớ cho *Queue*. Con số này đối với đồ thị có hướng là $m + n$ ô nhớ.

Rõ ràng đối với BFS, cách cài đặt như ban đầu sẽ tiết kiệm bộ nhớ hơn. Nhưng có điểm đặc biệt là nếu thay cấu trúc hàng đợi bởi cấu trúc ngăn xếp trong cách cài đặt thứ hai, ta sẽ được thứ tự duyệt đỉnh DFS. Đây chính là phương pháp khử đệ quy của DFS để cài đặt thuật toán trên các ngôn ngữ không cho phép đệ quy.

Bài tập

- 5.9. Viết chương trình cài đặt thuật toán DFS không đệ quy.
- 5.10. Xét đồ thị có hướng $G = (V, E)$, dùng thuật toán DFS duyệt đồ thị G . Cho một phản ví dụ để chứng minh giả thuyết sau là sai: Nếu từ đỉnh u có đường đi tới đỉnh v và u được duyệt đến trước v , thì v nằm trong nhánh DFS gốc u .
- 5.11. Cho đồ thị vô hướng $G = (V, E)$, tìm thuật toán $O(|V|)$ để phát hiện một chu trình đơn trong G .
- 5.12. Cho đồ thị có hướng $G = (V, E)$ có n đỉnh, và mỗi đỉnh i được gán một nhãn là số nguyên a_i , tập cung E của đồ thị được định nghĩa là $(u, v) \in E \Leftrightarrow a_u \geq a_v$. Giả sử rằng thuật toán DFS được sử dụng để duyệt đồ thị, hãy khảo sát tính chất của dãy các nhãn nếu ta xếp các đỉnh theo thứ tự từ đỉnh duyệt xong đầu tiên đến đỉnh duyệt xong sau cùng.
- 5.13. Mê cung hình chữ nhật kích thước $m \times n$ gồm các ô vuông đơn vị ($m, n \leq 1000$). Trên mỗi ô ghi một trong ba ký tự:

- O: Nếu ô đó an toàn
- X: Nếu ô đó có cạm bẫy
- E: Nếu là ô có một nhà thám hiểm đang đứng.

Duy nhất chỉ có 1 ô ghi chữ E. Nhà thám hiểm có thể từ một ô đi sang một trong số các ô chung cạnh với ô đang đứng. Một cách đi thoát khỏi mê cung là một hành trình đi qua các ô an toàn ra một ô biên. Hãy chỉ giúp cho nhà thám hiểm một hành trình thoát ra khỏi mê cung đi qua ít ô nhất.

4. Tính liên thông của đồ thị

4.1. Định nghĩa

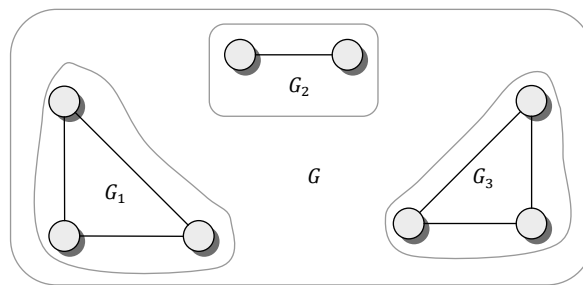
a) Tính liên thông trên đồ thị vô hướng

Đồ thị vô hướng $G = (V, E)$ được gọi là *liên thông (connected)* nếu giữa mọi cặp đỉnh của G luôn tồn tại đường đi. Đồ thị chỉ gồm một đỉnh duy nhất cũng được coi là đồ thị liên thông.

Cho đồ thị vô hướng $G = (V, E)$ và U là một tập con khác rỗng của tập đỉnh V . Ta nói U là một *thành phần liên thông (connected component)* của G nếu:

- Đồ thị G hạn chế trên tập U : $G_U = (U, E_U)$ là đồ thị liên thông.
- Không tồn tại một tập W chứa U mà đồ thị G hạn chế trên W là liên thông (tính tối đại của U).

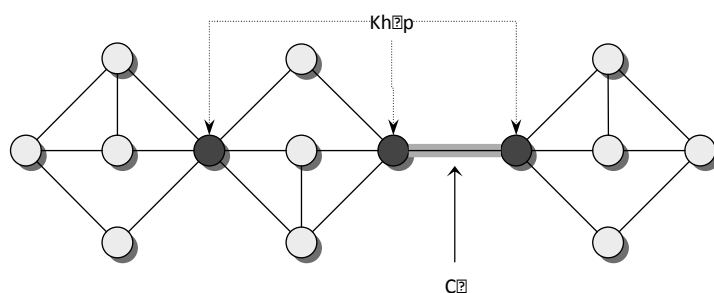
(Ta cũng đồng nhất khái niệm thành phần liên thông U với thành phần liên thông $G_U = (U, E_U)$).



Hình 5-14: Đồ thị và các thành phần liên thông

Một đồ thị liên thông chỉ có một thành phần liên thông là chính nó. Một đồ thị không liên thông sẽ có nhiều hơn 1 thành phần liên thông. Hình 5-14 là ví dụ về đồ thị G và các thành phần liên thông G_1, G_2, G_3 của nó.

Đôi khi, việc xoá đi một đỉnh và tất cả các cạnh liên thuộc với nó sẽ tạo ra một đồ thị con mới có nhiều thành phần liên thông hơn đồ thị ban đầu, các đỉnh như thế gọi là *đỉnh cắt* (cut vertices) hay *nút khớp* (articulation nodes). Hoàn toàn tương tự, những cạnh mà khi ta bỏ nó đi sẽ tạo ra một đồ thị có nhiều thành phần liên thông hơn so với đồ thị ban đầu được gọi là *cạnh cắt* (cut edges) hay *cầu* (bridges).

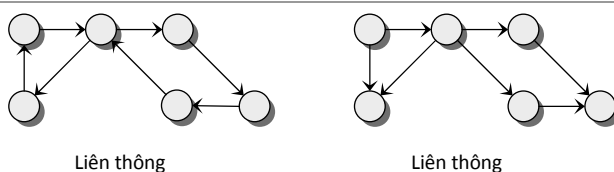


Hình 5-15: Khớp và cầu

b) Tính liên thông trên đồ thị có hướng

Cho đồ thị có hướng $G = (V, E)$, có hai khái niệm về tính liên thông của đồ thị có hướng tùy theo chúng ta có quan tâm tới hướng của các cung không.

G gọi là *liên thông mạnh* (strongly connected) nếu luôn tồn tại đường đi (theo các cung định hướng) giữa hai đỉnh bất kì của đồ thị, G gọi là *liên thông yếu* (weakly connected) nếu phiên bản vô hướng của nó là đồ thị liên thông.



Hình 5-16: Liên thông mạnh và liên thông yếu

4.2. Bài toán xác định các thành phần liên thông

Một bài toán quan trọng trong lý thuyết đồ thị là bài toán kiểm tra tính liên thông của đồ thị vô hướng hay tổng quát hơn: Bài toán liệt kê các thành phần liên thông của đồ thị vô hướng.

Để liệt kê các thành phần liên thông của đồ thị vô hướng $G = (V, E)$, phương pháp cơ bản nhất là bắt đầu từ một đỉnh bất kì, ta liệt kê những đỉnh đến được từ đỉnh đó vào một thành phần liên thông, sau đó loại tất cả các đỉnh đã liệt kê ra

khởi đồ thị và lặp lại, thuật toán sẽ kết thúc khi tập đỉnh của đồ thị trở thành \emptyset . Việc loại bỏ đỉnh của đồ thị có thể thực hiện bằng cơ chế đánh dấu những đỉnh bị loại:

```

procedure Scan( $u \in V$ )
begin
    «Dùng BFS hoặc DFS liệt kê và đánh dấu những đỉnh có thể
    đến được từ  $u$ »;
end;
begin
    for  $\forall u \in V$  do
        «Khởi tạo  $v$  chưa bị đánh dấu»;
        Count := 0;
        for  $\forall u \in V$  do
            if « $u$  chưa bị đánh dấu» then
                begin
                    Count := Count + 1;
                    Output  $\leftarrow$  «Thông báo thành phần liên thông thứ Count
                    gồm các đỉnh :»;
                    Scan( $u$ );
                end;
            end;
        end;
    end.

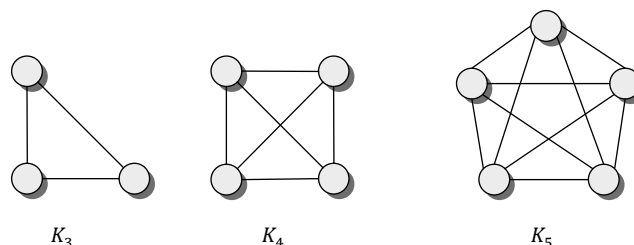
```

Thời gian thực hiện giải thuật đúng bằng thời gian thực hiện giải thuật duyệt đồ thị bằng DFS hoặc BFS.

4.3. Bao đóng của đồ thị vô hướng

a) Định nghĩa

Đồ thị đầy đủ với n đỉnh, kí hiệu K_n , là một đơn đồ thị vô hướng mà giữa hai đỉnh bất kì của nó đều có cạnh nối. Đồ thị đầy đủ K_n có đúng $\binom{n}{2} = \frac{n(n-1)}{2}$ cạnh, bậc của mọi đỉnh đều là $n - 1$



Hình 5-17: Đồ thị đầy đủ

b) Bao đóng đồ thị

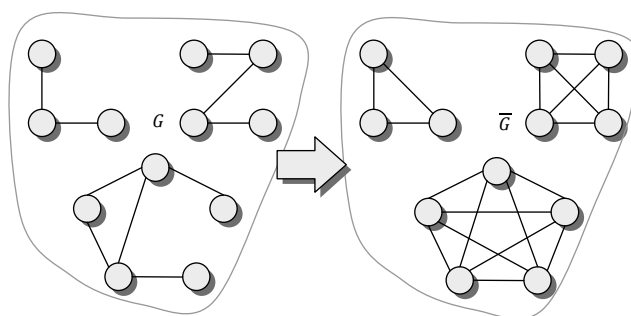
Với đồ thị $G = (V, E)$, người ta xây dựng đồ thị $\overline{G} = (V, \overline{E})$ cũng gồm những đỉnh của G còn các cạnh xây dựng như sau:

Giữa hai đỉnh u, v của \overline{G} có cạnh nối \Leftrightarrow Giữa hai đỉnh u, v của G có đường đi

Đồ thị \overline{G} xây dựng như vậy được gọi là bao đóng của đồ thị G .

Từ định nghĩa của đồ thị đầy đủ, và đồ thị liên thông, ta suy ra:

- Một đơn đồ thị vô hướng là liên thông nếu và chỉ nếu bao đóng của nó là đồ thị đầy đủ
- Một đơn đồ thị vô hướng có k thành phần liên thông nếu và chỉ nếu bao đóng của nó có k thành phần đầy đủ.



Hình 5-18: Đơn đồ thị vô hướng và bao đóng của nó

Bởi việc kiểm tra một đơn đồ thị có phải đồ thị đầy đủ hay không có thể thực hiện khá dễ dàng (đếm số cạnh chẳng hạn) nên người ta nảy ra ý tưởng có thể kiểm tra tính liên thông của đồ thị thông qua việc kiểm tra tính đầy đủ của bao đóng. Vấn đề đặt ra là phải có thuật toán xây dựng bao đóng của một đồ thị cho trước và một trong những thuật toán đó là:

c) Thuật toán Warshall

Thuật toán Warshall – gọi theo tên của Stephen Warshall, người đã mô tả thuật toán này vào năm 1960, đôi khi còn được gọi là thuật toán Roy-Warshall vì Bernard Roy cũng đã mô tả thuật toán này vào năm 1959. Thuật toán đó có thể mô tả rất gọn:

Giả sử đơn đồ thị vô hướng $G = (V, E)$ có n đỉnh đánh số từ 1 tới n , thuật toán Warshall xét tất cả các đỉnh $k \in V$, với mỗi đỉnh k được xét, thuật toán lại xét tiếp tất cả các cặp đỉnh (i, j) : nếu đồ thị có cạnh (i, k) và cạnh (k, j) thì ta tự nối thêm cạnh (i, j) nếu nó chưa có. Tư tưởng này dựa trên một quan sát đơn giản như sau:

Nếu từ i có đường đi tới k và từ k lại có đường đi tới j thì chắc chắn từ i sẽ có đường đi tới j .

Thuật toán Warshall yêu cầu đồ thị phải được biểu diễn bằng ma trận kề $A = \{a_{ij}\}$, trong đó $a_{ij} = \text{True} \Leftrightarrow (i, j) \in E$. Mô hình cài đặt thuật toán khá đơn giản:

```
for k := 1 to n do
  for i := 1 to n do
    for j := 1 to n do
      a[i, j] := a[i, j] or a[i, k] and a[k, j];
```

Việc chứng minh tính đúng đắn của thuật toán đòi hỏi phải lật lại các lý thuyết về bao đóng bắc cầu và quan hệ liên thông, ta sẽ không trình bày ở đây. Tuy thuật toán Warshall rất dễ cài đặt nhưng đòi hỏi thời gian thực hiện giải thuật khá lớn: $\Theta(n^3)$. Chính vì vậy thuật toán Warshall chỉ nên sử dụng khi thực sự cần tới bao đóng của đồ thị, còn nếu chỉ cần liệt kê các thành phần liên thông thì các thuật toán tìm kiếm trên đồ thị tỏ ra hiệu quả hơn nhiều.

Dưới đây, ta sẽ thử cài đặt thuật toán Warshall tìm bao đóng của đơn đồ thị vô hướng sau đó đếm số thành phần liên thông của đồ thị:

Việc cài đặt thuật toán sẽ qua những bước sau:

- Dùng ma trận kề A biểu diễn đồ thị, quy ước rằng $a_{ij} = \text{True}, \forall i$
- Dùng thuật toán Warshall tìm bao đóng, khi đó A là ma trận kề của đồ thị bao đóng
- Dựa vào ma trận kề A , đỉnh 1 và những đỉnh kề với nó sẽ thuộc thành phần liên thông thứ nhất; với đỉnh u nào đó không kề với đỉnh 1, thì u cùng với những đỉnh kề nó sẽ thuộc thành phần liên thông thứ hai; với đỉnh v nào đó không kề với cả đỉnh 1 và đỉnh u , thì v cùng với những đỉnh kề nó sẽ thuộc thành phần liên thông thứ ba v.v...

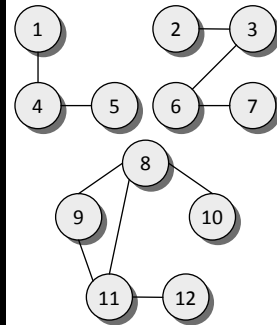
Input

- Dòng 1: Chứa số đỉnh $n \leq 200$ và số cạnh m của đồ thị
- m dòng tiếp theo, mỗi dòng chứa một cặp số u và v tương ứng với một cạnh (u, v)

Output

Liệt kê các thành phần liên thông của đồ thị

Sample Input	Sample Output
12 10 1 4 2 3 3 6 4 5 6 7 8 9 8 10 9 11 11 8 11 12	Connected Component 1: 1, 4, 5, Connected Component 2: 2, 3, 6, 7, Connected Component 3: 8, 9, 10, 11, 12,



WARSHALL.PAS ✓ Thuật toán Warshall liệt kê các thành phần liên thông

```

{$MODE OBJFPC}
program WarshallAlgorithm;
const
  maxN = 200;
var
  a: array[1..maxN, 1..maxN] of Boolean; //Ma trận kề của đồ thị
  n: Integer;
procedure Enter; //Nhập đồ thị
var
  i, j, k, m: Integer;
begin
  ReadLn(n, m);
  for i := 1 to n do
    begin
      FillChar(a[i][1], n * SizeOf(a[i][1]), False);
      a[i, i] := True;
    end;
  for k := 1 to m do
    begin
      ReadLn(i, j);
      a[i, j] := True;
      a[j, i] := True; //Đồ thị vô hướng: (i, j) = (j, i)
    end;
  end;
procedure ComputeTransitiveClosure; //Thuật toán Warshall
var

```

```

    k, i, j: Integer;
begin
    for k := 1 to n do
        for i := 1 to n do
            for j := 1 to n do
                a[i, j] := a[i, j] or a[i, k] and a[k, j];
            end;
        end;
    end;
    procedure PrintResult;
    var
        Count: Integer;
        avail: array[1..maxN] of Boolean; //avail[v] = True ↔ v chưa được liệt kê
        vào thành phần liên thông nào
        u, v: Integer;
    begin
        FillChar(avail, n * SizeOf(Boolean), True); //Mọi đỉnh đều chưa
        được liệt kê vào thành phần liên thông nào
        Count := 0;
        for u := 1 to n do
            if avail[u] then //Với một đỉnh u chưa được liệt kê vào thành phần liên thông
            nào
                begin //Liệt kê thành phần liên thông chứa u
                    Inc(Count);
                    Write('Connected Component ', Count, ': ');
                    for v := 1 to n do
                        if a[u, v] then //Xét những đỉnh v kề u (trên bao đóng)
                            begin
                                Write(v, ' '); //Liệt kê đỉnh đó vào thành phần liên thông
                                chứa u
                                avail[v] := False; //Liệt kê đỉnh nào đánh dấu đỉnh đó
                            end;
                        WriteLn;
                    end;
                end;
        end;
    begin
        Enter;
        ComputeTransitiveClosure;
        PrintResult;
    end.

```

4.4. Bài toán xác định các thành phần liên thông mạnh

Đối với đồ thị có hướng, người ta quan tâm đến bài toán kiểm tra tính liên thông mạnh, hay tổng quát hơn: Bài toán liệt kê các thành phần liên thông mạnh của đồ

thị có hướng. Các thuật toán tìm kiếm thành phần liên thông mạnh hiệu quả hiện nay đều dựa trên thuật toán tìm kiếm theo chiều sâu Depth-First Search.

Ta sẽ khảo sát và cài đặt hai thuật toán liệt kê thành phần liên thông mạnh với khuôn dạng Input/Output như sau:

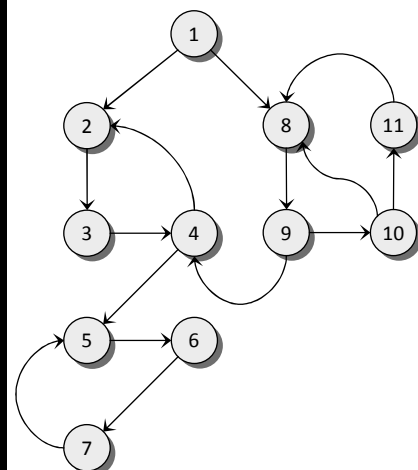
Input

- Dòng đầu: Chứa số đỉnh $n \leq 10^5$ và số cung $m \leq 10^6$ của đồ thị.
- m dòng tiếp theo, mỗi dòng chứa hai số nguyên u, v tương ứng với một cung (u, v) của đồ thị.

Output

Các thành phần liên thông mạnh.

Sample Input	Sample Output
11 15 1 2 1 8 2 3 3 4 4 2 4 5 5 6 6 7 7 5 8 9 9 4 9 10 10 8 10 11 11 8	Strongly Connected Component 1: 7, 6, 5, Strongly Connected Component 2: 4, 3, 2, Strongly Connected Component 3: 11, 10, 9, 8, Strongly Connected Component 4: 1,



a) Phân tích

Xét thuật toán tìm kiếm theo chiều sâu:

```

procedure DFSVisit(u ∈ V);
begin
    «Thêm u vào cây T»
    for ∀ v ∈ V: (u, v) ∈ E do
        if v ∉ T then
            begin

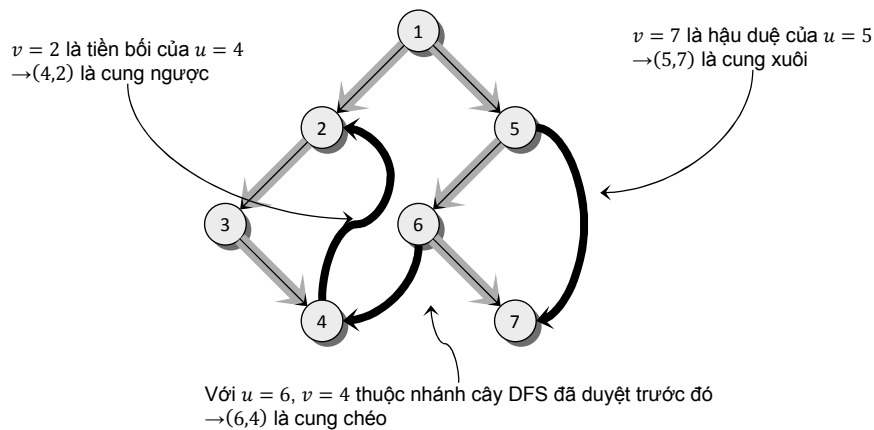
```

```

        «Thêm v và cung (u, v) vào cây T»;
        DFSVisit(v);
    end;
end;
begin
    Input → Đồ thị G;
    for  $\forall v \in V$  do avail[v] := True;
    for  $\forall v \in V$  do
        if avail[v] then
            begin
                «Tạo ra một cây rỗng, gọi là T»
                DFSVisit(v);
            end;
        end;
    end.
end.

```

Đề ý thủ tục thăm đỉnh đệ quy $DFSVisit(u)$. Thủ tục này xét tất cả những đỉnh v nối từ u :



Hình 5-19: Ba dạng cung ngoài cây DFS

- Nếu v chưa được thăm thì đi theo cung đó thăm v , tức là cho đỉnh v trở thành con của đỉnh u trong cây tìm kiếm DFS, cung (u, v) khi đó được gọi là cung DFS (Tree edge).
- Nếu v đã thăm thì có ba khả năng xảy ra đối với vị trí của u và v trong cây tìm kiếm DFS:
 - v là tiền bối (*ancestor*) của u , tức là v được thăm trước u và thủ tục $DFSVisit(u)$ do đây chuyển đệ quy từ thủ tục $DFSVisit(v)$ gọi tới. Cung (u, v) khi đó được gọi là *cung ngược* (*back edge*)

- v là hậu duệ (*descendant*) của u , tức là u được thăm trước v , nhưng thủ tục $DFSVisit(u)$ sau khi tiến đệ quy theo một hướng khác đã gọi $DFSVisit(v)$ rồi. Nên khi đây chuyển đệ quy lùi lại về thủ tục $DFSVisit(u)$ sẽ thấy v là đã thăm nên không thăm lại nữa. Cung (u, v) khi đó gọi là *cung xuôi* (*forward edge*).
- v thuộc một nhánh DFS đã duyệt trước đó, cung (u, v) khi đó gọi là *cung chéo* (*cross edge*)

Ta nhận thấy một đặc điểm của thuật toán tìm kiếm theo chiều sâu, thuật toán không chỉ duyệt qua các đỉnh, nó còn duyệt qua tất cả những cung nữa. Ngoài những cung nằm trên cây DFS, những cung còn lại có thể chia làm ba loại: cung ngược, cung xuôi, cung chéo (Hình 5-19).

b) Cây DFS và các thành phần liên thông mạnh

Định lí 5-9

Nếu x và y là hai đỉnh thuộc thành phần liên thông mạnh C thì với mọi đường đi từ x tới y cũng như từ y tới x . Tất cả đỉnh trung gian trên đường đi đều phải thuộc C .

Chứng minh

Vì x và y là hai đỉnh thuộc C nên có một đường đi từ x tới y và một đường đi khác từ y tới x . Nối tiếp hai đường đi này lại ta sẽ được một chu trình đi từ x tới y rồi quay lại x trong đó v là một đỉnh nằm trên chu trình. Điều này chỉ ra rằng nếu đi dọc theo chu trình ta có thể đi từ x tới v cũng như từ v tới x , nghĩa là v và x thuộc cùng một thành phần liên thông mạnh.

Định lí 5-10

Với một thành phần liên thông mạnh C bất kì, sẽ tồn tại duy nhất một đỉnh $r \in C$ sao cho mọi đỉnh của C đều thuộc nhánh cây DFS gốc r .

Chứng minh

Trong số các đỉnh của C , chọn r là đỉnh được thăm đầu tiên theo thuật toán DFS. Ta sẽ chứng minh C nằm trong nhánh DFS gốc r . Thật vậy: với một đỉnh v bất kì của C , do C liên thông mạnh nên phải tồn tại một đường đi từ r tới v :

$$\langle r = x_0, x_1, \dots, x_k = v \rangle$$

Từ Định lí 5-9, tất cả các đỉnh x_1, x_2, \dots, x_k đều thuộc C , lại do cách chọn r nên chúng sẽ phải thăm sau đỉnh r . Lại từ Định lí 5-8 (định lí đường đi trắng), tất cả các đỉnh $x_1, x_2, \dots, x_k = v$ phải là hậu duệ của r tức là chúng đều thuộc nhánh DFS gốc r .

Đỉnh r trong chứng minh định lí – đỉnh thăm trước tất cả các đỉnh khác trong C – gọi là chốt của thành phần liên thông mạnh C . Mỗi thành phần liên thông mạnh có duy nhất một chốt. Xét về vị trí trong cây DFS, chốt của một thành phần liên

thông mạnh là đỉnh nằm cao nhất so với các đỉnh khác thuộc thành phần đó, hay nói cách khác: là tiền bối của tất cả các đỉnh thuộc thành phần đó.

Định lí 5-11

Với một chốt r không là tiền bối của bất kì chốt nào khác thì các đỉnh thuộc nhánh DFS gốc r chính là thành phần liên thông mạnh chứa r .

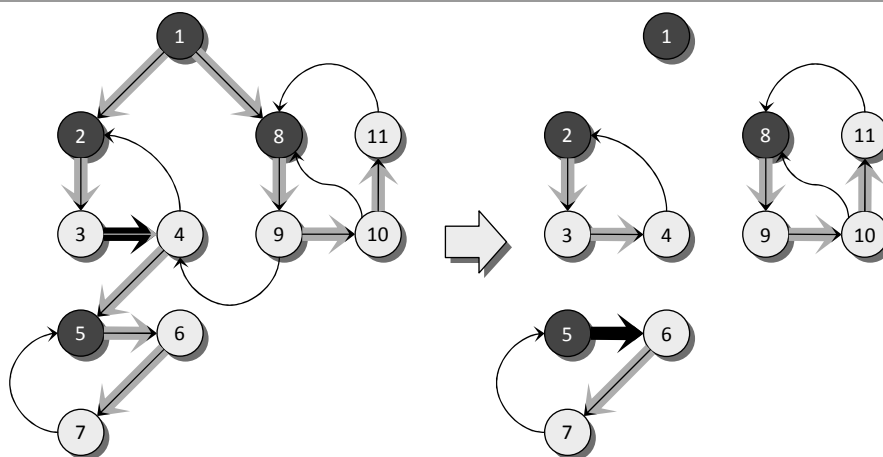
Chứng minh

Với mọi đỉnh v nằm trong nhánh DFS gốc r , gọi s là chốt của thành phần liên thông mạnh chứa v . Ta sẽ chứng minh $r = s$. Thật vậy, theo Định lí 5-10, v phải nằm trong nhánh DFS gốc s . Vậy v nằm trong cả nhánh DFS gốc r và nhánh DFS gốc s , nghĩa là r và s có quan hệ tiền bối–hậu duệ. Theo giả thiết r không là tiền bối của bất kì chốt nào khác nên r phải là hậu duệ của s . Ta có đường đi $s \rightsquigarrow r \rightsquigarrow v$, mà s và v thuộc cùng một thành phần liên thông mạnh nên theo Định lí 5-9, r cũng phải thuộc thành phần liên thông mạnh đó. Mỗi thành phần liên thông mạnh có duy nhất một chốt mà r và s đều là chốt nên $r = s$.

Theo Định lí 5-10, ta đã có thành phần liên thông mạnh chứa r nằm trong nhánh DFS gốc r , theo chứng minh trên ta lại có: Mọi đỉnh trong nhánh DFS gốc r nằm trong thành phần liên thông mạnh chứa r . Kết hợp lại được: Nhánh DFS gốc r chính là thành phần liên thông mạnh chứa r .

c) Thuật toán Tarjan

Ý tưởng



Hình 5-20: Thuật toán Tarjan “bẻ” cây DFS

Thuật toán Tarjan [40] có thể phát biểu như sau: Chọn r là chốt không là tiền bối của một chốt nào khác, chọn lấy thành phần liên thông mạnh thứ nhất là nhánh DFS gốc r . Sau đó loại bỏ nhánh DFS gốc r ra khỏi cây DFS, lại tìm thấy một chốt s khác mà nhánh DFS gốc s không chứa chốt nào khác, lại chọn lấy thành

phần liên thông mạnh thứ hai là nhánh DFS gốc s ... Tương tự như vậy cho thành phần liên thông mạnh thứ ba, thứ tư, v.v... Có thể hình dung thuật toán Tarjan “bè” cây DFS tại vị trí các chốt để được các nhánh rời rạc, mỗi nhánh là một thành phần liên thông mạnh.

Mô hình cài đặt của thuật toán Tarjan:

```

procedure DFSVisit(u ∈ V);
begin
    «Đánh dấu u đã thăm»
    for ∀v ∈ V: (u, v) ∈ E do
        if «v chưa thăm» then DFSVisit(v);
    if «u là chốt» then
        begin
            «Liệt kê thành phần liên thông mạnh tương ứng với chốt u»
            «Loại bỏ các đỉnh đã liệt kê khỏi đồ thị và cây DFS»
        end;
    end;
begin
    «Đánh dấu mọi đỉnh đều chưa thăm»
    for ∀v ∈ V do
        if «v chưa thăm» then DFSVisit(v);
    end.

```

Trình bày dài dòng như vậy, nhưng bây giờ chúng ta mới thảo luận tới vấn đề quan trọng nhất: Làm thế nào kiểm tra một đỉnh r nào đó có phải là chốt hay không?

Định lý 5-12

Trong mô hình cài đặt của thuật toán Tarjan, việc kiểm tra đỉnh r có phải là chốt không được thực hiện khi đỉnh r được duyệt xong, khi đó r là chốt nếu và chỉ nếu trong nhánh DFS gốc r không có cung tới đỉnh thăm trước r .

Chứng minh

Ta nhắc lại các tính chất của 4 loại cung:

- Cung DFS và cung xuôi nối từ đỉnh thăm trước đến đỉnh thăm sau, hơn nữa chúng đều là cung nối từ tiền bối tới hậu duệ
- Cung ngược và cung chéo nối từ đỉnh thăm sau tới đỉnh thăm trước, cung ngược nối từ hậu duệ tới tiền bối còn cung chéo nối hai đỉnh không có quan hệ tiền bối–hậu duệ.

Nếu trong nhánh DFS gốc r không có cung tới đỉnh thăm trước r thì tức là không tồn tại cung ngược và cung chéo đi ra khỏi nhánh DFS gốc r . Điều đó chỉ ra rằng từ r , đi theo các cung của đồ thị sẽ chỉ đến được những đỉnh nằm trong nội bộ nhánh DFS gốc r mà thôi. Thành phần liên thông mạnh chứa r phải nằm trong tập các đỉnh có thể đến từ r , tập này lại chính là nhánh DFS gốc r , vậy nên r là chốt.

Ngược lại, nếu từ đỉnh u của nhánh DFS gốc r có cung (u, v) tới đỉnh thăm trước r thì cung đó phải là cung ngược hoặc cung chéo.

Nếu cung (u, v) là cung ngược thì v là tiền bối của u , mà r cũng là tiền bối của u nhưng thăm sau v nên r là hậu duệ của v . Ta có một chu trình $v \rightsquigarrow r \rightsquigarrow u \rightarrow v$ nên cả v và r thuộc cùng một thành phần liên thông mạnh. Xét về vị trí trên cây DFS, v là tiền bối của r nên r không thể là chốt

Nếu cung (u, v) là cung chéo, ta gọi s là chốt của thành phần liên thông mạnh chứa v . Tại thời điểm thủ tục $DFSVisit(u)$ xét tới cung (u, v) , đỉnh r đã được duyệt đến nhưng chưa duyệt xong (do r là tiền bối của u), đỉnh s cũng đã duyệt đến (s được thăm trước v do s là chốt của thành phần liên thông mạnh chứa v , v được thăm trước r theo giả thiết, r được thăm trước u vì r là chốt của thành phần liên thông mạnh chứa u) nhưng chưa duyệt xong (vì nếu s được duyệt xong thì thuật toán đã loại bỏ tất cả các đỉnh thuộc thành phần liên thông mạnh chốt s trong đó có đỉnh v ra khỏi đồ thị nên cung (u, v) sẽ không được tính đến nữa), điều này chỉ ra rằng khi $DFSVisit(u)$ được gọi, hai thủ tục $DFSVisit(r)$ và $DFSVisit(s)$ đều đã được gọi nhưng chưa thoát, tức là chúng nằm trên một dây chuyền đệ quy, hay r và s có quan hệ tiền bối-hậu duệ. Vì s được thăm trước r nên s sẽ là tiền bối của r , ta có chu trình $s \rightsquigarrow r \rightsquigarrow u \rightarrow v \rightsquigarrow s$ nên r và s thuộc cùng một thành phần liên thông mạnh, thành phần này đã có chốt s rồi nên r không thể là chốt nữa.

Từ Định lý 5-12, việc sẽ kiểm tra đỉnh r có là chốt hay không có thể thay bằng việc *kiểm tra xem có tồn tại cung nối từ một đỉnh thuộc nhánh DFS gốc r tới một đỉnh thăm trước r hay không?*

Dưới đây là một cách cài đặt hết sức thông minh, nội dung của nó là đánh số thứ tự các đỉnh theo thứ tự duyệt đến. Định nghĩa $Number[u]$ là số thứ tự của đỉnh u theo cách đánh số đó. Ta tính thêm $Low[u]$ là giá trị $Number[.]$ nhỏ nhất trong các đỉnh có thể đến được từ một đỉnh v nào đó của nhánh DFS gốc u bằng một cung. Cụ thể cách tính $Low[u]$ như sau:

Trong thủ tục $DFSVisit(u)$, trước hết ta đánh số thứ tự thăm cho đỉnh u : $Number[u]$ và khởi tạo $Low[u] := +\infty$. Sau đó xét các đỉnh v nối từ u , có hai khả năng:

Nếu v đã thăm thì ta cực tiểu hoá $Low[u]$ theo công thức:

$$Low[u]_{\text{mới}} := \min(Low[u]_{\text{cũ}}, Number[v])$$

Nếu v chưa thăm thì ta gọi đệ quy $DFSVisit(v)$, sau đó cực tiểu hoá $Low[u]$ theo công thức:

$$Low[u]_{\text{mới}} := \min(Low[u]_{\text{cũ}}, Low[v])$$

Khi duyệt xong một đỉnh u (chuẩn bị thoát khỏi thủ tục $DFSVisit(u)$), ta so sánh $Low[u]$ và $Number[u]$, nếu như $Low[u] \geq Number[u]$ thì u là chốt, bởi không có cung nối từ một đỉnh thuộc nhánh DFS gốc u tới một đỉnh thăm trước u . Khi đó chỉ việc liệt kê các đỉnh thuộc thành phần liên thông mạnh chứa u chính là nhánh DFS gốc u .

Để công việc dễ dàng hơn nữa, ta định nghĩa một danh sách *Stack* được tổ chức dưới dạng ngăn xếp và dùng ngăn xếp này để lấy ra các đỉnh thuộc một nhánh nào đó. Khi duyệt đến một đỉnh u , ta đẩy ngay đỉnh u đó vào ngăn xếp, thì khi duyệt xong đỉnh u , mọi đỉnh thuộc nhánh DFS gốc u sẽ được đẩy vào ngăn xếp *Stack* ngay sau u . Nếu u là chót, ta chỉ việc lấy các đỉnh ra khỏi ngăn xếp *Stack* cho tới khi lấy tới đỉnh u là sẽ được nhánh DFS gốc u cũng chính là thành phần liên thông mạnh chứa u .

Mô hình

Dưới đây là mô hình cài đặt đầy đủ của thuật toán Tarjan

```

procedure DFSVisit( $u \in V$ ) ;
begin
    Count := Count + 1;
    Number[ $u$ ] := Count; //Đánh số  $u$  theo thứ tự duyệt đến
    Low[ $u$ ] :=  $+\infty$ ;
    Push( $u$ ) ; //Đẩy  $u$  vào ngăn xếp
    for  $\forall v \in V: (u, v) \in E$  do
        if Number[ $v$ ] > 0 then //v đã thăm
            Low[ $u$ ] := min(Low[ $u$ ], Number[ $v$ ])
        else //v chưa thăm
            begin
                DFSVisit( $v$ ) ; //Đi thăm v
                Low[ $u$ ] := min(Low[ $u$ ], Low[ $v$ ]) ;
            end;
        //Đến đây u được duyệt xong
    if Low[ $u$ ]  $\geq$  Number[ $u$ ] then //Nếu  $u$  là chót
        begin
            «Thông báo thành phần liên thông mạnh với chót  $u$  gồm có
            các đỉnh:»;
            repeat
                 $v :=$  Pop; //Lấy từ ngăn xếp ra một đỉnh v
                Output  $\leftarrow v$ ;
                «Xoá đỉnh v khỏi đồ thị:  $V := V - \{v\}$ »;
            until  $v = u$ ;
        end;
    end;
begin
    Count := 0;
    Stack :=  $\emptyset$ ; //Khởi tạo một ngăn xếp rỗng
    for  $\forall v \in V$  do Number[ $v$ ] := 0; //Number[ $v$ ] = 0  $\leftrightarrow$  v chưa thăm

```

```

for  $\forall v \in V$  do
    if Number[v] = 0 then DFSVisit(v);
end.

```

Bởi thuật toán Tarjan chỉ là sửa đổi của thuật toán DFS, các phép vào/ra ngăn xếp được thực hiện không quá n lần. Vậy nên thời gian thực hiện giải thuật vẫn là $\Theta(|V| + |E|)$ trong trường hợp đồ thị được biểu diễn bằng danh sách kề hoặc danh sách liên thuộc, là $\Theta(|V|^2)$ nếu dùng ma trận kề và là $\Theta(|V||E|)$ nếu dùng danh sách cạnh.

Cài đặt

Chương trình cài đặt dưới đây biểu diễn đồ thị bởi danh sách liên thuộc kiểu forward star: Mỗi đỉnh u sẽ được cho tương ứng với một danh sách các cung đi ra khỏi u , như vậy mỗi cung sẽ xuất hiện trong đúng một danh sách liên thuộc. Nếu các cung được lưu trữ trong mảng $e[1 \dots m]$, danh sách liên thuộc được xây dựng bằng hai mảng.

- $head[u]$ là chỉ số cung đầu tiên trong danh sách liên thuộc của đỉnh u . Nếu danh sách liên thuộc đỉnh u là \emptyset , $head[u]$ được gán bằng 0.
- $link[i]$ là chỉ số cung kế tiếp cung $e[i]$ trong danh sách liên thuộc chứa cung $e[i]$. Trường hợp $e[i]$ là cung cuối cùng của một danh sách liên thuộc, $link[i]$ được gán bằng 0

TARJAN.PAS ✓ Thuật toán Tarjan

```

{$MODE OBJFPC}
{$M 40000000}
program StronglyConnectedComponents;
const
    maxN = 1000000;
    maxM = 10000000;
type
    TStack = record
        Items: array[1..maxN] of Integer;
        Top: Integer;
    end;
    TEdge = record //Cấu trúc cung
        x, y: Integer; //Hai đỉnh đầu mút
    end;
var
    e: array[1..maxM] of TEdge; //Danh sách cạnh
    link: array[1..maxM] of Integer; //link[i]: chỉ số cung tiếp theo e[i] trong
    danh sách liên thuộc

```

```

    head: array[1..maxN] of Integer; //head[u]: chỉ số cung đầu tiên trong
danh sách liên thuộc các cung đi ra khỏi u
    avail: array[1..maxN] of Boolean;
    Number, Low: array[1..maxN] of Integer;
    Stack: TStack;
    n, Count, SCC: Integer;
procedure Enter; //Nhập dữ liệu
var
    i, u, v, m: Integer;
begin
    ReadLn(n, m);
    for i := 1 to m do //Đọc danh sách cạnh
        with e[i] do ReadLn(x, y);
    FillChar(head[1], n * SizeOf(head[1]), 0); //Khởi tạo các danh sách
liên thuộc rỗng
    for i := m downto 1 do //Xây dựng các danh sách liên thuộc
        with e[i] do
            begin
                link[i] := head[x]; //Móc nối e[i] = (x, y) vào danh sách liên thuộc
những cung đi ra khỏi x
                head[x] := i;
            end;
    end;
procedure Init; //Khởi tạo
begin
    FillChar(Number, n * SizeOf(Number[1]), 0); //Mọi đỉnh đều chưa
thăm
    FillChar(avail, n * SizeOf(avail[1]), True); //Chưa đỉnh nào bị
loại
    Stack.Top := 0; //Ngăn xếp rỗng
    Count := 0; //Biến đếm số thứ tự duyệt đến, dùng để đánh số
    SCC := 0; //Biến đánh số các thành phần liên thông
end;
procedure Push(v: Integer); //Đẩy một đỉnh v vào ngăn xếp
begin
    with Stack do
        begin
            Inc(Top); Items[Top] := v;
        end;
end;
function Pop: Integer; //Lấy một đỉnh v khỏi ngăn xếp, trả về trong kết quả
hàm
begin

```

```

with Stack do
begin
    Result := Items[Top]; Dec(Top);
end;
end;
//Hàm cực tiểu hoá: Target := Min(Target, Value)
procedure Minimize(var Target: Integer; Value: Integer);
begin
    if Value < Target then Target := Value;
end;
procedure DFSVisit(u: Integer); //Thuật toán tìm kiếm theo chiều sâu bắt đầu từ u
var
    i, v: Integer;
begin
    Inc(Count); Number[u] := Count; //Trước hết đánh số cho u
    Low[u] := maxN + 1; //khởi tạo Low[u]:=+∞ rồi sau cực tiểu hoá dần
    Push(u); //Đẩy u vào ngăn xếp
    i := head[u]; //Duyệt từ đầu danh sách liên thuộc các cung đi ra khỏi u
    while i <> 0 do
        begin
            v := e[i].y; //Xét những đỉnh v nối từ u
            if avail[v] then //Nếu v chưa bị loại
                if Number[v] <> 0 then //Nếu v đã thăm
                    Minimize(Low[u], Number[v]) //cực tiểu hoá Low[u] theo công thức này
                else //Nếu v chưa thăm
                    begin
                        DFSVisit(v); //Tiếp tục tìm kiếm theo chiều sâu bắt đầu từ v
                        Minimize(Low[u], Low[v]); //Rồi cực tiểu hoá Low[u] theo công thức này
                    end;
            i := link[i]; //Chuyển sang xét cung tiếp theo trong danh sách liên thuộc
        end;
    //Đến đây thì đỉnh u được duyệt xong, tức là các đỉnh thuộc nhánh DFS gốc u đều đã thăm
    if Low[u] >= Number[u] then //Nếu u là chót
        begin //Liệt kê thành phần liên thông mạnh có chót u
            Inc(SCC);
            WriteLn('Strongly Connected Component ', SCC, ': ');
            repeat
                v := Pop; //Lấy dần các đỉnh ra khỏi ngăn xếp
                Write(v, ', '); //Liệt kê các đỉnh đó
                avail[v] := False; //Rồi loại luôn khỏi đồ thị
            until v = 0;
        end;
    end;
end;

```

```

        until v = u; //Cho tới khi lấy tới đỉnh u
        WriteLn;
    end;
end;
procedure Tarjan; //Thuật toán Tarjan
var
    v: Integer;
begin
    for v := 1 to n do
        if avail[v] then DFSVisit(v);
    end;
begin
    Enter;
    Init;
    Tarjan;
end.

```

d) Thuật toán Kosaraju-Sharir

Mô hình

Có một thuật toán khác để liệt kê các thành phần liên thông mạnh là thuật toán Kosaraju-Sharir (1981). Thuật toán này thực hiện qua hai bước:

- Bước 1: Dùng thuật toán tìm kiếm theo chiều sâu với thủ tục *DFSVisit*, nhưng thêm vào một thao tác nhỏ: đánh số lại các đỉnh theo thứ tự duyệt xong.
- Bước 2: Đảo chiều các cung của đồ thị, xét lần lượt các đỉnh theo thứ tự từ đỉnh duyệt xong sau cùng tới đỉnh duyệt xong đầu tiên, với mỗi đỉnh đó, ta lại dùng thuật toán tìm kiếm trên đồ thị (BFS hay DFS) liệt kê những đỉnh nào đến được từ đỉnh đang xét, đó chính là một thành phần liên thông mạnh. Liệt kê xong thành phần nào, ta loại ngay các đỉnh của thành phần đó khỏi đồ thị.

Định lý 5-13

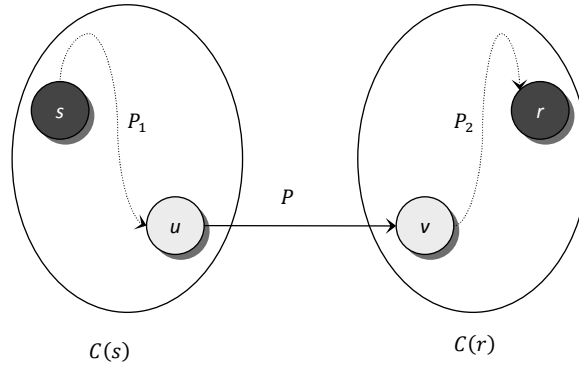
|| Với r là đỉnh duyệt xong sau cùng thì r là chót của một thành phần liên thông mạnh không có cung đi vào.

Chứng minh

Dễ thấy rằng đỉnh r duyệt xong sau cùng phải là gốc của một cây DFS nên r sẽ là chót của một thành phần liên thông mạnh, kí hiệu $C(r)$.

Gọi s là chót của một thành phần liên thông mạnh $C(s)$ khác. Ta chứng minh rằng không thể tồn tại cung đi từ $C(s)$ sang $C(r)$, giả sử phản chứng rằng có cung (u, v) trong đó $u \in C(s)$ và $v \in C(r)$. Khi đó tồn tại một đường đi $P_1: s \rightsquigarrow u$ trong nội bộ $C(s)$ và tồn tại

một đường đi $P_2: v \rightsquigarrow r$ nội bộ $C(r)$. Do tính chất của chốt, s được thăm trước mọi đỉnh khác trên đường P_1 và r được thăm trước mọi đỉnh khác trên đường P_2 . Nối đường đi $P_1: s \rightsquigarrow u$ với cung (u, v) và nối tiếp với đường đi $P_2: v \rightsquigarrow r$ ta được một đường đi $P: s \rightsquigarrow r$ (Hình 5-21)



Hình 5-21

Có hai khả năng xảy ra:

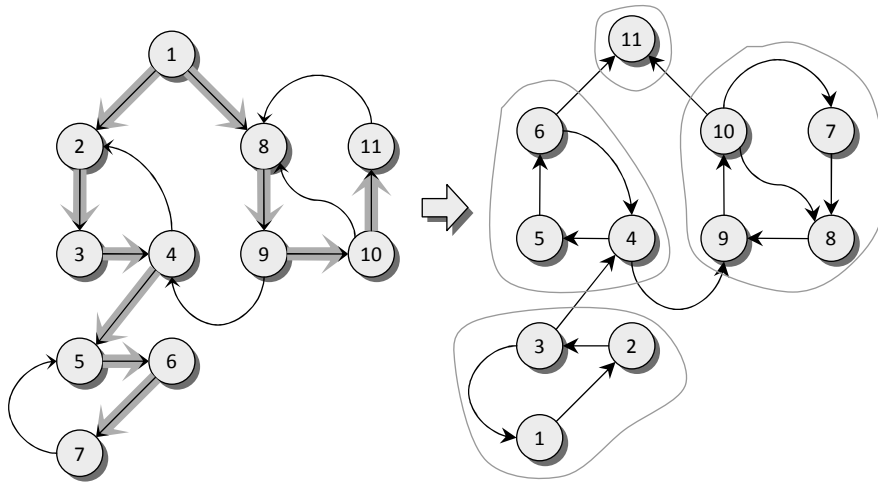
- Nếu s được thăm trước r thì vào thời điểm s được thăm, mọi đỉnh khác trên đường đi P chưa thăm. Theo Định lý 5-8 (định lý đường đi trắng), s sẽ là tiền bối của r và phải được duyệt xong sau r . Trái với giả thiết r là đỉnh duyệt xong sau cùng.
- Nếu s được thăm sau r , nghĩa là vào thời điểm r được duyệt đến thì s chưa duyệt đến, lại do r được duyệt xong sau cùng nên vào thời điểm r duyệt xong thì s đã duyệt xong. Theo Định lý 5-13, s sẽ là hậu duệ của r . Vậy từ s có đường đi tới r và ngược lại, nghĩa là r và s thuộc cùng một thành phần liên thông mạnh. Mâu thuẫn.

Định lý được chứng minh.

Định lý 5-13 chỉ ra tính đúng đắn của thuật toán Kosaraju-Sharir: Đỉnh r duyệt xong sau cùng chắc chắn là chốt của một thành phần liên thông mạnh và thành phần liên thông mạnh này gồm mọi đỉnh đến được r . Việc liệt kê các đỉnh thuộc thành phần liên thông mạnh chốt r được thực hiện trong thuật toán thông qua thao tác đảo chiều các cung của đồ thị rồi liệt kê các đỉnh đến được từ r .

Loại bỏ thành phần liên thông mạnh với chốt r khỏi đồ thị. Cây DFS gốc r lại phân rã thành nhiều cây con. Lập luận tương tự như trên với đỉnh duyệt xong sau cùng (Hình 5-22)

Ví dụ:



Hình 5-22. Đánh số lại, đảo chiều các cung và thực hiện thuật toán tìm kiếm trên đồ thị với cách chọn các đỉnh xuất phát ngược lại với thứ tự duyệt xong (thứ tự 11, 10... 3, 2, 1)

Cài đặt

Trong việc lập trình thuật toán Kosaraju–Sharir, việc đánh số lại các đỉnh được thực hiện bằng danh sách. Tại bước duyệt đồ thị lần 1, mỗi khi duyệt xong một đỉnh thì đỉnh đó được đưa vào cuối danh sách. Sau khi đảo chiều các cung của đồ thị, chúng ta chỉ cần duyệt từ cuối danh sách sẽ được các đỉnh đúng thứ tự ngược với thứ tự duyệt xong (cơ chế tương tự như ngăn xếp)

Để liệt kê các thành phần liên thông mạnh của đơn đồ thị có hướng bằng thuật toán Tarjan cũng như thuật toán Kosaraju–Sharir, cách biểu diễn đồ thị tốt nhất là sử dụng danh sách kề hoặc danh sách liên thuộc. Tuy nhiên với thuật toán Kosaraju–Sharir, việc cài đặt bằng danh sách liên thuộc là hợp lý hơn bởi nó cho phép chuyển từ cách biểu diễn forward star sang cách biểu diễn reverse star một cách dễ dàng bằng cách chỉnh lại mảng *link* và *head*. Cấu trúc forward star được sử dụng ở pha đánh số lại các đỉnh, còn cấu trúc reverse star được sử dụng khi liệt kê các thành phần liên thông mạnh (bởi cần thực hiện trên đồ thị đảo chiều)

KOSARAJUSHARIR.PAS ✓ Thuật toán Kosaraju–Sharir

```
{ $MODE OBJFPC }
{ $M 4000000 }
program StronglyConnectedComponents;
const
  maxN = 100000;
  maxM = 1000000;
```

```

type
  TEdge = record //Cấu trúc cung
    x, y: Integer; //Hai đỉnh đầu nút
  end;
var
  e: array[1..maxM] of TEdge; //Danh sách cạnh
  link: array[1..maxM] of Integer; //link[i]: Chỉ số cung kế tiếp e[i] trong
danh sách liên thuộc
  head: array[1..maxN] of Integer; //head[u]: Chỉ số cung đầu tiên trong
danh sách liên thuộc
  avail: array[1..maxN] of Boolean;
  List: array[1..maxN] of Integer;
  Top: Integer;
  n, m, v, SCC: Integer;
procedure Enter; //Nhập dữ liệu
var
  i, u, v: Integer;
begin
  ReadLn(n, m);
  for i := 1 to m do
    with e[i] do
      ReadLn(x, y);
  end;
procedure Numbering; //Liệt kê các đỉnh theo thứ tự duyệt xong vào danh sách List
var
  i, u: Integer;
  procedure DFSVisit(u: Integer); //Thuật toán DFS từ u
  var
    i, v: Integer;
  begin
    avail[u] := False;
    i := head[u];
    while i <> 0 do //Xét các cung e[i] đi ra khỏi u
      begin
        v := e[i].y;
        if avail[v] then DFSVisit(v);
        i := link[i];
      end;
    Inc(Top); List[Top] := u; //u duyệt xong, đưa u vào cuối danh sách List
  end;

```

```

begin
    //Xây dựng danh sách liên thuộc dạng forward star: Mỗi đỉnh u tương ứng với danh sách các cung đi ra khỏi u
    FillChar(head[1], n * SizeOf(head[1]), 0);
    for i := m downto 1 do
        with e[i] do
            begin
                link[i] := head[x];
                head[x] := i;
            end;
    FillChar(avail[1], n * SizeOf(avail[1]), True);
    Top := 0; //Khởi tạo danh sách List rỗng
    for u := 1 to n do
        if avail[u] then DFSVisit(u);
end;
procedure KosarajuSharir;
var
    i, u: Integer;
    procedure Enum(u: Integer); //Thuật toán DFS từ u trên đồ thị đảo chiều
    var
        i, v: Integer;
    begin
        avail[u] := False;
        Write(u, ', ');
        i := head[u];
        while i <> 0 do //Xét các cung e[i] đi vào u
            begin
                v := e[i].x;
                if avail[v] then Enum(v);
                i := link[i];
            end;
    end;
begin
    //Xây dựng danh sách liên thuộc dạng reverse star: mỗi đỉnh u tương ứng với danh sách các cung đi vào u
    FillChar(head[1], n * SizeOf(head[1]), 0);
    for i := m downto 1 do
        with e[i] do
            begin
                link[i] := head[y];
                head[y] := i;
            end;

```

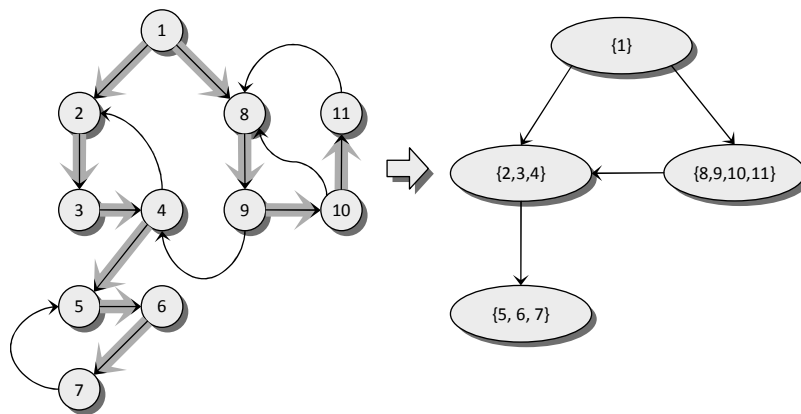
```

FillChar(avail[1], n * SizeOf(avail[1]), True);
SCC := 0;
for u := n downto 1 do
  if avail[List[u]] then //Liệt kê thành phần liên thông chốt List[u]
  begin
    Inc(SCC);
    WriteLn('Strongly Connected Component ', SCC, ': ');
    Enum(List[u]);
    WriteLn;
  end;
end;
begin
  Enter;
  Numbering;
  KosarajuSharir;
end.

```

Thời gian thực hiện giải thuật có thể tính bằng hai lượt DFS, vậy nên thời gian thực hiện giải thuật sẽ là $\Theta(|V| + |E|)$ trong trường hợp đồ thị được biểu diễn bằng danh sách kề hoặc danh sách liên thuộc, là $\Theta(|V|^2)$ nếu dùng ma trận kề và là $\Theta(|V||E|)$ nếu dùng danh sách cạnh.

4.5. Sắp xếp tô pô



Hình 5-23. Đồ thị có hướng và đồ thị các thành phần liên thông mạnh

Xét đồ thị có hướng $G = (V, E)$, ta xây dựng đồ thị có hướng $G^{SCC} = (V^{SCC}, E^{SCC})$ như sau: Mỗi đỉnh thuộc V^{SCC} tương ứng với một thành phần liên thông mạnh của G . Một cung $(r, s) \in E^{SCC}$ nếu và chỉ nếu tồn tại một cung $(u, v) \in E$ trên G trong đó $u \in r; v \in s$.

Đồ thị G^{SCC} gọi là đồ thị các thành phần liên thông mạnh

Đồ thị G^{SCC} là đồ thị có hướng không có chu trình (*directed acyclic graph-DAG*) vì nếu G^{SCC} có chu trình, ta có thể hợp tất cả các thành phần liên thông tương ứng với các đỉnh dọc trên chu trình để được một thành phần liên thông mạnh lớn trên đồ thị G , mâu thuẫn với tính tối đại của một thành phần liên thông mạnh.

Trong thuật toán Tarjan, khi một thành phần liên thông mạnh được liệt kê, thành phần đó sẽ tương ứng với một đỉnh không có cung đi ra trên G^{SCC} . Còn trong thuật toán Kosaraju–Sharir, khi một thành phần liên thông mạnh được liệt kê, thành phần đó sẽ tương ứng với một đỉnh không có cung đi vào trên G^{SCC} . Cả hai thuật toán đều loại bỏ thành phần liên thông mạnh mỗi khi liệt kê xong, tức là loại bỏ đỉnh tương ứng trên G^{SCC} .


Nếu ta đánh số các đỉnh của G^{SCC} từ 1 trở đi theo thứ tự các thành phần liên thông mạnh được liệt kê thì thuật toán Kosaraju–Sharir sẽ cho ta một cách đánh số gọi là *sắp xếp tô pô (topological sorting)* trên G^{SCC} : Các cung trên G^{SCC} khi đó sẽ chỉ nối từ đỉnh mang chỉ số nhỏ tới đỉnh mang chỉ số lớn. Nếu đánh số các đỉnh của G^{SCC} theo thuật toán Tarjan thì ngược lại, các cung trên G^{SCC} khi đó sẽ chỉ nối từ đỉnh mang chỉ số lớn tới đỉnh mang chỉ số nhỏ.

Bài tập

- 5.14. Chứng minh rằng đồ thị có hướng $G = (V, E)$ là không có chu trình nếu và chỉ nếu quá trình thực hiện thuật toán tìm kiếm theo chiều sâu trên G không có cung ngược.
- 5.15. Cho đồ thị có hướng không có chu trình $G = (V, E)$ và hai đỉnh s, t . Hãy tìm thuật toán đếm số đường đi từ s tới t (chỉ cần đếm số lượng, không cần liệt kê các đường).
- 5.16. Trên mặt phẳng với hệ toạ độ Decartes vuông góc cho n đường tròn, mỗi đường tròn xác định bởi bộ 3 số thực (x, y, r) ở đây (x, y) là toạ độ tâm và r là bán kính. Hai đường tròn gọi là thông nhau nếu chúng có điểm chung. Hãy chia các đường tròn thành một số tối thiểu các nhóm sao cho hai đường tròn bất kì trong một nhóm bất kì có thể đi được sang nhau sau một số hữu hạn các bước di chuyển giữa hai đường tròn thông nhau.
- 5.17. Cho một lưới ô vuông kích thước $m \times n$ gồm các số nhị phân $\in \{0, 1\}$ ($m, n \leq 1000$). Ta định nghĩa một hình là một miền liên thông các ô kề cạnh mang số 1. Hai hình được gọi là giống nhau nếu hai miền liên thông tương ứng có thể đặt chồng khít lên nhau qua một phép dời hình. Hãy phân

loại các hình trong lưới ra thành một số các nhóm thỏa mãn: Mỗi nhóm gồm các hình giống nhau và hai hình bất kì thuộc hai nhóm khác nhau thì không giống nhau:

1	1	1	0	1	1	0	0	1
1	0	0	0	1	0	0	1	1
1	1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0
0	0	1	1	0	1	0	0	0
1	0	0	0	0	1	0	0	1
1	0	1	0	0	1	1	0	1
1	1	1	1	1	0	0	1	1



1	1	1	0	2	2	0	0	2
1	0	0	0	2	0	0	2	2
1	1	0	0	0	0	0	0	0
1	0	0	3	0	0	0	0	0
1	0	0	3	0	0	0	0	0
0	0	3	3	0	3	0	0	0
1	0	0	0	0	3	0	0	3
1	0	1	0	0	3	3	0	3
1	1	1	1	1	0	0	3	3

- 5.18. Cho đồ thị có hướng $G = (V, E)$, hãy tìm thuật toán và viết chương trình để chọn ra một tập ít nhất các đỉnh $S \subseteq V$ để mọi đỉnh của V đều có thể đến được từ ít nhất một đỉnh của S bằng một đường đi trên G .
- 5.19. Một đồ thị có hướng $G = (V, E)$ gọi là *nửa liên thông (semi-connected)* nếu với mọi cặp đỉnh $u, v \in V$ thì hoặc u có đường đi đến v , hoặc v có đường đi đến u .
- a) Chứng minh rằng đồ thị có hướng $G = (V, E)$ là nửa liên thông nếu và chỉ nếu trên G tồn tại đường đi qua tất cả các đỉnh (không nhất thiết phải là đường đi đơn)
- b) Tìm thuật toán và viết chương trình kiểm tra tính nửa liên thông của đồ thị.

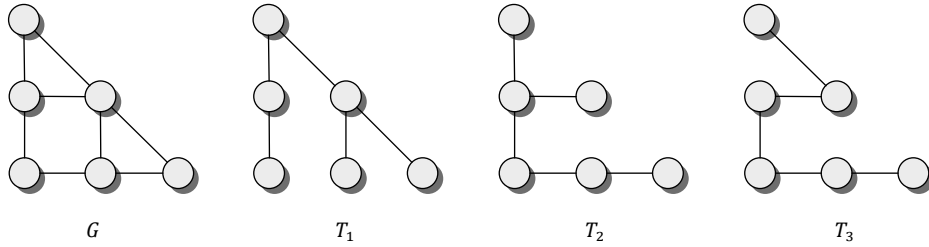
5. Vài ứng dụng của DFS và BFS

5.1. Xây dựng cây khung của đồ thị

Cây là đồ thị vô hướng, liên thông, không có chu trình đơn. Đồ thị vô hướng không có chu trình đơn gọi là rừng (hợp của nhiều cây). Như vậy mỗi thành phần liên thông của rừng là một cây.

Xét đồ thị $G = (V, E)$ và $T = (V, E_T)$ là một đồ thị con của đồ thị G ($E_T \subseteq E$), nếu T là một cây thì ta gọi T là *cây khung* hay *cây bao trùm (spanning tree)* của đồ thị G . Điều kiện cần và đủ để một đồ thị vô hướng có cây khung là đồ thị đó phải liên thông.

Dễ thấy rằng với một đồ thị vô hướng liên thông có thể có nhiều cây khung (Hình 5-24).



Hình 5-24: Đồ thị và một số ví dụ cây khung

Định lý 5-14 (Daisy Chain Theorem)

Giả sử $T = (V, E)$ là đồ thị vô hướng với n đỉnh. Khi đó các mệnh đề sau là tương đương:

1. T là cây
2. T không chứa chu trình đơn và có $n - 1$ cạnh
3. T liên thông và mỗi cạnh của nó đều là cầu
4. Giữa hai đỉnh bất kì của T đều tồn tại đúng một đường đi đơn
5. T không chứa chu trình đơn nhưng hề cứ thêm vào một cạnh ta thu được một chu trình đơn.
6. T liên thông và có $n - 1$ cạnh

Chứng minh:

1 \Rightarrow 2:

Từ T là cây, theo định nghĩa T không chứa chu trình đơn. Ta sẽ chứng minh cây T có n đỉnh thì phải có $n - 1$ cạnh bằng quy nạp theo số đỉnh n . Rõ ràng khi $n = 1$ thì cây có 1 đỉnh sẽ chứa 0 cạnh. Nếu $n = 1$, gọi $P = \langle v_1, v_2, \dots, v_k \rangle$ là đường đi dài nhất (qua nhiều cạnh nhất) trong T . Đỉnh v_1 không thể kề với đỉnh nào trong số các đỉnh v_3, v_4, \dots, v_k , bởi nếu có cạnh (v_1, v_p) ($3 \leq p \leq k$), ta sẽ thiết lập được chu trình đơn $\langle v_1, v_2, \dots, v_p, v_1 \rangle$. Mặt khác, đỉnh v_1 cũng không thể kề với đỉnh nào khác ngoài các đỉnh trên đường đi P trên bởi nếu có cạnh $(v_0, v_1) \in E$, $v_0 \notin P$ thì ta thiết lập được đường đi $\langle v_0, v_1, v_2, \dots, v_k \rangle$ dài hơn P . Vậy đỉnh v_1 chỉ có đúng một cạnh nối với v_2 , nói cách khác, v_1 là đỉnh treo. Loại bỏ v_1 và cạnh (v_1, v_2) khỏi T , ta được đồ thị mới cũng là cây và có $n - 1$ đỉnh, cây này theo giả thiết quy nạp có $n - 2$ cạnh. Vậy cây T có $n - 1$ cạnh.

2 \Rightarrow 3:

Giả sử T có k thành phần liên thông T_1, T_2, \dots, T_k . Vì T không chứa chu trình đơn nên các thành phần liên thông của T cũng không chứa chu trình đơn, tức là các T_1, T_2, \dots, T_k đều là cây. Gọi n_1, n_2, \dots, n_k lần lượt là số đỉnh của T_1, T_2, \dots, T_k thì cây T_1 có $n_1 - 1$ cạnh, cây T_2 có $n_2 - 1$ cạnh..., cây T_k có $n_k - 1$ cạnh. Cộng lại ta có số cạnh của T là $n - k$ cạnh. Theo giả thiết, cây T có $n - 1$ cạnh, suy ra $k = 1$, đồ thị chỉ có một thành phần liên thông là đồ thị liên thông.

Bây giờ khi T đã liên thông, kết hợp với giả thiết T không có chu trình nên nếu bỏ đi một cạnh bất kì thì đồ thị mới vẫn không chứa chu trình. Đồ thị mới này không thể liên thông vì nếu không nó sẽ phải là một cây và theo chứng minh trên, đồ thị mới sẽ có $n - 1$ cạnh, tức là T có n cạnh. Mâu thuẫn này chứng tỏ tất cả các cạnh của T đều là cầu.

3 \Rightarrow 4:

Gọi x và y là 2 đỉnh bất kì trong T , vì T liên thông nên sẽ có một đường đi đơn từ x tới y . Nếu tồn tại một đường đi đơn khác từ x tới y thì nếu ta bỏ đi một cạnh (u, v) nằm trên đường đi thứ nhất nhưng không nằm trên đường đi thứ hai thì từ u vẫn có thể đến được v bằng cách: đi từ u đi theo chiều tới x theo các cạnh thuộc đường thứ nhất, sau đó đi từ x tới y theo đường thứ hai, rồi lại đi từ y tới v theo các cạnh thuộc đường đi thứ nhất. Điều này chỉ ra việc bỏ đi cạnh (u, v) không ảnh hưởng tới việc có thể đi lại được giữa hai đỉnh bất kì. Mâu thuẫn với giả thiết (u, v) là cầu.

4 \Rightarrow 5:

Thứ nhất T không chứa chu trình đơn vì nếu T chứa chu trình đơn thì chu trình đó qua ít nhất hai đỉnh (u, v) . Rõ ràng dọc theo các cạnh trên chu trình đó thì từ u có hai đường đi đơn tới v . Vô lí.

Giữa hai đỉnh (u, v) bất kì của T có một đường đi đơn nối u với v , vậy khi thêm cạnh (u, v) vào đường đi này thì sẽ tạo thành chu trình.

5 \Rightarrow 6:

Gọi u và v là hai đỉnh bất kì trong T , thêm vào T một cạnh (u, v) nữa thì theo giả thiết sẽ tạo thành một chu trình chứa cạnh (u, v) . Loại bỏ cạnh này đi thì phần còn lại của chu trình sẽ là một đường đi từ u tới v . Mọi cặp đỉnh của T đều có một đường đi nối chúng tức là T liên thông, theo giả thiết T không chứa chu trình đơn nên T là cây và có $n - 1$ cạnh.

6 \Rightarrow 1:

Giả sử T không là cây thì T có chu trình, huỷ bỏ một cạnh trên chu trình này thì T vẫn liên thông, nếu đồ thị mới nhận được vẫn có chu trình thì lại huỷ một cạnh trong chu trình mới. Cứ như thế cho tới khi ta nhận được một đồ thị liên thông không có chu trình. Đồ thị này là cây nhưng lại có $< n - 1$ cạnh (vô lí). Vậy T là cây.

Định lí 5-15

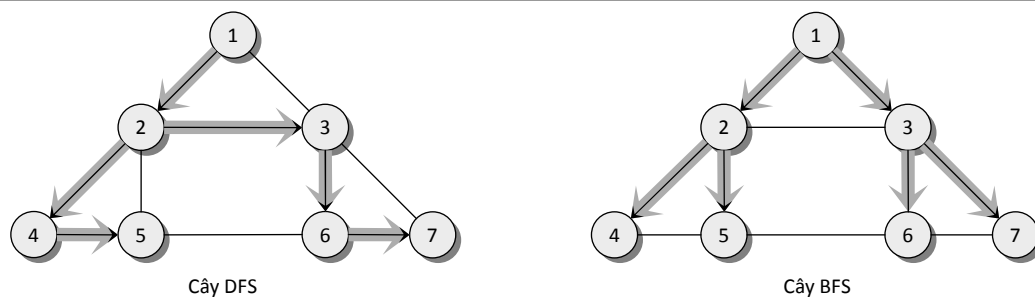
Số cây khung của đồ thị đầy đủ K_n là n^{n-2} .

Ta sẽ khảo sát hai thuật toán tìm cây khung trên đồ thị vô hướng liên thông $G = (V, E)$.

a) Xây dựng cây khung bằng thuật toán hợp nhất

Trước hết, đặt $T = (V, \emptyset)$; T không chứa cạnh nào thì có thể coi T gồm $|V|$ cây rời rạc, mỗi cây chỉ có 1 đỉnh. Sau đó xét lần lượt các cạnh của G , nếu cạnh đang xét nối hai cây khác nhau trong T thì thêm cạnh đó vào T , đồng thời hợp nhất hai cây đó lại thành một cây. Cứ làm như vậy cho tới khi kết nạp đủ $|V| - 1$ cạnh vào T thì ta được T là cây khung của đồ thị. Trong việc xây dựng cây khung bằng thuật toán hợp nhất, một cấu trúc dữ liệu biểu diễn các tập rời nhau thường được sử dụng để tăng tốc phép hợp nhất hai cây cũng như phép kiểm tra hai đỉnh có thuộc hai cây khác nhau không.

b) Xây dựng cây khung bằng các thuật toán tìm kiếm trên đồ thị.



Hình 5-25: Cây khung DFS và cây khung BFS trên cùng một đồ thị (mũi tên chỉ chiều đi thăm các đỉnh)

Áp dụng thuật toán BFS hay DFS bắt đầu từ đỉnh s nào đó, tại mỗi bước từ đỉnh u tới thăm đỉnh v , ta thêm vào thao tác ghi nhận luôn cạnh (u, v) vào cây khung. Do đồ thị liên thông nên thuật toán sẽ xuất phát từ s và tới thăm tất cả các đỉnh còn lại, mỗi đỉnh đúng một lần, tức là quá trình duyệt sẽ ghi nhận được đúng $|V| - 1$ cạnh. Tất cả những cạnh đó không tạo thành chu trình đơn bởi thuật toán không

thăm lại những đỉnh đã thăm. Theo mệnh đề tương đương thứ hai, ta có những cạnh ghi nhận được tạo thành một cây khung của đồ thị.

5.2. Tập các chu trình cơ sở của đồ thị

Xét một đồ thị vô hướng liên thông $G = (V, E)$; gọi $T = (V, E_T)$ là một cây khung của nó. Các cạnh của cây khung được gọi là các cạnh trong, còn các cạnh khác là các cạnh ngoài cây.

Nếu thêm một cạnh ngoài $e \in E - E_T$ vào cây khung T , thì ta được đúng một chu trình đơn trong T , kí hiệu chu trình này là C_e . Chu trình C_e chỉ chứa duy nhất một cạnh ngoài cây còn các cạnh còn lại đều là cạnh trong cây T

Tập các chu trình:

$$\Psi = \{C_e | e \in E - E_T\}$$

được gọi là tập các chu trình cơ sở của đồ thị G .

Các tính chất quan trọng của tập các chu trình cơ sở:

- Tập các chu trình cơ sở là phụ thuộc vào cây khung, hai cây khung khác nhau có thể cho hai tập chu trình cơ sở khác nhau.
- Cây khung của đồ thị liên thông $G = (V, E)$ luôn chứa $|V| - 1$ cạnh, còn lại $|E| - |V| + 1$ cạnh ngoài. Tương ứng với mỗi cạnh ngoài có một chu trình cơ sở, vậy số chu trình cơ sở của đồ thị liên thông là $|E| - |V| + 1$.
- Tập các chu trình cơ sở là tập nhiều nhất các chu trình thoả mãn: Mỗi chu trình có đúng một cạnh riêng, cạnh đó không nằm trong bất cứ một chu trình nào khác. Điều này có thể chứng minh được bằng cách lấy trong đồ thị liên thông một tập gồm k chu trình thoả mãn điều đó thì việc loại bỏ cạnh riêng của một chu trình sẽ không làm mất tính liên thông của đồ thị, đồng thời không ảnh hưởng tới sự tồn tại của các chu trình khác. Như vậy nếu loại bỏ tất cả các cạnh riêng thì đồ thị vẫn liên thông và còn $|E| - k$ cạnh. Đồ thị liên thông thì không thể có ít hơn $|V| - 1$ cạnh nên ta có $|E| - k \geq |V| - 1$ hay $k \leq |E| - |V| + 1$.
- Mọi cạnh trong một chu trình đơn bất kì đều phải thuộc một chu trình cơ sở. Bởi nếu có một cạnh (u, v) không thuộc một chu trình cơ sở nào, thì khi ta bỏ cạnh đó đi đồ thị vẫn liên thông và không ảnh hưởng tới sự tồn tại của các chu trình cơ sở. Lại bỏ tiếp $|E| - |V| + 1$ cạnh ngoài của các chu trình cơ sở thì đồ thị vẫn liên thông và còn lại $|V| - 2$ cạnh. Điều này vô lí.

Đối với đồ thị $G = (V, E)$ có k thành phần liên thông, ta có thể xét các thành phần liên thông và xét rùng các cây khung của các thành phần đó. Khi đó có thể mở

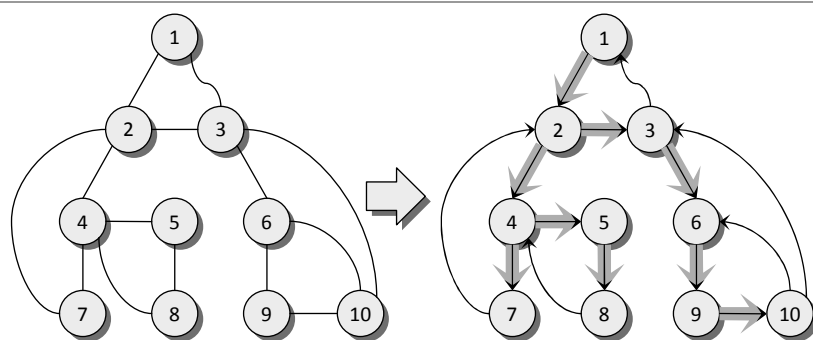
rộng khái niệm tập các chu trình cơ sở cho đồ thị vô hướng tổng quát: Mỗi khi thêm một cạnh không nằm trong các cây khung vào rừng, ta được đúng một chu trình đơn, tập các chu trình đơn tạo thành bằng cách ghép các cạnh ngoài như vậy gọi là tập các chu trình cơ sở của đồ thị G . Số các chu trình cơ sở là $|E| - |V| + k$.

5.3. Bài toán định chiều đồ thị

Bài toán đặt ra là cho một đồ thị vô hướng liên thông $G = (V, E)$, hãy thay mỗi cạnh của đồ thị bằng một cung định hướng để được một đồ thị có hướng liên thông mạnh. Nếu có phương án định chiều như vậy thì G được gọi là đồ thị định chiều được. Bài toán định chiều đồ thị có ứng dụng rõ nhất trong sơ đồ giao thông đường bộ. Chẳng hạn như trả lời câu hỏi: Trong một hệ thống đường phố, liệu có thể quy định các đường phố đó thành đường một chiều mà vẫn đảm bảo sự đi lại giữa hai nút giao thông bất kì hay không.

Có thể tổng quát hoá bài toán định chiều đồ thị: Với đồ thị vô hướng $G = (V, E)$ hãy tìm cách thay mỗi cạnh của đồ thị bằng một cung định hướng để được đồ thị mới có ít thành phần liên thông mạnh nhất. Dưới đây ta xét một tính chất hữu ích của thuật toán thuật toán tìm kiếm theo chiều sâu để giải quyết bài toán định chiều đồ thị

Xét mô hình duyệt đồ thị bằng thuật toán tìm kiếm theo chiều sâu, tuy nhiên trong quá trình duyệt, mỗi khi xét qua cạnh (u, v) thì ta định chiều luôn cạnh đó thành cung (u, v) . Nếu coi một cạnh của đồ thị tương đương với hai cung có hướng ngược chiều nhau thì việc định chiều cạnh (u, v) thành cung (u, v) tương đương với việc loại bỏ cung (v, u) của đồ thị. Ta có một phép định chiều gọi là phép định chiều DFS.



Hình 5-26. Phép định chiều DFS

Thuật toán thực hiện phép định chiều DFS có thể viết như sau:

```

procedure DFSVisit( $u \in V$ );
begin
    «Thông báo thăm  $u$  và đánh dấu  $u$  đã thăm»;
    for  $\forall v: (u, v) \in E$  do
        begin
            «Định chiều cạnh  $(u, v)$  thành cung  $(u, v) \Leftrightarrow$  xoá cung  $(v, u)$  khỏi đồ thị»;
            if « $v$  chưa thăm» then
                DFSVisit( $v$ );
            end;
        end;
    end;
begin
    «Đánh dấu mọi đỉnh đều chưa thăm»;
    for  $\forall v \in V$  do
        if « $v$  chưa thăm» then DFSVisit( $v$ );
    end;
end;

```

Thuật toán DFS sẽ cho ta một rừng các cây DFS và các cung ngoài cây. Ta có các tính chất sau:

Định lí 5-16

Sau quá trình duyệt DFS và định chiều, đồ thị sẽ chỉ còn cung DFS và cung ngược.

Chứng minh

Xét một cạnh (u, v) bất kì, không giảm tính tổng quát, giả sử rằng u được duyệt đến trước v . Theo Định lí 5-8 (định lí đường đi trắng), ta có v là hậu duệ của u . Nhìn vào mô hình cài đặt thuật toán, có nhận xét rằng việc định chiều cạnh (u, v) chỉ có thể được thực hiện trong thủ tục $DFSVisit(u)$ hoặc trong thủ tục $DFSVisit(v)$.

Nếu cạnh (u, v) được định chiều trước khi đỉnh v được duyệt đến, nghĩa là việc định chiều được thực hiện trong thủ tục $DFSVisit(u)$, và ngay sau khi cạnh (u, v) được định chiều thành cung (u, v) thì đỉnh v sẽ được thăm. Điều đó chỉ ra rằng cung (u, v) là cung DFS.

Nếu cạnh (u, v) được định chiều sau khi đỉnh v được duyệt đến, nghĩa là khi thủ tục $DFSVisit(v)$ được gọi thì cạnh (u, v) chưa định chiều. Vòng lặp bên trong thủ tục $DFSVisit(v)$ chắc chắn sẽ quét vào cạnh này và định chiều thành cung ngược (v, u) .

Trong đồ thị vô hướng ban đầu, cạnh bị định hướng thành cung ngược chính là cạnh ngoài của cây DFS. Chính vì vậy, mọi chu trình cơ sở của cây DFS trong đồ thị vô hướng ban đầu vẫn sẽ là chu trình trong đồ thị có hướng tạo ra. Đây là một phương pháp hiệu quả để liệt kê các chu trình cơ sở của cây khung DFS: Vừa duyệt DFS vừa định chiều, nếu duyệt phải cung ngược (u, v) thì truy vết đường đi của DFS để tìm đường từ v đến u , sau đó nối thêm cung ngược (u, v) để được một chu trình cơ sở.

Định lý 5-17

Điều kiện cần và đủ để một đồ thị vô hướng liên thông có thể định chiều được là mỗi cạnh của đồ thị nằm trên ít nhất một chu trình đơn (hay nói cách khác mọi cạnh của đồ thị đều không phải là cầu).

Chứng minh

Gọi $G = (V, E)$ là một đồ thị vô hướng liên thông.

" \Rightarrow "

Nếu G là định chiều được thì sau khi định hướng sẽ được đồ thị liên thông mạnh G' . Với một cạnh (u, v) được định chiều thành cung (u, v) thì sẽ tồn tại một đường đi đơn trong G' theo các cạnh định hướng từ v về u . Đường đi đó nối thêm cung (u, v) sẽ thành một chu trình đơn có hướng trong G' . Tức là trên đồ thị ban đầu, cạnh (u, v) nằm trên một chu trình đơn.

" \Leftarrow "

Nếu mỗi cạnh của G đều nằm trên một chu trình đơn, ta sẽ chứng minh rằng: phép định chiều DFS sẽ tạo ra đồ thị G' liên thông mạnh.

Lấy một cạnh (u, v) của G , vì (u, v) nằm trong một chu trình đơn, mà mọi cạnh của một chu trình đơn đều phải thuộc một chu trình cơ sở của cây DFS, nên sẽ có một chu trình cơ sở chứa cạnh (u, v) . Có thể nhận thấy rằng chu trình cơ sở của cây DFS qua phép định chiều DFS vẫn là chu trình trong G' nên theo các cung đã định hướng của chu trình đó ta có thể đi từ u tới v và ngược lại.

Lấy x và y là hai đỉnh bất kì của G , do G liên thông, tồn tại một đường đi

$$\langle x = v_0, v_1, \dots, v_k = y \rangle$$

Vì (v_i, v_{i+1}) là cạnh của G nên theo chứng minh trên, từ v_i có thể đi đến được v_{i+1} trên G' , $\forall i: 1 \leq i < k$, tức là từ x vẫn có thể đi đến y bằng các cung định hướng của G' . Suy ra G' là đồ thị liên thông mạnh.

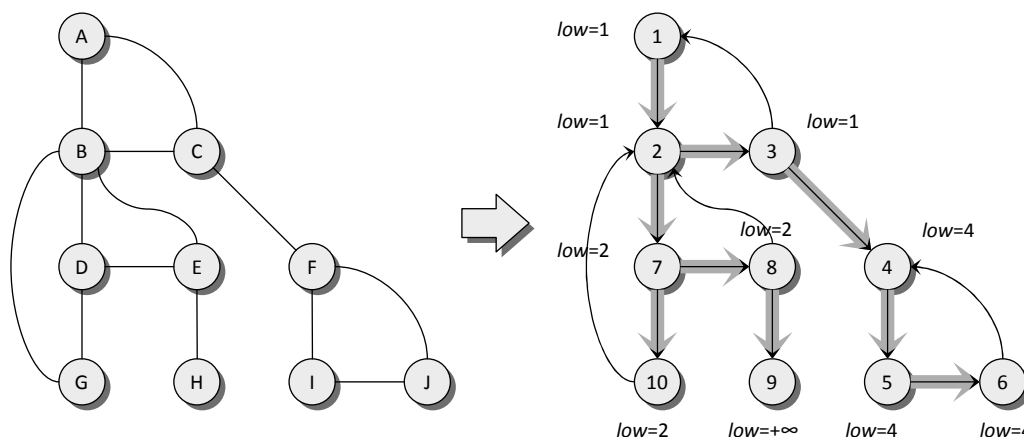
Với những kết quả đã chứng minh trên, ta còn suy ra được: Nếu đồ thị liên thông và mỗi cạnh của nó nằm trên ít nhất một chu trình đơn thì phép định chiều DFS sẽ cho một đồ thị liên thông mạnh. Còn nếu không, thì phép định chiều DFS sẽ cho một đồ thị định hướng có ít thành phần liên thông mạnh nhất, một cạnh không nằm trên một chu trình đơn nào (cầu) của đồ thị ban đầu sẽ được định hướng thành cung nối giữa hai thành phần liên thông mạnh.

5.4. Liệt kê các khớp và cầu của đồ thị

Nếu trong quá trình định chiều ta thêm vào đó thao tác đánh số các đỉnh theo thứ tự duyệt đến của thuật toán DFS, gọi $Number[u]$ là số thứ tự của đỉnh u theo cách đánh số đó. Định nghĩa thêm $Low[u]$ là giá trị $Number[.]$ nhỏ nhất của những đỉnh đến được từ nhánh DFS gốc u bằng một cung ngược. Tức là nếu nhánh DFS gốc u có nhiều cung ngược hướng lên phía gốc thì ta ghi nhận lại cung ngược hướng lên cao nhất. Nếu nhánh DFS gốc u không chứa cung ngược thì ta cho

$Low[u] := +\infty$. Cách tính các giá trị $Number[.]$ và $Low[.]$ tương tự như trong thuật toán Tarjan: Trong thủ tục $DFSVisit(u)$, trước hết ta đánh số thứ tự thăm cho đỉnh u ($Number[u]$) và khởi tạo $Low[u] := +\infty$, sau đó xét tất cả những đỉnh v kề u , định chiều cạnh (u, v) thành cung (u, v) . Có hai khả năng xảy ra:

- Nếu v chưa thăm thì ta gọi $DFSVisit(v)$ để thăm v , khi thủ tục $DFSVisit(v)$ thoát có nghĩa là đã xây dựng được nhánh DFS gốc v nằm trong nhánh DFS gốc u , những cung ngược đi từ nhánh DFS gốc v cũng là cung ngược đi từ nhánh DFS gốc $u \Rightarrow$ ta cực tiểu hoá $Low[u]$ theo công thức: $Low[u]_{\text{mới}} := \min(Low[u]_{\text{cũ}}, Low[v])$
- Nếu v đã thăm thì (u, v) là một cung ngược đi từ nhánh DFS gốc $u \Rightarrow$ ta cực tiểu hoá $Low[u]$ theo công thức: $Low[u]_{\text{mới}} := \min(Low[u]_{\text{cũ}}, Number[v])$



Hình 5-27. Cách đánh số và ghi nhận cung ngược lên cao nhất

Hãy để ý một cung DFS (u, v) (u là nút cha của nút v trên cây DFS)

- Nếu từ nhánh DFS gốc v không có cung nào ngược lên phía trên v có nghĩa là từ một đỉnh thuộc nhánh DFS gốc v đi theo các cung định hướng chỉ đi được tới những đỉnh nội bộ trong nhánh DFS gốc v mà thôi chứ không thể tới được u , suy ra (u, v) là một cầu. Cũng dễ dàng chứng minh được điều ngược lại. Vậy (u, v) là cầu nếu và chỉ nếu $Low[v] \geq Number[v]$. Như ví dụ ở Hình 5-27, ta có (C, F) và (E, H) là cầu.
- Nếu từ nhánh DFS gốc v không có cung nào ngược lên phía trên u , tức là nếu bỏ u đi thì từ v không có cách nào lên được các tiền bối của u . Điều này chỉ ra rằng nếu u không phải là nút gốc của một cây DFS thì u là khớp. Cũng không khó khăn để chứng minh điều ngược lại. Vậy nếu u không là gốc của

một cây DFS thì u là khớp nếu và chỉ nếu $Low[v] \geq Number[u]$. Như ví dụ ở Hình 5-27, ta có B, C, E và F là khớp.

- Gốc của một cây DFS thì là khớp nếu và chỉ nếu nó có từ hai 2 nhánh con trở lên. Như ví dụ ở Hình 5-27, gốc A không là khớp vì nó chỉ có một nhánh con.

Đến đây ta đã có đủ điều kiện để giải bài toán liệt kê các khớp và cầu của đồ thị: đơn giản là dùng phép định chiều DFS đánh số các đỉnh theo thứ tự thăm và ghi nhận cung ngược lên trên cao nhất xuất phát từ một nhánh cây DFS, sau đó dùng ba nhận xét kể trên để liệt kê ra tất cả các cầu và khớp của đồ thị.

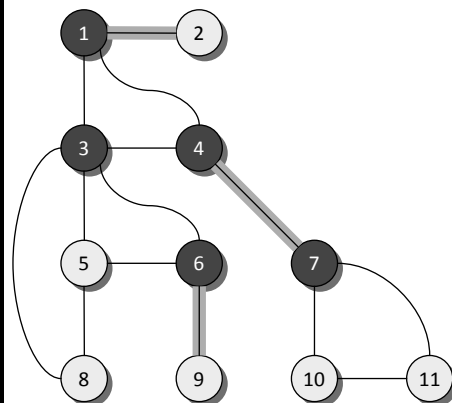
Input

- Dòng 1: Chứa số đỉnh $n \leq 1000$, số cạnh m của đồ thị vô hướng G .
- m dòng tiếp theo, mỗi dòng chứa hai số u, v tương ứng với một cạnh (u, v) của G

Output

Các khớp và cầu của G

Sample Input	Sample Output
11 14 1 2 1 3 1 4 3 4 3 5 3 6 3 8 4 7 5 6 5 8 6 9 7 10 7 11 10 11	Bridges: (1, 2) (4, 7) (6, 9) Articulations: 1 3 4 6 7



Về kĩ thuật cài đặt, ngoài các mảng đã được nói tới khi trình bày thuật toán, có thêm một mảng $Parent[1 \dots n]$, trong đó $Parent[v]$ chỉ ra nút cha của nút v trên cây DFS, nếu v là gốc của một cây DFS thì $Parent[v]$ được đặt bằng -1 . Công dụng của mảng $Parent[1 \dots n]$ là để duyệt tất cả các cung DFS và kiểm tra một đỉnh có phải là gốc của cây DFS hay không.

 CUTVE.PAS ✓ Liệt kê các khớp và cầu của đồ thị

```
{ $MODE OBJFPC }  
program ArticulationsAndBridges;
```

```

const
    maxN = 1000;
var
    a: array[1..maxN, 1..maxN] of Boolean;
    Number, Low, Parent: array[1..maxN] of Integer;
    n, Count: Integer;
procedure Enter; //Nhập dữ liệu
var
    i, m, u, v: Integer;
begin
    FillChar(a, SizeOf(a), False);
    ReadLn(n, m);
    for i := 1 to m do
        begin
            ReadLn(u, v);
            a[u, v] := True;
            a[v, u] := True;
        end;
    end;
//Hàm cực tiểu hoá: Target := min(Target, Value)
procedure Minimize(var Target: Integer; Value: Integer);
begin
    if Value < Target then Target := Value;
end;
procedure DFSVisit(u: Integer); //Thuật toán tìm kiếm theo chiều sâu bắt đầu từ u
var
    v: Integer;
begin
    Inc(Count);
    Number[u] := Count; //Đánh số u theo thứ tự duyệt đến
    Low[u] := maxN + 1; //Đặt Low[u] := +∞
    for v := 1 to n do
        if a[u, v] then //Xét các đỉnh v kề u
            begin
                a[v, u] := False; //Định chiều cạnh (u, v) thành cung (u, v)
                if Parent[v] = 0 then //Nếu v chưa thăm
                    begin
                        Parent[v] := u; //cung (u, v) là cung DFS
                        DFSVisit(v); //Đi thăm v
                        Minimize(Low[u], Low[v]); //Cực tiểu hoá Low[u] theo Low[v]
                    end
                end;
            end;
    end;
end

```



```

        else
            Minimize(Low[u], Number[v]); //Cực tiểu hoá Low[u] theo
Number[v]
        end;
    end;
end;
procedure Solve;
var
    u, v: Integer;
begin
    Count := 0; //Khởi tạo bộ đếm
    FillChar(Parent, SizeOf(Parent), 0); //Các đỉnh đều chưa thăm
    for u := 1 to n do
        if Parent[u] = 0 then
            begin
                Parent[u] := -1;
                DFSVisit(u);
            end;
    end;
end;
procedure PrintResult; //In kết quả
var
    u, v: Integer;
    nChildren: array[1..maxN] of Integer;
    IsArticulation: array[1..maxN] of Boolean;
begin
    WriteLn('Bridges: '); //Liệt kê các cầu
    for v := 1 to n do
        begin
            u := Parent[v];
            if (u <> -1) and (Low[v] >= Number[v]) then
                WriteLn('(', u, ', ', v, ')');
        end;
    WriteLn('Articulations:'); //Liệt kê các khớp
    FillChar(nChildren, n * SizeOf(Integer), 0);
    for v := 1 to n do
        begin
            u := Parent[v];
            if u <> -1 then Inc(nChildren[u]);
        end;
    //Đánh dấu các gốc cây có nhiều hơn 1 nhánh con
    for u := 1 to n do
        IsArticulation[u] := (Parent[u] = -1) and (nChildren[u]
    >= 2);

```

```

for v := 1 to n do
begin
  u := Parent[v];
  if (u <> -1) and (Parent[u] <> -1) and (Low[v] >=
Number[u]) then
    IsArticulation[u] := True; //Đánh dấu các khớp không phải gốc
cây
  end;
  for u := 1 to n do //Liệt kê
    if IsArticulation[u] then
      WriteLn(u);
  end;
begin
  Enter;
  Solve;
  PrintResult;
end.

```

Trong bài toán liệt kê các khớp và cầu của đồ thị, ta biểu diễn đồ thị bằng ma trận kề để tiện lợi cho thao tác định chiều. Nếu đồ thị có số đỉnh n lớn (không thể biểu diễn được bằng ma trận kề) và số cạnh m nhỏ (đồ thị thưa), chúng ta phải tìm một cấu trúc dữ liệu khác để biểu diễn đồ thị để chi phí về bộ nhớ và thời gian phụ thuộc chủ yếu vào m thay vì n^2 như ma trận kề. Trong các cấu trúc dữ liệu biểu diễn đồ thị phổ biến, chỉ có danh sách kề và danh sách liên thuộc cho phép thực hiện điều này, tuy nhiên việc thực hiện định chiều cạnh vô hướng thành cung có hướng sẽ trở nên khá phức tạp.

Error! Reference source not found. yêu cầu bạn sửa đổi thuật toán để bỏ đi thao tác định chiều, từ đó có thể biểu diễn đồ thị thưa bởi danh sách kề mà không còn gặp khó khăn trong việc định chiều đồ thị nữa.

5.5. Các thành phần song liên thông

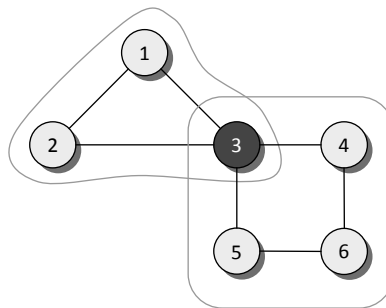
a) Các khái niệm và thuật toán

Đồ thị vô hướng liên thông được gọi là đồ thị song liên thông nếu nó không có khớp, tức là việc bỏ đi một đỉnh bất kì của đồ thị không ảnh hưởng tới tính liên thông của các đỉnh còn lại. Ta quy ước rằng đồ thị chỉ gồm một đỉnh và không có cạnh nào cũng là một đồ thị song liên thông.

Cho đồ thị vô hướng $G = (V, E)$, xét một tập con $V' \subset V$. Gọi G' là đồ thị G hạn chế trên V' . Đồ thị G' được gọi là một thành phần song liên thông của đồ thị G nếu G' song liên thông và không tồn tại đồ thị con song liên thông nào khác của G

nhận G' làm đồ thị con. Ta cũng đồng nhất khái niệm G' là thành phần song liên thông với khái niệm V' là thành phần song liên thông.

Cần phân biệt hai khái niệm đồ thị định chiều được (không có cầu) và đồ thị song liên thông (không có khớp). Nếu như đồ thị G không định chiều được thì *tập đỉnh* của G có thể phân hoạch thành các tập con rời nhau để đồ thị G hạn chế trên các tập con đó là các đồ thị định chiều được. Còn nếu đồ thị G không phải đồ thị song liên thông thì *tập cạnh* của G có thể phân hoạch thành các tập con rời nhau để trên mỗi tập con, các cạnh và các đỉnh đầu mút của chúng trở thành một đồ thị song liên thông. Hai thành phần song liên thông có thể có chung một điểm khớp nhưng không có cạnh nào chung



Hình 5-28. Đồ thị và hai thành phần song liên thông có chung khớp

Xét mô hình định chiều đồ thị đánh số đỉnh theo thứ tự duyệt đến và ghi nhận cung ngược lên cao nhất...

```

procedure DFSVisit(u ∈ V);
begin
    Count := Count + 1;
    Number[u] := Count; //Đánh số u theo thứ tự duyệt đến
    Low[u] := +∞;
    for ∀v ∈ V: (u, v) ∈ E do
        begin
            «Định chiều cạnh (u, v) thành cung (u, v)»;
            if Number[v] > 0 then //v đã thăm
                Low[u] := min(Low[u], Number[v])
            else //v chưa thăm
                begin
                    DFSVisit(v); //Đi thăm v
                    Low[u] := min(Low[u], Low[v]); //Cập nhật Low[u]
                end;
            end;
        end;
end;

```

```

end;
begin
  Count := 0;
  for  $\forall v \in V$  do Number[v] := 0; //Number[v] = 0  $\leftrightarrow$  v chưa thăm
  for  $\forall v \in V$  do
    if Number[v] = 0 then DFSVisit(v);
  end.

```

Trong thủ tục $DFSVisit(u)$, mỗi khi xét các đỉnh v kề u chưa được thăm, thuật toán sẽ gọi $DFSVisit(v)$ để đi thăm v sau đó cực tiểu hoá $Low[u]$ theo $Low[v]$. Tại thời điểm này, nếu $Low[v] \geq Number[u]$ thì hoặc u là khớp hoặc u là gốc của một cây DFS. Để tiện, trong trường hợp này ta gọi cung (u, v) là *cung chốt* của thành phần song liên thông.

Thuật toán tìm kiếm theo chiều sâu không chỉ duyệt qua các đỉnh mà còn duyệt và định chiều các cung nữa. Ta sẽ quan tâm tới cả thời điểm một cạnh được duyệt đến, duyệt xong, cũng như thứ tự tiền bối-hậu duệ của các cung DFS: Cung DFS (u, v) được coi là tiền bối thực sự của cung DFS (u', v') (hay cung (u', v') là hậu duệ thực sự của cung (u, v)) nếu cung (u', v') nằm trong nhánh DFS gốc v . Xét về vị trí trên cây, cung (u', v') nằm dưới cung (u, v) .

Có thể nhận thấy rằng nếu (u, v) là một cung chốt thỏa mãn: Khi $DFSVisit(u)$ gọi $DFSVisit(v)$ và quá trình tìm kiếm theo chiều sâu tiếp tục từ v không thăm tiếp bất cứ một cung chốt nào (tức là nhánh DFS gốc v không chứa cung chốt nào) thì cung (u, v) hợp với tất cả các cung hậu duệ của nó sẽ tạo thành một nhánh cây mà mọi đỉnh thuộc nhánh cây đó là một thành phần song liên thông. Chính vì vậy thuật toán liệt kê các thành phần song liên thông có tư tưởng khá giống với thuật toán Tarjan tìm thành phần liên thông mạnh. Việc cài đặt thuật toán liệt kê các thành phần song liên thông chính là sự sửa đổi đối ngẫu của thuật toán Tarjan: Thay khái niệm “chốt” bằng “cung chốt” và thay vì dùng ngăn xếp chứa chốt và các đỉnh hậu duệ của chốt để liệt kê các thành phần liên thông mạnh, chúng ta sẽ dùng ngăn xếp chứa cung chốt và các hậu duệ của cung chốt để liệt kê các thành phần song liên thông.

Vấn đề rắc rối duy nhất gặp phải là quy ước một đỉnh cô lập của đồ thị cũng là một thành phần song liên thông. Nếu thực hiện thuật toán trên, thành phần song liên thông chỉ gồm duy nhất một đỉnh sẽ không có cung chốt nào cả và như vậy sẽ bị sót khi liệt kê. Ta sẽ phải xử lý các đỉnh cô lập như trường hợp riêng khi liệt kê các thành phần song liên thông của đồ thị.

```

procedure DFSVisit(u  $\in$  V);

```

```

begin
    Count := Count + 1;
    Number[u] := Count; //Đánh số u theo thứ tự duyệt đến
    Low[u] := +∞;
    for  $\forall v \in V: (u, v) \in E$  do
        begin
            «Định chiều cạnh (u, v) thành cung (u, v)»;
            if Number[v] > 0 then //v đã thăm
                Low[u] := min(Low[u], Number[v])
            else //v chưa thăm
                begin
                    Push((u, v)); //Đẩy cung (u, v) vào ngăn xếp
                    DFSVisit(v); //Đi thăm v
                    Low[u] := min(Low[u], Low[v]); //Cập nhật Low[u]
                    if Low[v] ≥ Number[u] then //[(u, v) là cung chốt]
                        begin
                            «Thông báo thành phần song liên thông với cung
chốt (u, v)»;
                            repeat
                                (p, q) := Pop; //Lấy từ ngăn xếp ra một cung (p, q)
                                Output ← q; //Liệt kê các đỉnh nên chỉ cần xuất ra một đầu mút
                            until (p, q) = (u, v);
                                Output ← u; //Còn thiếu đỉnh u, liệt kê nốt
                            end;
                        end;
                    end;
                end;
            end;
        end;
    begin
        Count := 0;
        for  $\forall v \in V$  do Number[v] := 0; //Number[v] = 0 ↔ v chưa thăm
        Stack := ∅;
        for  $\forall v \in V$  do
            if Number[v] = 0 then
                begin
                    DFSVisit(v);
                    if «v là đỉnh cô lập» then
                        «Liệt kê thành phần song liên thông chỉ gồm một
đỉnh v»
                    end;
                end;
            end;
        end.
    
```

b) Cài đặt

Về kĩ thuật cài đặt không có gì mới, có một chú ý nhỏ là chúng ta chỉ dùng ngăn xếp *Stack* để chứa các cung DFS, vì vậy *Stack* không bao giờ phải chứa quá $n - 1$ cung

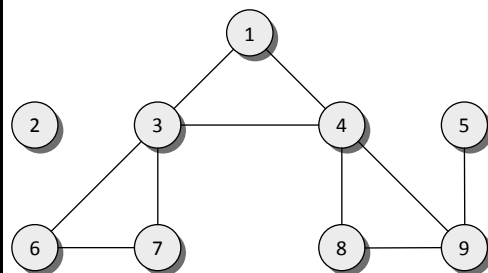
Input

- Dòng 1: Chứa số đỉnh $n \leq 1000$ và số cạnh m của một đồ thị vô hướng
- m dòng tiếp theo, mỗi dòng chứa hai số u, v tương ứng với một cạnh (u, v) của đồ thị.

Output

Các thành phần song liên thông của đồ thị

Sample Input	Sample Output
9 10	Biconnected
1 3	component: 1
1 4	5, 9
3 4	Biconnected
3 6	component: 2
3 7	9, 8, 4
4 8	Biconnected
4 9	component: 3
5 9	7, 6, 3
6 7	Biconnected
8 9	component: 4
	4, 3, 1
	Biconnected
	component: 5
	2



BCC.PAS ✓ Liệt kê các thành phần song liên thông

```
{ $MODE OBJFPC }
program BiconnectedComponents;
const
  maxN = 1000;
type
  TStack = record
    x, y: array[1..maxN - 1] of Integer;
    Top: Integer;
  end;
var
  a: array[1..maxN, 1..maxN] of Boolean;
```

```

Number, Low: array[1..maxN] of Integer;
Stack: TStack;
BCC, PrevCount, Count, n, u: Integer;
procedure Enter; //Nhập dữ liệu
var
  i, m, u, v: Integer;
begin
  FillChar(a, SizeOf(a), False);
  ReadLn(n, m);
  for i := 1 to m do
    begin
      ReadLn(u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
  end;
procedure Push(u, v: Integer); //Đẩy một cung (u, v) vào ngăn xếp
begin
  with Stack do
    begin
      Inc(Top);
      x[Top] := u;
      y[Top] := v;
    end;
end;
procedure Pop(var u, v: Integer); //Lấy một cung (u, v) khỏi ngăn xếp
begin
  with Stack do
    begin
      u := x[Top];
      v := y[Top];
      Dec(Top);
    end;
end;
//Hàm cực tiểu hoá: Target := min(Target, Value)
procedure Minimize(var Target: Integer; Value: Integer);
begin
  if Value < Target then Target := Value;
end;
procedure DFSVisit(u: Integer); //Thuật toán tìm kiếm theo chiều sâu
var
  v, p, q: Integer;

```

```

begin
  Inc(Count);
  Number[u] := Count;
  Low[u] := maxN + 1;
  for v := 1 to n do
    if a[u, v] then //Xét mọi cạnh (u, v)
      begin
        a[v, u] := False; //Định chiều luôn
        if Number[v] <> 0 then //v đã thăm
          Minimize(Low[u], Number[v])
        else //v chưa thăm
          begin
            Push(u, v); //Đẩy cung DFS (u, v) vào Stack
            DFSVisit(v); //Tiếp tục quá trình DFS từ v
            Minimize(Low[u], Low[v]);
            if Low[v] >= Number[u] then //Nếu (u, v) là cung chốt
              begin //Liệt kê thành phần song liên thông với cung chốt (u, v)
                Inc(BCC);
                WriteLn('Biconnected component: ', BCC);
                repeat
                  Pop(p, q); //Lấy một cung DFS (p, q) khỏi Stack
                  Write(q, ', '); //Chỉ in ra một đầu cung, tránh in lặp
                until (p = u) and (q = v); //Đến khi lấy ra cung (u, v)
              end;
            WriteLn(u); //In nốt ra đỉnh u
          end;
        end;
      end;
  end;
end;
begin
  Enter;
  FillChar(Number, n * SizeOf(Integer), 0);
  Stack.Top := 0;
  Count := 0;
  BCC := 0;
  for u := 1 to n do
    if Number[u] = 0 then
      begin
        PrevCount := Count;
        DFSVisit(u);
        if Count = PrevCount + 1 then //u là đỉnh cô lập
          begin

```



```
Inc(BCC);  
WriteLn('Biconnected component: ', BCC);  
WriteLn(u);  
end;  
end;  
end.
```

Bài tập

- 5.20.** Hãy sửa đổi thuật toán liệt kê khớp và cầu của đồ thị, sửa đổi thuật toán liệt kê các thành phần song liên thông sao cho không cần phải thực hiện việc định chiều đồ thị nữa (Bởi vì việc định chiều một đồ thị tỏ ra khá cồng kềnh và không hiệu quả nếu đồ thị được biểu diễn bằng danh sách kề hay danh sách cạnh)
- 5.21.** Tìm thuật toán đếm số cây khung của đồ thị (Hai cây khung gọi là khác nhau nếu chúng có ít nhất một cạnh khác nhau)

6. Đồ thị Euler và đồ thị Hamilton

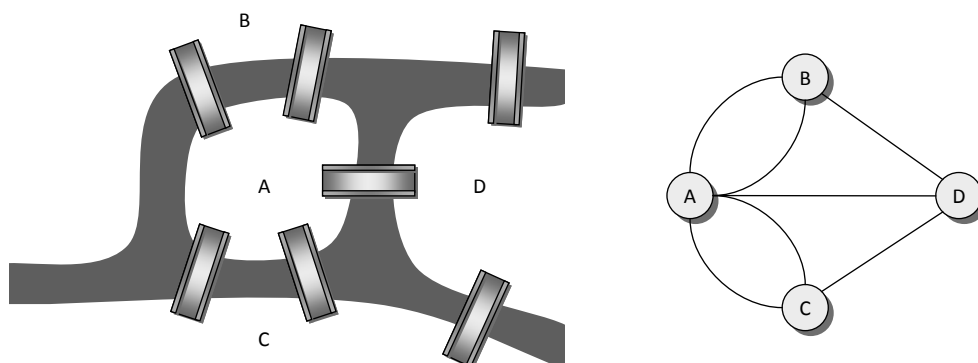
6.1. Đồ thị Euler

a) Bài toán

Bài toán về đồ thị Euler được coi là bài toán đầu tiên của lý thuyết đồ thị. Bài toán này xuất phát từ một bài toán nổi tiếng: Bài toán bảy cây cầu ở Königsberg:

Thành phố Königsberg thuộc Đức (nay là Kaliningrad thuộc Cộng hoà Nga), được chia làm 4 vùng bằng các nhánh sông Pregel. Các vùng này gồm 2 vùng bên bờ sông (B, C), đảo Kneiphof (A) và một miền nằm giữa hai nhánh sông Pregel (D). Vào thế kỉ XVIII, người ta đã xây 7 chiếc cầu nối những vùng này với nhau. Người dân ở đây tự hỏi: Liệu có cách nào xuất phát tại một địa điểm trong thành phố, đi qua 7 chiếc cầu, mỗi chiếc đúng 1 lần rồi quay trở về nơi xuất phát không?

Nhà toán học Thụy sĩ Leonhard Euler đã giải bài toán này và có thể coi đây là ứng dụng đầu tiên của Lý thuyết đồ thị, ông đã mô hình hoá sơ đồ 7 cái cầu bằng một đa đồ thị, bốn vùng được biểu diễn bằng 4 đỉnh, các cầu là các cạnh. Bài toán tìm đường qua 7 cầu, mỗi cầu đúng một lần có thể tổng quát hoá bằng bài toán: Có tồn tại chu trình trong đa đồ thị đi qua tất cả các cạnh và mỗi cạnh đúng một lần.



Hình 5-29: Mô hình đồ thị của bài toán bảy cái cầu

Chu trình qua tất cả các cạnh của đồ thị, mỗi cạnh đúng một lần được gọi là *chu trình Euler* (*Euler circuit/Euler circle/Euler tour*). Đường đi qua tất cả các cạnh của đồ thị, mỗi cạnh đúng một lần gọi là *đường đi Euler* (*Euler path/Euler trail/Euler walk*). Một đồ thị có chu trình Euler được gọi là *đồ thị Euler* (*Eulerian graph/unicursal graph*). Một đồ thị có đường đi Euler được gọi là *đồ thị nửa Euler* (*Semi-Eulerian graph/Traversable graph*).

b) Các định lý và thuật toán

Định lý 5-18 (Euler)

|| Một đồ thị vô hướng liên thông $G = (V, E)$ có chu trình Euler khi và chỉ khi mọi đỉnh của nó đều có bậc chẵn.

Chứng minh

Nếu G có chu trình Euler thì khi đi dọc chu trình đó, mỗi khi đi qua một đỉnh thì bậc của đỉnh đó tăng lên 2 (một lần vào + một lần ra). Chu trình Euler lại đi qua tất cả các cạnh nên suy ra mọi đỉnh của đồ thị đều có bậc chẵn.

Ngược lại nếu G liên thông và mọi đỉnh đều có bậc chẵn, ta sẽ chỉ ra thuật toán xây dựng chu trình Euler trên G .

Xuất phát từ một đỉnh bất kì, ta đi sang một đỉnh tùy ý kế nó, đi qua cạnh nào xóa luôn cạnh đó cho tới khi không đi được nữa, có thể nhận thấy rằng sau mỗi bước đi, chỉ có đỉnh đầu và đỉnh cuối của đường đi có bậc lẻ còn mọi đỉnh khác trong đồ thị đều có bậc chẵn. Cạnh cuối cùng đi qua chắc chắn là đi tới một đỉnh bậc lẻ, vì nếu là cạnh đi tới một đỉnh bậc chẵn thì đỉnh này sẽ có ít nhất 2 cạnh liên thuộc, và như vậy khi đi tới đỉnh này và xóa cạnh vào ta vẫn còn một cạnh để ra, quá trình đi chưa kết thúc. Điều này chỉ ra rằng cạnh cuối cùng bắt buộc phải đi về nơi xuất phát tức là chúng ta có một chu trình C . Cũng dễ dàng nhận thấy rằng khi quá trình này kết thúc, mọi đỉnh của G vẫn có bậc chẵn.

Nếu G còn lại cạnh liên thuộc với một đỉnh v nào đó trên C thì lại bắt đầu từ v , ta đi một cách tùy ý theo các cạnh còn lại của G ta sẽ được một chu trình C' bắt đầu từ v và kết thúc

tại v . Thay thế một bước đi qua đỉnh v trên C bằng cả chu trình C' , ta sẽ được một chu trình mới lớn hơn. Quy trình được lặp lại cho tới khi C không còn đỉnh nào có cạnh liên thuộc nằm ngoài C . Do tính liên thông của G , điều này có nghĩa là C chứa tất cả các cạnh của G hay C là chu trình Euler trên đồ thị ban đầu.

Hệ quả

Một đồ thị vô hướng liên thông $G = (V, E)$ có đường đi Euler khi và chỉ khi nó có đúng 2 đỉnh bậc lẻ.

Chứng minh

Nếu G có đường đi Euler thì chỉ có đỉnh bắt đầu và đỉnh kết thúc đường đi có bậc lẻ còn mọi đỉnh khác đều có bậc chẵn. Ngược lại nếu đồ thị liên thông có đúng 2 đỉnh bậc lẻ thì ta thêm vào một cạnh giả nối hai đỉnh bậc lẻ đó và tìm chu trình Euler. Loại bỏ cạnh giả khỏi chu trình, chúng ta sẽ được đường đi Euler.

Định lý 5-19

Một đồ thị có hướng liên thông yếu $G = (V, E)$ có chu trình Euler thì mọi đỉnh của nó có bán bậc ra bằng bán bậc vào: $\deg^+(v) = \deg^-(v), \forall v \in V$; Ngược lại, nếu G liên thông yếu và mọi đỉnh của nó có bán bậc ra bằng bán bậc vào, thì G có chu trình Euler (suy ra G sẽ là liên thông mạnh).

Chứng minh

Tương tự như phép chứng minh Định lý 5.18.

Hệ quả

Một đồ thị có hướng liên thông yếu $G = (V, E)$ có đường đi Euler nhưng không có chu trình Euler nếu tồn tại đúng hai đỉnh $s, t \in V$ sao cho:

$$\deg^+(s) - \deg^-(s) = \deg^-(t) - \deg^+(t) = 1$$

còn tất cả những đỉnh còn lại của đồ thị đều có bán bậc ra bằng bán bậc vào.

Việc chứng minh Định lý 5-18 (Euler) cho ta một thuật toán hữu hiệu để chỉ ra chu trình Euler trên đồ thị Euler. Thuật toán này hoạt động dựa trên một ngăn xếp *Stack* và được mô tả cụ thể như sau: Bắt đầu từ đỉnh 1, ta đi thoải mái theo các cạnh của đồ thị cho tới khi không đi được nữa, đi tới đỉnh nào ta đẩy đỉnh đó vào ngăn xếp và đi qua cạnh nào thì ta xóa cạnh đó khỏi đồ thị. Khi không đi được nữa thì ngăn xếp sẽ chứa các đỉnh trên một chu trình C bắt đầu và kết thúc ở đỉnh 1. Sau đó chúng ta lấy lần lượt các đỉnh ra khỏi ngăn xếp tương đương với việc đi ngược chu trình C . Nếu đỉnh được lấy ra (u) không có cạnh nào còn lại liên thuộc với nó thì u sẽ được ghi ra chu trình Euler, ngược lại, nếu u vẫn còn có cạnh liên thuộc thì ta lại đi tiếp từ u theo cách trên và đẩy thêm vào ngăn xếp một chu trình

C' bắt đầu và kết thúc tại u , để khi lấy các đỉnh ra khỏi ngăn xếp sẽ tương đương với việc đi ngược lại chu trình C' rồi tiếp tục đi ngược phần còn lại của chu trình C trong ngăn xếp... Có thể hình dung là thuật toán lần ngược chu trình C , khi đến đỉnh u thì thay u bằng cả một chu trình C' ...

Khi cài đặt thuật toán, chúng ta cần trang bị ba phép toán trên ngăn xếp *Stack*:

- *Push(v)*: Đẩy một đỉnh v vào *Stack*
- *Pop*: Lấy ra một đỉnh khỏi *Stack*
- *Get*: Đọc phần tử ở đỉnh *Stack*

```
Stack := (1); //Ngăn xếp ban đầu chỉ chứa một đỉnh bất kì,
chẳng hạn đỉnh 1
repeat
  u := Get; //Đọc phần tử ở đỉnh ngăn xếp
  if  $\exists (u, v) \in E$  then //Từ u còn đi tiếp được
    begin
      Push(v);
      E := E - {(u, v)}; //Xoá cạnh (u, v) khỏi đồ thị
    end;
  else //Từ u không đi đâu được nữa
    begin
      u := Pop; //Lấy u khỏi ngăn xếp
      Output  $\leftarrow$  u; //In ra u
    end;
until Stack =  $\emptyset$ ; //Lặp tới khi ngăn xếp rỗng
```

c) Cài đặt

Dưới đây chúng ta sẽ cài đặt thuật toán tìm chu trình Euler trên đa đồ thị Euler vô hướng $G = (V, E)$. Dữ liệu vào luôn đảm bảo đồ thị liên thông, có ít nhất một đỉnh và mọi đỉnh đều có bậc chẵn.

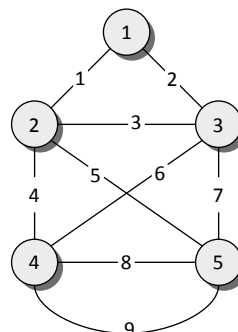
Input

- Dòng 1 chứa số đỉnh $n \leq 10^5$ và số cạnh $m \leq 10^6$
- m dòng tiếp, mỗi dòng chứa số hiệu hai đầu mút của một cạnh.

Output

Chu trình Euler

Sample Input	Sample Output
5 9	1 2 4 5 4 3 5 2 3 1
1 2	
1 3	
2 3	
2 4	
2 5	
3 4	
3 5	
4 5	
4 5	



Ngoài các thao tác đối với ngăn xếp, thuật toán tìm chu trình Euler còn yêu cầu cài đặt hai thao tác sau đây một cách hiệu quả:

- Với mỗi đỉnh kiểm tra xem có tồn tại cạnh liên thuộc với nó hay không, nếu có thì chỉ ra một cạnh liên thuộc.
- Loại bỏ một cạnh khỏi đồ thị

Các cạnh của đồ thị được đánh số từ 1 tới m , sau đó mỗi cạnh vô hướng (x, y) sẽ được thay thế bởi hai cung có hướng ngược chiều: (x, y) và (y, x) . Mỗi cung là một bản ghi gồm hai đỉnh đầu mút và chỉ số cạnh vô hướng tương ứng.

```

const
    maxM = 1000000;
type
    TArc = record
        x, y: Integer; //cung (x, y)
        edge: Integer; //chỉ số cạnh vô hướng tương ứng
    end;
var
    a: array[1..2 * maxM] of TArc;

```

Danh sách liên thuộc được xây dựng theo kiểu reverse star: Mỗi đỉnh u cho tương ứng với một danh sách các cung đi vào u . Các danh sách này được cho bởi hai mảng $head[1 \dots n]$ và $link[1 \dots 2m]$ trong đó:

- $head[u]$ là chỉ số cung đầu tiên trong danh sách liên thuộc các cung đi vào u , trường hợp đỉnh u không còn cung đi vào, $head[u]$ được gán bằng 0.
- $link[i]$ là chỉ số cung kế tiếp cung a_i trong cùng danh sách liên thuộc chứa cung a_i , trường hợp a_i là cung cuối cùng trong một danh sách liên thuộc, $link[i]$ được gán bằng 0.

Để thực hiện thao tác xoá cạnh, ta duy trì một mảng đánh dấu $deleted[1 \dots m]$ trong đó $deleted[i] = \text{True}$ nếu cạnh vô hướng thứ i đã bị xoá. Mỗi khi cạnh vô hướng bị xoá, cả hai cung có hướng tương ứng đều không còn tồn tại, việc kiểm tra một cung có hướng a_i còn tồn tại hay không có thể thực hiện bằng việc kiểm tra: $deleted[a_i.edge] \stackrel{?}{=} \text{False}$.

Chúng ta sẽ cài đặt các thao tác sau trên cấu trúc dữ liệu:

- Hàm *Get*: Trả về phần tử nằm ở đỉnh ngăn xếp.
- Hàm *Pop*: Trả về phần tử nằm ở đỉnh ngăn xếp và rút phần tử đó khỏi ngăn xếp.
- Thủ tục *Push(v)*: Đẩy một đỉnh v vào ngăn xếp.

Tất cả các thao tác trên ngăn xếp có thể cài đặt để thực hiện trong thời gian $O(1)$. Thuật toán tìm chu trình Euler có thể viết cụ thể hơn:

```
Stack := (1); //Khởi tạo ngăn xếp chỉ chứa một đỉnh
repeat
  u := Get; //Đọc đỉnh u từ ngăn xếp
  i := head[u]; //Xét cung a[i] đứng đầu danh sách liên thuộc
  các cung đi vào u
  while (i > 0) and (deleted[a[i].edge]) do //cung a[i] ứng
  với cạnh vô hướng đã xoá
    i := link[i]; //Dịch sang cung kế tiếp
  head[u] := i; //Những cung đã duyệt qua bị loại ngay, cập
  nhật lại chỉ số đầu danh sách liên thuộc
  if i > 0 then //u còn cung đi vào ứng với cạnh vô hướng
  chưa xoá
    begin
      Push(a[i].x); //Đẩy đỉnh nối tới u vào ngăn xếp (đi
  ngược cung a[i])
      Deleted[a[i].edge] := True; //Xoá ngay cạnh vô hướng
  ứng với cung a[i]
    end
  else
    Output ← Pop;
until Top = 0; //Lặp tới khi ngăn xếp rỗng
```

Xét vòng lặp repeat...until, mỗi bước lặp có một thao tác *Push* hoặc *Pop* được thực hiện. Mỗi lần thao tác *Push* được thực hiện phải có một cạnh vô hướng bị xoá và ngăn xếp có thêm một đỉnh. Mỗi lần thao tác *Pop* được thực hiện thì ngăn xếp bị bớt đi một đỉnh. Vì thuật toán in ra $m + 1$ đỉnh trên chu trình Euler nên sẽ phải có tổng cộng $m + 1$ thao tác *Pop*. Trước khi vào vòng lặp ngăn xếp có một

đỉnh và khi vòng lặp kết thúc ngăn xếp trở thành rỗng, suy ra số thao tác *Push* phải là m . Từ đó, vòng lặp repeat...until thực hiện $2m + 1$ lần.

Tiếp theo ta đánh giá số thao tác duyệt danh sách liên thuộc của đỉnh u . Bởi sau vòng lặp while có lệnh cập nhật $head[u] := i$ nên có thể thấy rằng lệnh gán $i := link[i]$ được thực hiện bao nhiêu lần thì danh sách liên thuộc của u bị giảm đi đúng chừng đó cung. Tổng số phần tử của các danh sách liên thuộc là $2m$ và khi thuật toán kết thúc, các danh sách liên thuộc đều rỗng. Suy ra tổng thời gian thực hiện phép duyệt danh sách liên thuộc (vòng lặp while) trong toàn bộ thuật toán là $\Theta(m)$.

Suy ra thời gian thực hiện giải thuật là $\Theta(m)$.

EULER.PAS ✓ Tìm chu trình Euler trong đa đồ thị Euler vô hướng

```
{ $MODE OBJFPC }
program EulerTour;
const
  maxN = 1000000;
  maxM = 10000000;
type
  TArc = record //Cấu trúc một cung
    x, y: Integer; //Đỉnh đầu và đỉnh cuối
    edge: Integer; //Chỉ số cạnh vô hướng tương ứng
  end;
var
  n, m: Integer;
  a: array[1..2 * maxM] of TArc; //Danh sách các cung
  link: array[1..2 * maxM] of Integer; //link[i]: Chỉ số cung kế tiếp a[i]
  trong cùng danh sách liên thuộc
  head: array[1..maxN] of Integer; //head[u]: chỉ số cung đầu tiên trong
  danh sách các cung đi vào u
  deleted: array[1..maxM] of Boolean; //Đánh dấu cạnh vô hướng bị xóa
  hay chưa
  Stack: array[1..maxM + 1] of Integer; //Ngăn xếp
  Top: Integer; //Phần tử đỉnh ngăn xếp
procedure Enter; //Nhập dữ liệu và xây dựng danh sách liên thuộc
var
  i, j, u, v: Integer;
begin
  ReadLn(n, m);
  j := 2 * m;
  for i := 1 to m do
```

```

begin
    ReadLn(u, v); //Đọc một cạnh vô hướng, thêm 2 cung có hướng tương ứng
    a[i].x := u; a[i].y := v; a[i].edge := i;
    a[j].x := v; a[j].y := u; a[j].edge := i;
    Dec(j);
end;
FillChar(head[1], n * SizeOf(head[1]), 0); //Khởi tạo các danh
sách liên thuộc rỗng
for i := 2 * m downto 1 do
    with a[i] do //Duyệt từng cung (x, y)
        begin //Đưa cung đó vào danh sách liên thuộc các cung đi vào y
            link[i] := head[y];
            head[y] := i;
        end;
    FillChar(deleted[1], n * SizeOf(deleted[1]), False); //Các
cạnh vô hướng đều chưa xoá
end;
procedure FindEulerTour;
var
    u, i: Integer;
begin
    Top := 1; Stack[1] := 1; //Khởi tạo ngăn xếp chứa đỉnh 1
    repeat
        u := Stack[Top]; //Đọc phần tử ở đỉnh ngăn xếp
        i := head[u]; //Cung a[i] đang đứng đầu danh sách liên thuộc
        while (i > 0) and (deleted[a[i].edge]) do
            i := link[i]; //Dịch chỉ số i dọc danh sách liên thuộc để tìm cung ứng với
cạnh vô hướng chưa xoá
        head[u] := i; //Cập nhật lại head[u], "nhảy" qua các cung ứng với cạnh vô
hướng đã xoá
        if i > 0 then //u còn cung đi vào ứng với cạnh vô hướng chưa xoá
            begin
                Inc(Top); Stack[Top] := a[i].x; //Đi ngược cung a[i], đẩy đỉnh
nối tới u vào ngăn xếp
                Deleted[a[i].edge] := True; //Xoá cạnh vô hướng tương ứng với a[i]
            end
        else //u không còn cung đi vào
            begin
                Write(u, ' '); //In ra u trên chu trình Euler
                Dec(Top); //Lấy u khỏi ngăn xếp
            end
        until Top = 0; //Lặp tới khi ngăn xếp rỗng
    WriteLn;

```



```

end;
begin
  Enter;
  FindEulerTour;
end.

```

d) Vài nhận xét

Bằng việc quan sát hoạt động của ngăn xếp, chúng ta có thể sửa mô hình cài đặt của thuật toán nhằm tận dụng chính ngăn xếp của chương trình con đệ quy chứ không cần cài đặt cấu trúc dữ liệu ngăn xếp để chứa các đỉnh:

```

procedure Visit(u: Integer);
var
  i: Integer;
begin
  i := head[u];
  while i ≠ 0 do
    begin //Xét cung a[i] đi vào u
      if not deleted[a[i].edge] then //Cạnh vô hướng tương ứng chưa bị xoá
        begin
          deleted[a[i].edge] := True; //Xoá cạnh vô hướng tương ứng
          Visit(a[i].x); //Đi ngược chiều cung a[i] thăm đỉnh nối tới u
        end;
      end;
    Output ← u; //Từ u không thể đi ngược chiều cung nào nữa, in ra u trên chu trình Euler
  end;
begin
  «Nhập đồ thị và xây dựng danh sách liên thuộc»;
  Visit(1); //Khởi động thuật toán tìm chu trình Euler
end.

```

Cách cài đặt này khá đơn giản vì thao tác trên ngăn xếp được thực hiện tự nhiên qua cơ chế gọi và thoát thủ tục đệ quy. Tuy nhiên cần chú ý rằng độ sâu của dây chuyền đệ quy có thể lên tới $m + 1$ cấp nên với một số công cụ lập trình cần đặt lại dung lượng bộ nhớ Stack¹.

Chúng ta có thể liên hệ thuật toán này với thuật toán tìm kiếm theo chiều sâu: Từ mô hình DFS, nếu thay vì đi *thăm đỉnh* chúng ta đi *thăm cạnh* (một cạnh có thể đi tiếp sang cạnh chung đầu mút với nó). Đồng thời ta đánh dấu cạnh đã qua/chưa

¹ Trong Free Pascal 32 bit, dung lượng bộ nhớ Stack dành cho biến địa phương và tham số chương trình con mặc định là 64 KiB. Có thể đặt lại bằng dẫn hướng biên dịch {\$M...}

qua thay cho cơ chế đánh dấu một đỉnh đã thăm/chưa thăm. Khi đó *thứ tự duyệt xong (finish)* của các cạnh cho ta một chu trình Euler.

Thuật toán không có gì sai nếu ta xây dựng danh sách liên thuộc kiểu forward star thay vì kiểu reverse star. Tuy nhiên ta chọn kiểu reverse star bởi cách biểu diễn này thích hợp để tìm chu trình Euler trên cả đồ thị vô hướng và có hướng.

Người ta còn có thuật toán Fleury (1883) để tìm chu trình Euler bằng tay: Bắt đầu từ một đỉnh, chúng ta đi thoải mái theo các cạnh theo nguyên tắc: xoá bỏ các cạnh đi qua và chỉ đi qua cầu khi không còn cách nào khác để chọn. Khi không thể đi tiếp được nữa thì đường đi tìm được chính là chu trình Euler.

Bằng cách “lạm dụng thuật ngữ”, ta có thể mô tả được thuật toán tìm Fleury cho cả đồ thị Euler có hướng cũng như vô hướng:

- Dưới đây nếu ta nói cạnh (u, v) thì hiểu là cạnh (u, v) trên đồ thị vô hướng, hiểu là cung (u, v) trên đồ thị có hướng.
- Ta gọi cạnh (u, v) là “một đi không trở lại” nếu như từ u đi tới v , sau đó xoá cạnh này đi thì không có cách nào từ v quay lại u .

Thuật toán Fleury tìm chu trình Euler: Xuất phát từ một đỉnh, ta đi một cách tùy ý theo các cạnh tuân theo hai nguyên tắc: Xoá bỏ cạnh vừa đi qua và chỉ chọn cạnh “một đi không trở lại” nếu như không còn cạnh nào khác để chọn.

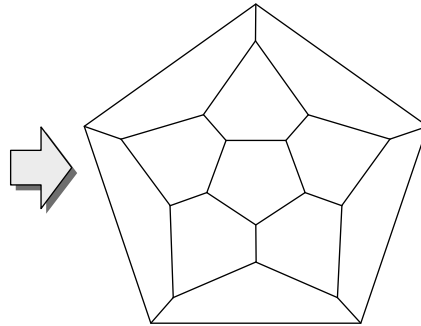
Thuật toán Fleury là một thuật toán thích hợp cho việc tìm chu trình Euler bằng tay (với những đồ thị vẽ ra được trên mặt phẳng thì việc kiểm tra cầu bằng mắt thường là tương đối dễ dàng). Tuy vậy khi cài đặt thuật toán trên máy tính thì thuật toán này tỏ ra không hiệu quả.

6.2. Đồ thị Hamilton

a) Bài toán

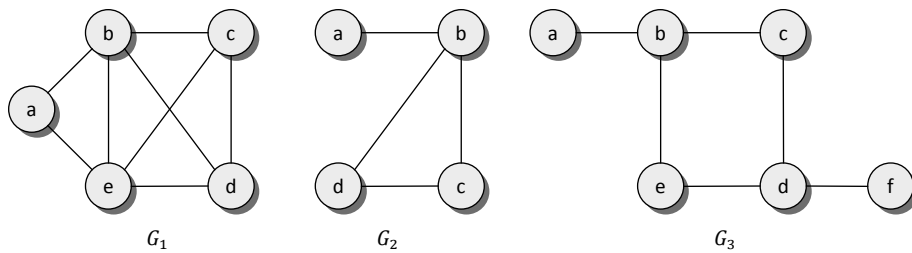
Khái niệm về đường đi và chu trình Hamilton được đưa ra bởi William Rowan Hamilton (1856) khi ông thiết kế một trò chơi trên khối đa diện 20 đỉnh, 30 cạnh, 12 mặt, mỗi mặt là một ngũ giác đều và người chơi cần chọn các cạnh để thành lập một đường đi qua 5 đỉnh cho trước (Hình 5-30).

Đồ thị $G = (V, E)$ được gọi là *đồ thị Hamilton (Hamiltonian graph)* nếu tồn tại chu trình đơn đi qua tất cả các đỉnh. Chu trình đơn đi qua tất cả các đỉnh được gọi là *chu trình Hamilton (Hamiltonian Circuit/Hamiltonian Circle)*. Để thuận tiện, người ta quy ước rằng đồ thị chỉ gồm 1 đỉnh là đồ thị Hamilton, nhưng đồ thị gồm 2 đỉnh liên thông không phải là đồ thị Hamilton.



Hình 5-30

Đồ thị $G = (V, E)$ được gọi là *đồ thị nửa Hamilton (traceable graph)* nếu tồn tại đường đi đơn qua tất cả các đỉnh. Đường đi đơn đi qua tất cả các đỉnh được gọi là *đường đi Hamilton (Hamiltonian Path)*.



Hình 5-31

Trong Hình 5-31, Đồ thị G_1 có chu trình Hamilton $\langle a, b, c, d, e, a \rangle$. G_2 không có chu trình Hamilton nhưng có đường đi Hamilton $\langle a, b, c, d \rangle$. G_3 không có cả chu trình Hamilton lẫn đường đi Hamilton.

b) Các định lý liên quan

Từ định nghĩa ta suy ra được đồ thị đường của đồ thị Euler là một đồ thị Hamilton. Ngoài ra những định lý sau đây cho chúng ta vài cách nhận biết đồ thị Hamilton.

Định lý 5-20

Đồ thị vô hướng G , trong đó tồn tại k đỉnh sao cho nếu xoá đi k đỉnh này cùng với những cạnh liên thuộc của chúng thì đồ thị nhận được sẽ có nhiều hơn k thành phần liên thông thì khẳng định là G không phải đồ thị Hamilton

Định lí 5-21 (Định lí Dirak, 1952)

Xét đơn đồ thị vô hướng $G = (V, E)$ có $n \geq 3$ đỉnh. Nếu mọi đỉnh đều có bậc không nhỏ hơn $n/2$ thì G là đồ thị Hamilton.

Định lí 5-22 (Định lí Ghouila-Houiri, 1960)

Xét đơn đồ thị có hướng liên thông mạnh $G = (V, E)$ có n đỉnh. Nếu trên phiên bản vô hướng của G , mọi đỉnh đều có bậc không nhỏ hơn n thì G là đồ thị Hamilton.

Định lí 5-23 (Định lí Ore, 1960)

Xét đơn đồ thị vô hướng $G = (V, E)$ có $n \geq 3$ đỉnh. Với mọi cặp đỉnh không kề nhau có tổng bậc $\geq n$ thì G là đồ thị Hamilton.

Định lí 5-24 (Định lí Meynie, 1973)

Xét đơn đồ thị có hướng liên thông mạnh $G = (V, E)$ có n đỉnh. Nếu trên phiên bản vô hướng của G , với mọi cặp đỉnh không kề nhau có tổng bậc $\geq 2n - 1$ thì G là đồ thị Hamilton.

Định lí 5-25 (Định lí Bondy-Chvátal, 1972)

Xét đồ thị vô hướng $G = (V, E)$ có n đỉnh, với mỗi cặp đỉnh không kề nhau u, v mà $\deg(u) + \deg(v) \geq n$ ta thêm một cạnh nối u và v , cứ làm như vậy cho tới khi không thêm được cạnh nào nữa ta thu được đồ thị mới kí hiệu $cl(G)$. Khi đó G là đồ thị Hamilton nếu và chỉ nếu $cl(G)$ là đồ thị Hamilton.

Nếu đồ thị G thỏa mãn điều kiện của Định lí 5-21 hoặc Định lí 5-23 thì $cl(G)$ là đồ thị đầy đủ, khi đó $cl(G)$ chắc chắn có chu trình Hamilton. Như vậy định lí Bondy-Chvátal là mở rộng của định lí Dirak và định lí Ore.

c) Cài đặt

Mặc dù chu trình Hamilton và chu trình Euler có tính đối ngẫu, người ta vẫn chưa tìm ra phương pháp với độ phức tạp đa thức để tìm chu trình Hamilton cũng như đường đi Hamilton trong trường hợp đồ thị tổng quát. Tất cả các thuật toán tìm chu trình Hamilton hiện nay đều dựa trên mô hình duyệt, có thể kết hợp với một số mẹo cài đặt (*heuristics*).

Chúng ta sẽ lập trình tìm một chu trình Hamilton (nếu có) trên một đơn đồ thị vô hướng với khuôn dạng Input/Output như sau:

Input

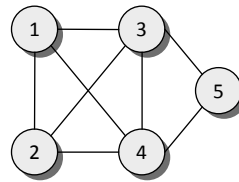
- Dòng 1 chứa số đỉnh n và số cạnh m của đơn đồ thị ($2 \leq n \leq 1000$)

- m dòng tiếp theo, mỗi dòng chứa hai số u, v tương ứng với một cạnh (u, v) của đồ thị

Output

Một chu trình Hamilton nếu có

Sample Input	Sample Output
5 8	1 2 3 5 4 1
1 2	
1 3	
1 4	
2 3	
2 4	
3 4	
3 5	
4 5	



Tìm chu trình Hamilton trên đồ thị vô hướng

```
{ $MODE OBJFPC }
program HamiltonCycle;
const
  maxN = 1000;
var
  a: array[1..maxN, 1..maxN] of Boolean; //Ma trận kề
  avail: array[2..maxN] of Boolean;
  x: array[1..maxN] of Integer;
  Found: Boolean;
  n: Integer;
procedure Enter; //Nhập dữ liệu và khởi tạo
var
  m, i, u, v: Integer;
begin
  FillChar(a, SizeOf(a), False);
  ReadLn(n, m);
  for i := 1 to m do
    begin
      Read(u, v);
      a[u, v] := True;
      a[v, u] := True;
    end;
  FillChar(avail, SizeOf(avail), True); //Mọi đỉnh 2...n đều chưa đi qua
  Found := False; //Found = False: Chưa tìm ra nghiệm
```

```

    x[1] := 1;
end;
procedure Attempt(i: Integer); //Thuật toán quay lui
var
    v: Integer;
begin
    for v := 2 to n do
        if avail[v] and a[x[i - 1], v] then //Xét các đỉnh v chưa đi qua kề
        với x[i - 1]
        begin
            x[i] := v; //Thử đi sang v
            if i = n then //Nếu đã qua đủ n đỉnh, đến đỉnh thứ n
            begin
                if a[v, 1] then Found := True; //Đỉnh thứ n quay về được
                1 thì tìm ra nghiệm
                Exit; //Thoát luôn
            end
            else //Qua chưa đủ n đỉnh
            begin
                avail[v] := False; //Đánh dấu đỉnh đã qua
                Attempt(i + 1); //Đi tiếp
                if Found then Exit; //Nếu đã tìm ra nghiệm thì thoát ngay
                avail[v] := True;
            end;
        end;
    end;
end;
procedure PrintResult; //In kết quả
var
    i: Integer;
begin
    if not Found then
        WriteLn('There is no Hamilton cycle')
    else
        begin
            for i := 1 to n do
                Write(x[i], ' ');
            WriteLn(1);
        end;
    end;
begin
    Enter;

```

```
Attempt(2);  
PrintResult;  
end.
```

6.3. Hai bài toán nổi tiếng ★

a) Bài toán người đưa thư Trung Hoa

Bài toán người đưa thư Trung Hoa (Chinese Postman) được phát biểu đầu tiên dưới dạng tìm hành trình tối ưu cho người đưa thư: Anh ta phải đi qua tất cả các quãng đường để chuyển phát thư tín và mong muốn tìm hành trình ngắn nhất để đi hết các quãng đường trong khu vực mà anh ta phụ trách. Chúng ta có thể phát biểu trên mô hình đồ thị như sau:

Bài toán: Cho đồ thị $G = (V, E)$, mỗi cạnh $e \in E$ có độ dài (trọng số) $c(e)$. Hãy tìm một chu trình đi qua tất cả các cạnh, mỗi cạnh ít nhất một lần sao cho tổng độ dài các cạnh đi qua là nhỏ nhất.

Dĩ nhiên nếu G là đồ thị Euler thì lời giải chính là chu trình Euler, nhưng nếu G không phải đồ thị Euler thì sao?. Người ta đã có thuật toán với độ phức tạp đa thức để giải bài toán người đưa thư Trung Hoa nếu G là đồ thị vô hướng hoặc có hướng. Một trong những thuật toán đó là kết hợp thuật toán tìm chu trình Euler với một thuật toán tìm bộ ghép cực đại trên đồ thị. Tuy nhiên nếu G là đồ thị hỗn hợp (có cả cung có hướng và cạnh vô hướng) thì bài toán người đưa thư Trung Hoa là bài toán NP-đầy đủ, trong trường hợp này, việc chỉ ra một thuật toán đa thức cũng như việc chứng minh không tồn tại thuật toán đa thức để giải quyết hiện vẫn đang là thách thức của ngành khoa học máy tính.

Thật đáng tiếc, sơ đồ giao thông của hầu hết các thành phố trên thế giới đều ở dạng đồ thị hỗn hợp (có cả đường hai chiều và đường một chiều) và như vậy chưa thể có một thuật toán đa thức tối ưu dành cho các nhân viên bưu chính.

b) Bài toán người du lịch

Bài toán người du lịch (Travelling Salesman) đặt ra là có n thành phố và chi phí di chuyển giữa hai thành phố bất kì trong n thành phố đó. Một người muốn đi du lịch qua tất cả các thành phố, mỗi thành phố ít nhất một lần và quay về thành phố xuất phát, sao cho tổng chi phí di chuyển là nhỏ nhất có thể. Chúng ta có thể phát biểu bài toán này trên mô hình đồ thị như sau:

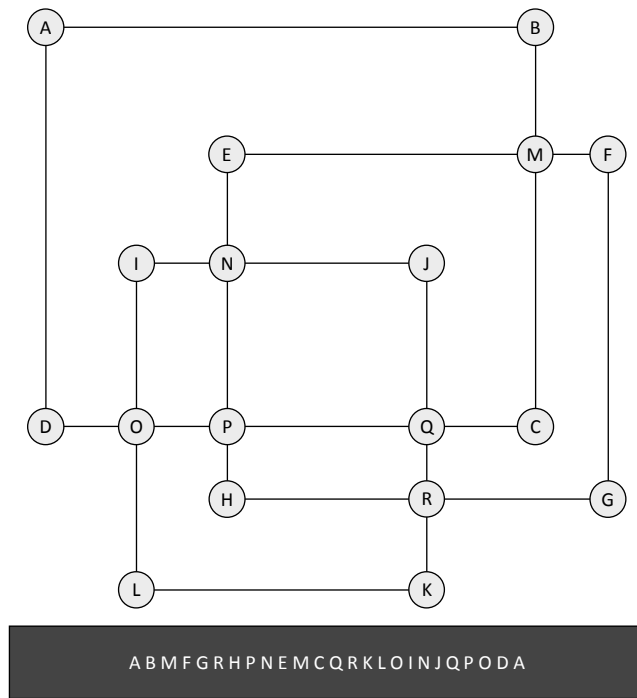
Bài toán: Cho đồ thị $G = (V, E)$, mỗi cạnh $e \in E$ có độ dài (trọng số) $c(e)$. Hãy tìm một chu trình đi qua tất cả các đỉnh, mỗi đỉnh ít nhất một lần sao cho tổng độ dài các cạnh đi qua là nhỏ nhất.

Thực ra yêu cầu đi qua mỗi đỉnh ít nhất một lần hay đi qua mỗi đỉnh đúng một lần đều khó như nhau cả. Bài toán người du lịch là NP-đầy đủ, hiện tại chưa có thuật toán đa thức để giải quyết, chỉ có một số thuật toán xấp xỉ hoặc phương pháp duyệt nhánh cận mà thôi.

Bài tập

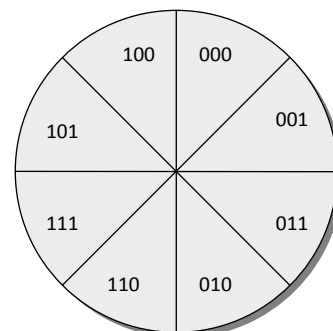
5.22. Trên mặt phẳng cho n hình chữ nhật có các cạnh song song với các trục toạ độ. Hãy chỉ ra một chu trình:

- Chỉ đi trên cạnh của các hình chữ nhật
- Trên cạnh của mỗi hình chữ nhật, ngoại trừ những giao điểm với cạnh của hình chữ nhật khác có thể qua nhiều lần, những điểm còn lại chỉ được qua đúng một lần.

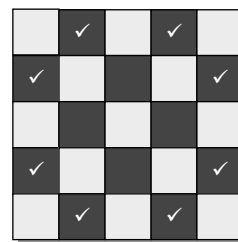


5.23. Trong đám cưới của Persée và Andromède có $2n$ hiệp sĩ. Mỗi hiệp sĩ có không quá $n - 1$ kẻ thù. Hãy giúp Cassiopée, mẹ của Andromède xếp $2n$ hiệp sĩ ngồi quanh một bàn tròn sao cho không có hiệp sĩ nào phải ngồi cạnh kẻ thù của mình. Mỗi hiệp sĩ sẽ cho biết những kẻ thù của mình khi họ đến sân rồng.

5.24. Gray code: Một hình tròn được chia thành $2n$ hình quạt đồng tâm. Hãy xếp tất cả các xâu nhị phân độ dài n vào các hình quạt, mỗi xâu vào một hình quạt sao cho bất cứ hai xâu nào ở hai hình quạt cạnh nhau đều chỉ khác nhau đúng 1 bit. Ví dụ với $n = 3$:



- 5.25.** Bài toán mã đi tuần: Trên bàn cờ tổng quát kích thước $m \times n$ ô vuông ($5 \leq m, n \leq 1000$). Một quân mã đang ở ô (x_1, y_1) có thể di chuyển sang ô (x_2, y_2) nếu $|x_1 - x_2| \cdot |y_1 - y_2| = 2$ (Xem hình vẽ).



Hãy tìm hành trình của quân mã từ ô xuất phát từ một ô tùy chọn, đi qua tất cả các ô của bàn cờ, mỗi ô đúng 1 lần.

Ví dụ với $n = 8$

Hướng dẫn: Nếu coi các ô của bàn cờ là các đỉnh của đồ thị và các cạnh là nối giữa hai đỉnh tương ứng với hai ô mã giao chân thì dễ thấy rằng hành trình của quân mã cần tìm sẽ là một đường đi Hamilton. Tuy vậy thuật toán duyệt thuận túy là bất khả thi với dữ liệu lớn, bạn có thể thử cài đặt và ngồi xem máy tính vẫn toát mồ hôi 😊.

15	26	39	58	17	28	37	50
40	59	16	27	38	51	18	29
25	14	47	52	57	30	49	36
46	41	60	31	48	53	56	19
13	24	45	62	1	20	35	54
42	61	10	23	32	55	2	5
9	12	63	44	7	4	21	34
64	43	8	11	22	33	6	3

Để giải quyết bài toán mã đi tuần, có một mẹo nhỏ được Warnsdorff đưa ra cách đây gần 2 thế kỉ (1823). Mẹo này không chỉ áp dụng được vào bài toán mã đi tuần mà còn có thể kết hợp vào thuật toán duyệt để tìm đường đi Hamilton trên đồ thị bất kì nếu biết chắc đường đi đó tồn tại (duyet tham phối hợp).

Với mỗi ô (x, y) ta gọi bậc của ô đó, $\deg(x, y)$, là số ô kề với ô (x, y) chưa được thăm (kề ở đây theo nghĩa đỉnh kề chứ không phải là ô kề cạnh). Đặt ngẫu nhiên quân mã vào ô (x, y) nào đó và cứ di chuyển quân mã sang ô kề có bậc nhỏ nhất. Nếu đi được hết bàn cờ thì xong, nếu không ta đặt ngẫu nhiên quân mã vào một ô xuất phát khác và làm lại.

Thuật toán này đã được thử nghiệm và nhận thấy rằng việc tìm ra một bộ m, n : $5 \leq m, n \leq 1000$ để chương trình chạy > 10 giây cũng là một chuyện...bất khả thi.

MỤC LỤC

CHUYÊN ĐỀ 1. THUẬT TOÁN VÀ PHÂN TÍCH THUẬT TOÁN	5
1. Thuật toán	5
2. Phân tích thuật toán.....	6
Bài tập	11
CHUYÊN ĐỀ 2. CÁC KIẾN THỨC CƠ BẢN.....	13
1. Hệ đếm	13
2. Số nguyên tố	14
3. Ước số, bội số	17
4. Lí thuyết tập hợp	18
5. Số Fibonacci	21
6. Số Catalan	23
7. Xử lí số nguyên lớn.....	24
Bài tập	33
CHUYÊN ĐỀ 3. SẮP XẾP	39
1. Phát biểu bài toán.....	39
2. Các thuật toán sắp xếp thông dụng	40
3. Sắp xếp bằng đếm phân phối (Distribution Counting)	43
Bài tập	51
CHUYÊN ĐỀ 4. THIẾT KẾ GIẢI THUẬT	59
1. Quay lui (Backtracking).....	59
2. Nhánh và cận	71
3. Tham ăn (Greedy Method).....	78
4. Chia để trị (Divide and Conquer)	88
5. Quy hoạch động (Dynamic programming)	97
Bài tập	107
CHUYÊN ĐỀ 5. CÁC THUẬT TOÁN TRÊN ĐỒ THỊ	126
1. Các khái niệm cơ bản.....	127
2. Biểu diễn đồ thị	132
3. Các thuật toán tìm kiếm trên đồ thị.....	143
4. Tính liên thông của đồ thị	158
5. Vài ứng dụng của DFS và BFS.....	182
6. Đồ thị Euler và đồ thị Hamilton.....	201
HƯỚNG DẪN GIẢI BÀI TẬP	thiếu

Chịu trách nhiệm xuất bản :

Chủ tịch HĐQT kiêm Tổng Giám đốc NGÔ TRẦN ÁI

Phó Tổng Giám đốc kiêm Tổng biên tập NGUYỄN QUÝ THAO

Tổ chức bản thảo và chịu trách nhiệm nội dung:

Phó tổng biên tập PHAN XUÂN THÀNH

Giám đốc Công ty CP. Dịch vụ Xuất bản Giáo dục Hà Nội PHAN KẾ THÁI

Biên tập và sửa bản in:

NGUYỄN THỊ THANH XUÂN

Trình bày bìa:

LƯƠNG QUỐC HIỆP

Chế bản:

NGUYỄN THỊ THANH XUÂN

Tài liệu giáo khoa chuyên Tin –Quyển 1

Mã số : 8I746H9

In bản, khổ 17 × 24 cm tại

Số in ; Số xuất bản :

In xong và nộp lưu chiểu tháng năm 2009.