

Chương III. KỸ THUẬT THIẾT KẾ THUẬT TOÁN

“It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change”

Charles Darwin

Chương này giới thiệu một số kỹ thuật quan trọng trong việc tiếp cận bài toán và tìm thuật toán. Các lớp thuật toán sẽ được thảo luận trong chương này là: Vét cạn (exhaustive search), Chia để trị (divide and conquer), Quy hoạch động (dynamic programming) và Tham lam (greedy).

Các bài toán trên thực tế có muôn hình muôn vẻ, không thể đưa ra một cách thức chung để tìm giải thuật cho mọi bài toán. Các phương pháp này cũng chỉ là những “chiến lược” kinh điển.

Khác với những thuật toán cụ thể mà chúng ta đã biết như QuickSort, tìm kiếm nhị phân,..., các vấn đề trong chương này không thể học theo kiểu “thuộc và cài đặt”, cũng như không thể tìm thấy các thuật toán này trong bất cứ thư viện lập trình nào. Chúng ta chỉ có thể khảo sát một vài bài toán cụ thể và học cách nghĩ, cách tiếp cận vấn đề, cách thiết kế giải thuật. Từ đó rèn luyện kỹ năng linh hoạt khi giải các bài toán thực tế.

Bài 1. Liệt kê

Có một số bài toán trên thực tế yêu cầu chỉ rõ: trong một tập các đối tượng cho trước có bao nhiêu đối tượng thoả mãn những điều kiện nhất định và đó là những đối tượng nào. Bài toán này gọi là *bài toán liệt kê* hay *bài toán duyệt*.

Nếu ta biểu diễn các đối tượng cần tìm dưới dạng một cấu hình các biến số thì để giải bài toán liệt kê, cần phải xác định được một *thuật toán* để có thể theo đó lần lượt xây dựng được tất cả các cấu hình đang quan tâm. Có nhiều phương pháp liệt kê, nhưng chúng cần phải đáp ứng được hai yêu cầu dưới đây:

- Không được lặp lại một cấu hình
- Không được bỏ sót một cấu hình

Trước khi nói về các thuật toán liệt kê, chúng ta giới thiệu một số khái niệm cơ bản:

1.1. Vài khái niệm cơ bản

1.1.1. Thứ tự từ điển

Nhắc lại rằng quan hệ thứ tự toàn phần “nhỏ hơn hoặc bằng” ký hiệu “ \leq ” trên một tập hợp S , là quan hệ hai ngôi thoả mãn bốn tính chất:

Với $\forall a, b, c \in S$

- Tính phổ biến (Universality): Hoặc là $a \leq b$, hoặc $b \leq a$;
- Tính phản xạ (Reflexivity): $a \leq a$
- Tính phản đối xứng (Antisymmetry): Nếu $a \leq b$ và $b \leq a$ thì bắt buộc $a = b$
- Tính bắc cầu (Transitivity): Nếu có $a \leq b$ và $b \leq c$ thì $a \leq c$.

Các quan hệ $\geq, >, <$ có thể tự suy ra từ quan hệ \leq này.

Trên các dãy hữu hạn, người ta cũng xác định một quan hệ thứ tự:

Xét $a_{1\dots n}$ và $b_{1\dots n}$ là hai dãy độ dài n , trên các phần tử của a và b đã có quan hệ thứ tự toàn phần “ \leq ”. Khi đó $a_{1\dots n} \leq b_{1\dots n}$ nếu như:

- Hoặc hai dãy giống nhau: $a_i = b_i, \forall i: 1 \leq i \leq n$
- Hoặc tồn tại một số nguyên dương $k \leq n$ để $a_k < b_k$ và $a_i = b_i, \forall i: 1 \leq i < k$

Thứ tự đó gọi là *thứ tự từ điển* (*lexicographic order*) trên các dãy độ dài n .

Khi hai dãy a và b có số phần tử khác nhau, người ta cũng xác định được thứ tự từ điển. Bằng cách thêm vào cuối dãy a hoặc dãy b những phần tử đặc biệt gọi là ϵ để độ dài của a và b bằng nhau, và coi những phần tử ϵ này nhỏ hơn tất cả các phần tử khác, ta lại đưa về xác định thứ tự từ điển của hai dãy cùng độ dài.

Ví dụ:

$$(1,2,3,4) < (5,6)$$

$$(a, b, c) < (a, b, c, d)$$

"calculator" < "computer"

Thứ tự từ điển cũng là một quan hệ thứ tự toàn phần trên các dãy.

1.1.2. Chỉnh hợp, tổ hợp, hoán vị.

Cho S là một tập hữu hạn gồm n phần tử và k là một số tự nhiên. Gọi X là tập các số nguyên dương từ 1 tới k : $X = \{1, 2, \dots, k\}$

□ Chỉnh hợp lặp

Một ánh xạ $f: X \rightarrow S$ cho tương ứng mỗi phần tử $i \in X$ một và chỉ một phần tử $f(i) \in S$, được gọi là một chỉnh hợp lặp chập k của S

Do X là tập hữu hạn (k phần tử) nên ánh xạ f có thể xác định qua bảng các giá trị $(f(1), f(2), \dots, f(k))$, vì vậy ta có thể đồng nhất f với dãy giá trị $(f(1), f(2), \dots, f(k))$ và coi dãy giá trị này cũng là một chỉnh hợp lặp chập k của S .

Ví dụ $S = \{A, B, C, D, E, F\}$. Một ánh xạ f cho bởi:

i	1	2	3
$f(i)$	E	C	E

tương ứng với tập ảnh (E, C, E) là một chỉnh hợp lặp của S

Số chỉnh hợp lặp chập k của tập n phần tử là n^k

□ Chỉnh hợp không lặp

Mỗi đơn ánh $f: X \rightarrow S$ được gọi là một chỉnh hợp không lặp chập k của S . Nói cách khác, một chỉnh hợp không lặp là một chỉnh hợp lặp có các phần tử khác nhau đôi một.

Ví dụ một chỉnh hợp không lặp chập 3 (A, C, E) của tập $S = \{A, B, C, D, E, F\}$

i	1	2	3
$f(i)$	A	C	E

Số chỉnh hợp không lặp chập k của tập n phần tử là $nP_k = \frac{n!}{(n-k)!}$

□ Hoán vị

Khi $k = n$ mỗi song ánh $f: X \rightarrow S$ được gọi là một hoán vị của S . Nói cách khác một hoán vị của S là một chỉnh hợp không lặp chập n của S .

Ví dụ: (A, D, C, E, B, F) là một hoán vị của $S = \{A, B, C, D, E, F\}$

i	1	2	3	4	5	6
$f(i)$	A	D	C	E	B	F

Số hoán vị của tập n phần tử là $P_n = {}_n P_n = n!$

□ Tổ hợp

Mỗi tập con gồm k phần tử của S được gọi là một tổ hợp chập k của S .

Lấy một tổ hợp chập k của S , xét tất cả $k!$ hoán vị của nó, mỗi hoán vị sẽ là một chỉnh hợp không lặp chập k của S . Điều đó tức là khi liệt kê tất cả các chỉnh hợp không lặp chập k thì mỗi tổ hợp chập k sẽ được tính $k!$ lần. Như vậy nếu xét về mặt số lượng:

$$\text{Số tổ hợp chập } k \text{ của tập } n \text{ phần tử là } \binom{n}{k} = \frac{{}_n P_k}{k!} = \frac{n!}{k!(n-k)!}$$

Ta có công thức khai triển nhị thức:

$$(x + a)^n = \sum_{k=0}^n \binom{n}{k} x^k a^{n-k}$$

Vì vậy số $\binom{n}{k}$ còn được gọi là *hệ số nhị thức (binomial coefficient)* thứ k , bậc n

1.2. Phương pháp sinh

Phương pháp sinh có thể áp dụng để giải bài toán liệt kê nếu như hai điều kiện sau thoả mãn:

- Có thể xác định được một thứ tự trên tập các cấu hình tổ hợp cần liệt kê. Từ đó có thể biết được cấu hình đầu tiên và cấu hình cuối cùng theo thứ tự đó.
- Xây dựng được thuật toán từ một cấu hình chưa phải cấu hình cuối, sinh ra được cấu hình kế tiếp nó.

1.2.1. Mô hình sinh

Phương pháp sinh có thể viết bằng mô hình chung:

```
«Xây dựng cấu hình đầu tiên»;
repeat
  «Đưa ra cấu hình đang có»;
  «Từ cấu hình đang có sinh ra cấu hình kế tiếp nếu còn»;
until «hết cấu hình»;
```

1.2.2. Liệt kê các dãy nhị phân độ dài n

Một dãy nhị phân độ dài n là một dãy $x_{1\dots n}$ trong đó $x_i \in \{0,1\}, \forall i: 1 \leq i \leq n$.

Có thể nhận thấy rằng một dãy nhị phân $x_{1\dots n}$ là biểu diễn nhị phân của một giá trị nguyên $v(x)$ nào đó ($0 \leq v(x) < 2^n$). Số các dãy nhị phân độ dài n bằng 2^n , thứ tự từ điển trên các dãy nhị phân độ dài n tương đương với quan hệ thứ tự trên các giá trị số mà chúng biểu diễn. Vì vậy, liệt kê các dãy nhị phân theo thứ tự từ điển nghĩa là phải chỉ ra lần lượt các dãy nhị phân biểu diễn các số nguyên theo thứ tự $0, 1, \dots, 2^n - 1$.

Ví dụ với $n = 3$, có 8 dãy nhị phân độ dài 3 được liệt kê:

x	000	001	010	011	100	101	110	111
$v(x)$	0	1	2	3	4	5	6	7

Theo thứ tự liệt kê, dãy đầu tiên là $\underbrace{00 \dots 0}_n$ và dãy cuối cùng là $\underbrace{11 \dots 1}_n$. Nếu ta có một dãy nhị phân độ dài n , ta có thể sinh ra dãy nhị phân kế tiếp bằng cách cộng thêm 1 (theo cơ số 2 có nhớ) vào dãy hiện tại.

$$\begin{array}{r} 10101111 \\ + 1 \\ \hline 10110000 \end{array}$$

Dựa vào tính chất của phép cộng hai số nhị phân, cấu hình kế tiếp có thể sinh từ cấu hình hiện tại bằng cách: xét từ cuối dãy lên đầu dãy (xét từ hàng đơn vị lên), tìm số 0 gặp đầu tiên...

- Nếu thấy thì thay số 0 đó bằng số 1 và đặt tất cả các phần tử phía sau vị trí đó bằng 0.
- Nếu không thấy thì toàn dãy là số 1, đây là cấu hình cuối cùng.

Input

Số nguyên dương n .

Output

Các dãy nhị phân độ dài n .

Sample Input	Sample Output
3	000 001 010 011 100 101 110 111

BINARYSTRINGS_GEN.PAS ✓ Thuật toán sinh liệt kê các dãy nhị phân

```
{ $MODE OBJFPC }
program BinaryStringEnumeration;
var
  x: AnsiString;
  n, i: Integer;
begin
  ReadLn(n);
  SetLength(x, n);
  FillChar(x[1], n, '0'); // Cấu hình ban đầu x=00..0
  repeat
    WriteLn(x);
    // Tìm số 0 đầu tiên từ cuối dãy
```

```

i := n;
while (i > 0) and (x[i] = '1') do Dec(i);
if i > 0 then //Nếu tìm thấy
begin
x[i] := '1'; //Thay x[i] bằng số 1
if i < n then //Đặt x[i+1..n]:=0
FillChar(x[i + 1], n - i, '0');
end
else
Break; //Không tìm thấy số 0 nào trong dãy thì dừng
until False;
end.

```

1.2.3. Liệt kê các tập con có k phần tử

Ta sẽ lập chương trình liệt kê các tập con k phần tử của tập $S = \{1, 2, \dots, n\}$ theo thứ tự từ điển.

Ví dụ: $n = 5, k = 3$, có 10 tập con:

{1, 2, 3}	{1, 2, 4}	{1, 2, 5}	{1, 3, 4}	{1, 3, 5}
{1, 4, 5}	{2, 3, 4}	{2, 3, 5}	{2, 4, 5}	{3, 4, 5}

Bài toán liệt kê các tập con k phần tử của tập $S = \{1, 2, \dots, n\}$ có thể quy về bài toán liệt kê các dãy k phần tử $x_{1\dots k}$, trong đó $1 \leq x_1 < x_2 < \dots < x_k \leq n$. Nếu sắp xếp các dãy này theo thứ tự từ điển, ta nhận thấy:

Tập con đầu tiên (cấu hình khởi tạo) là $\{1, 2, \dots, k\}$.

Tập con cuối cùng (cấu hình kết thúc) là $\{n - k + 1, n - k + 2, \dots, n\}$.

Xét một tập con $\{x_{1\dots k}\}$ trong đó $1 \leq x_1 < x_2 < \dots < x_k \leq n$, ta có nhận xét rằng giới hạn trên (giá trị lớn nhất có thể nhận) của x_k là n , của x_{k-1} là $n - 1$, của x_{k-2} là $n - 2$... Tổng quát: giới hạn trên của x_i là $n - k + i$.

Còn tất nhiên, giới hạn dưới (giá trị nhỏ nhất có thể nhận) của x_i là $x_{i-1} + 1$.

Từ một dãy $x_{1\dots k}$ đại diện cho một tập con của S , nếu tất cả các phần tử trong x đều đã đạt tới giới hạn trên thì x là cấu hình cuối cùng, nếu không thì ta phải sinh ra một dãy mới tăng dần thoả mãn: dãy mới vừa đủ lớn hơn dãy cũ theo nghĩa không có một dãy k phần tử nào chen giữa chúng khi sắp thứ tự từ điển.

Ví dụ: $n = 9, k = 6$. Cấu hình đang có $x = (1, 2, \underline{6, 7, 8, 9})$. Các phần tử $x_{3\dots 6}$ đã đạt tới giới hạn trên, nên để sinh cấu hình mới ta không thể sinh bằng cách tăng một phần tử trong số các phần tử $x_{3\dots 6}$ lên được, ta phải tăng $x_2 = 2$ lên 1 đơn vị thành $x_2 = 3$. Được cấu hình mới $x = (1, 3, 6, 7, 8, 9)$. Cấu hình này lớn hơn cấu hình trước nhưng chưa thoả mãn tính chất vừa đủ lớn. Muốn tìm cấu hình vừa đủ lớn hơn cấu hình cũ, cần có thêm thao tác: Thay các giá trị $x_{3\dots 6}$ bằng các giới hạn dưới của chúng. Tức là:

$$\begin{aligned}
x_3 &:= x_2 + 1 = 4 \\
x_4 &:= x_3 + 1 = 5 \\
x_5 &:= x_4 + 1 = 6 \\
x_6 &:= x_5 + 1 = 7
\end{aligned}$$

Ta được cấu hình mới $x = (1,3,4,5,6,7)$ là cấu hình kế tiếp. Tiếp tục với cấu hình này, ta lại nhận thấy rằng $x_6 = 7$ chưa đạt giới hạn trên, như vậy chỉ cần tăng x_6 lên 1 là được cấu hình mới $x = (1,3,4,5,6,8)$.

Thuật toán sinh dãy con kế tiếp từ dãy đang có $x_{1\dots k}$ có thể xây dựng như sau:

Tìm từ cuối dãy lên đầu cho tới khi gặp một phần tử x_i chưa đạt giới hạn trên $n - k + i \dots$

- Nếu tìm thấy:
 - Tăng x_i lên 1
 - Đặt tất cả các phần tử $x_{i+1\dots k}$ bằng giới hạn dưới của chúng
- Nếu không tìm thấy tức là mọi phần tử đã đạt giới hạn trên, đây là cấu hình cuối cùng

Input

Hai số nguyên dương $n, k, (1 \leq k \leq n \leq 100)$

Output

Các tập con k phần tử của tập $\{1, 2, \dots, n\}$

Sample Input	Sample Output
5 3	{1, 2, 3}
	{1, 2, 4}
	{1, 2, 5}
	{1, 3, 4}
	{1, 3, 5}
	{1, 4, 5}
	{2, 3, 4}
	{2, 3, 5}
	{2, 4, 5}
	{3, 4, 5}

SUBSETS_GEN.PAS ✓ Thuật toán sinh liệt kê các tập con k phần tử

```
{ $MODE OBJFPC }
program SubSetEnumeration;
const
  max = 100;
var
  x: array[1..max] of Integer;
  n, k, i, j: Integer;
begin
  ReadLn(n, k);
  for i := 1 to k do x[i] := i; //Khởi tạo x := (1, 2, ..., k)
  repeat
    //In ra cấu hình hiện tại
    Write('{');
    for i := 1 to k do
      begin
        Write(x[i]);
        if i < k then Write(', ');
      end;
    WriteLn('}');
  end;
```

```

//Duyệt từ cuối dãy lên tìm x[i] chưa đạt giới hạn trên n - k + i
i := k;
while (i > 0) and (x[i] = n - k + i) do Dec(i);
if i > 0 then //Nếu tìm thấy
begin
Inc(x[i]); //Tăng x[i] lên 1
//Đặt x[i+1..k] bằng giới hạn dưới của chúng
for j := i + 1 to k do x[j] := x[j - 1] + 1;
end
else Break;
until False;
end.

```

1.2.4. Liệt kê các hoán vị

Ta sẽ lập chương trình liệt kê các hoán vị của tập $S = \{1, 2, \dots, n\}$ theo thứ tự từ điển.

Ví dụ với $n = 3$, có 6 hoán vị:

$(1, 2, 3); (1, 3, 2); (2, 1, 3); (2, 3, 1); (3, 1, 2); (3, 2, 1)$

Mỗi hoán vị của tập $S = \{1, 2, \dots, n\}$ có thể biểu diễn dưới dạng một dãy số $x_{1\dots n}$. Theo thứ tự từ điển, ta nhận thấy:

Hoán vị đầu tiên cần liệt kê: $(1, 2, \dots, n)$

Hoán vị cuối cùng cần liệt kê: $(n, n - 1, \dots, 1)$

Bắt đầu từ hoán vị $(1, 2, \dots, n)$, ta sẽ sinh ra các hoán vị còn lại theo quy tắc: Hoán vị sẽ sinh ra phải là hoán vị vừa đủ lớn hơn hoán vị hiện tại theo nghĩa không thể có một hoán vị nào khác chen giữa chúng khi sắp thứ tự.

Giả sử hoán vị hiện tại là $x = (3, 2, 6, 5, 4, 1)$, xét 4 phần tử cuối cùng, ta thấy chúng được xếp giảm dần, điều đó có nghĩa là cho dù ta có hoán vị 4 phần tử này thế nào, ta cũng được một hoán vị bé hơn hoán vị hiện tại. Như vậy ta phải xét đến $x_2 = 2$ và thay nó bằng một giá trị khác. Ta sẽ thay bằng giá trị nào?, không thể là 1 bởi nếu vậy sẽ được hoán vị nhỏ hơn, không thể là 3 vì đã có $x_1 = 3$ rồi (phần tử sau không được chọn vào những giá trị mà phần tử trước đã chọn). Còn lại các giá trị: 4, 5 và 6. Vì cần một hoán vị vừa đủ lớn hơn hiện tại nên ta chọn $x_2 := 4$. Còn các giá trị $x_{3\dots 6}$ sẽ lấy trong tập $\{2, 6, 5, 1\}$. Cũng vì tính vừa đủ lớn nên ta sẽ tìm biểu diễn nhỏ nhất của 4 số này gán cho $x_{3\dots 6}$ tức là $x_{3\dots 6} := (1, 2, 5, 6)$. Vậy hoán vị mới sẽ là $(3, 4, 1, 2, 5, 6)$.

Ta có nhận xét gì qua ví dụ này: Đoạn cuối của hoán vị hiện tại $x_{3\dots 6}$ được xếp giảm dần, số $x_5 = 4$ là số nhỏ nhất trong đoạn cuối giảm dần thoả mãn điều kiện lớn hơn $x_2 = 2$. Nếu đảo giá trị x_5 và x_2 thì ta sẽ được hoán vị $(3, 4, 6, 5, 2, 1)$, trong đó đoạn cuối $x_{3\dots 6}$ vẫn được sắp xếp giảm dần. Khi đó muốn biểu diễn nhỏ nhất cho các giá trị trong đoạn cuối thì ta chỉ cần đảo ngược đoạn cuối.

Trong trường hợp hoán vị hiện tại là $(2, 1, 3, 4)$ thì hoán vị kế tiếp sẽ là $(2, 1, 4, 3)$. Ta cũng có thể coi hoán vị $(2, 1, 3, 4)$ có đoạn cuối giảm dần, đoạn cuối này chỉ gồm 1 phần tử (4)

Thuật toán sinh hoán vị kế tiếp từ hoán vị hiện tại có thể xây dựng như sau:

Xác định đoạn cuối giảm dần dài nhất, tìm chỉ số i của phần tử x_i đứng liền trước đoạn cuối đó. Điều này đồng nghĩa với việc tìm từ vị trí sát cuối dãy lên đầu, gặp chỉ số i đầu tiên thỏa mãn $x_i < x_{i+1}$.

- Nếu tìm thấy chỉ số i như trên
 - Trong đoạn cuối giảm dần, tìm phần tử x_k nhỏ nhất vừa đủ lớn hơn x_i . Do đoạn cuối giảm dần, điều này thực hiện bằng cách tìm từ cuối dãy lên đầu gặp chỉ số k đầu tiên thỏa mãn $x_k > x_i$ (có thể dùng tìm kiếm nhị phân).
 - Đảo giá trị x_k và x_i
 - Lật ngược thứ tự đoạn cuối giảm dần ($x_{i+1...n}$), đoạn cuối trở thành tăng dần.
- Nếu không tìm thấy tức là toàn dãy đã sắp giảm dần, đây là cấu hình cuối cùng

Input

Số nguyên dương $n \leq 100$

Output

Các hoán vị của dãy $(1, 2, \dots, n)$

Sample Input	Sample Output
3	(1, 2, 3) (1, 3, 2) (2, 1, 3) (2, 3, 1) (3, 1, 2) (3, 2, 1)

PERMUTATIONS_GEN.PAS ✓ Thuật toán sinh liệt kê hoán vị

```
{ $MODE OBJFPC }
program PermutationEnumeration;
const
  max = 100;
var
  x: array[1..max] of Integer;
  n, i, k, l, h: Integer;

// Thủ tục đảo giá trị hai tham biến x, y
procedure Swap(var x, y: Integer);
var
  temp: Integer;
begin
  temp := x; x := y; y := temp;
end;

begin
  ReadLn(n);
  for i := 1 to n do x[i] := i;
  repeat
    // In cấu hình hiện tại
    Write(' ');
    for i := 1 to n do
```

```

begin
  Write(x[i]);
  if i < n then Write(' ', ' ');
end;
WriteLn(' ');
//Sinh cấu hình kế tiếp
//Tìm i là chỉ số đứng trước đoạn cuối giảm dần
i := n - 1;
while (i > 0) and (x[i] > x[i + 1]) do Dec(i);
if i > 0 then //Nếu tìm thấy
begin
  //Tìm từ cuối dãy phần tử đầu tiên (x[k]) lớn hơn x[i]
  k := n;
  while x[k] < x[i] do Dec(k);
  //Đảo giá trị x[k] và x[i]
  Swap(x[k], x[i]);
  //Lật ngược thứ tự đoạn cuối giảm dần, đoạn cuối trở thành tăng dần
  l := i + 1; h := n;
  while l < h do
begin
  Swap(x[l], x[h]);
  Inc(l);
  Dec(h);
end;
end
else Break; //Cả dãy là giảm dần, hết cấu hình
until False;
end.

```

Nhược điểm của phương pháp sinh là không thể sinh ra được cấu hình thứ p nếu như chưa có cấu hình thứ $p - 1$, điều đó làm phương pháp sinh ít tính phổ dụng trong những thuật toán duyệt hạn chế. Hơn thế nữa, không phải cấu hình ban đầu lúc nào cũng dễ tìm được, không phải kỹ thuật sinh cấu hình kế tiếp cho mọi bài toán đều đơn giản (Sinh các chỉnh hợp không lặp chập k theo thứ tự từ điển chẳng hạn). Ta sang một chuyên mục sau nói đến một phương pháp liệt kê có tính phổ dụng cao hơn, để giải các bài toán liệt kê phức tạp hơn đó là: Thuật toán quay lui (Back tracking).

1.3. Thuật toán quay lui

Thuật toán quay lui dùng để giải bài toán liệt kê các cấu hình. Thuật toán này làm việc theo cách:

- Mỗi cấu hình được xây dựng bằng cách xây dựng từng phần tử
- Mỗi phần tử được chọn bằng cách thử tất cả các khả năng.

Giả sử cấu hình cần liệt kê có dạng $x_{1...n}$, khi đó thuật toán quay lui sẽ xét tất cả các giá trị x_1 có thể nhận, thử cho x_1 nhận lần lượt các giá trị đó. Với mỗi giá trị thử gán cho x_1 , thuật toán sẽ xét tất cả các giá trị x_2 có thể nhận, lại thử cho x_2 nhận lần lượt các giá trị đó. Với mỗi giá trị thử gán cho x_2 lại xét tiếp các khả năng chọn x_3 , cứ tiếp tục như vậy... Mỗi khi ta tìm được đầy đủ một cấu hình thì liệt kê ngay cấu hình đó.

Có thể mô tả thuật toán quay lui theo cách quy nạp: Thuật toán sẽ liệt kê các cấu hình n

phần tử dạng $x_{1\dots n}$ bằng cách thử cho x_1 nhận lần lượt các giá trị có thể. Với mỗi giá trị thử gán cho x_1 , thuật toán tiếp tục liệt kê toàn bộ các cấu hình $n - 1$ phần tử $x_{2\dots n}$...

1.3.1. Mô hình quay lui

```
//Thủ tục này thử cho x[i] nhận lần lượt các giá trị mà nó có thể nhận
procedure Attempt(i);
begin
  for «mọi giá trị v có thể gán cho x[i]» do
    begin
      «Thử cho x[i] := v»;
      if «x[i] là phần tử cuối cùng trong cấu hình» then
        «Thông báo cấu hình tìm được»
      else
        begin
          «Ghi nhận việc cho x[i] nhận giá trị V (nếu cần)»;
          Attempt(i + 1); //Gọi đệ quy để chọn tiếp x[i+1]
          «Nếu cần, bỏ ghi nhận việc thử x[i] := V để thử giá trị khác»;
        end;
    end;
end;
```

Thuật toán quay lui sẽ bắt đầu bằng lời gọi *Attempt(1)*.

Tên gọi thuật toán quay lui là dựa trên cơ chế duyệt các cấu hình: Mỗi khi thử chọn một giá trị cho x_i , thuật toán sẽ gọi đệ quy để tìm tiếp x_{i+1} , ... và cứ như vậy cho tới khi tiến trình duyệt xét tìm tới phần tử cuối cùng của cấu hình. Còn sau khi đã xét hết tất cả khả năng chọn x_i , tiến trình sẽ lùi lại thử áp đặt một giá trị khác cho x_{i-1} .

1.3.2. Liệt kê các dãy nhị phân

Biểu diễn dãy nhị phân độ dài n dưới dạng dãy $x_{1\dots n}$. Ta sẽ liệt kê các dãy này bằng cách thử dùng các giá trị $\{0,1\}$ gán cho x_i . Với mỗi giá trị thử gán cho x_i lại thử các giá trị có thể gán cho x_{i+1} ...

Sau đây là chương trình liệt kê các dãy nhị phân với quy định khuôn dạng Input/Output như trong mục 1.2.2.

BINARYSTRINGS_BT.PAS ✓ Thuật toán quay lui liệt kê các dãy nhị phân

```
{ $MODE OBJFPC }
program BinaryStringEnumeration;
var
  x: AnsiString;
  n: Integer;

procedure Attempt(i: Integer); //Thử các cách chọn x[i]
var
  j: AnsiChar;
begin
  for j := '0' to '1' do //Xét các giá trị j có thể gán cho x[i]
    begin //Với mỗi giá trị đó
      x[i] := j; //Thử đặt x[i]
```

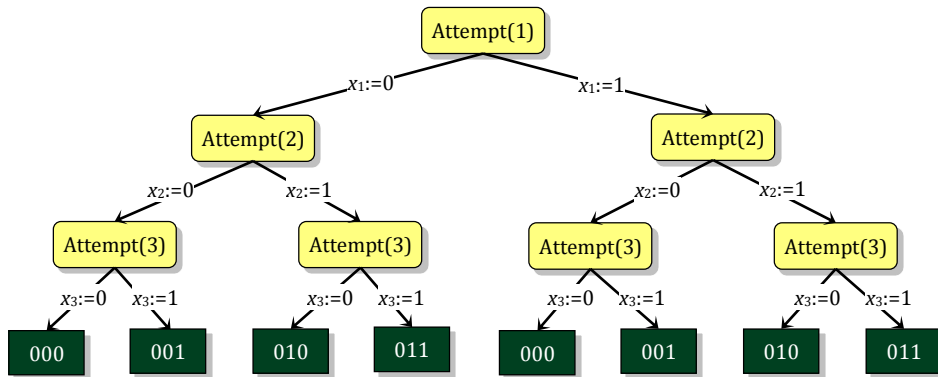
```

    if i = n then WriteLn(x) //Nếu i = n thì in kết quả
    else Attempt(i + 1); //Nếu x[i] chưa phải phần tử cuối thì tìm tiếp x[i + 1]
end;
end;

begin
  ReadLn(n);
  SetLength(x, n);
  Attempt(1); //Khởi động thuật toán quay lui
end.

```

Ví dụ: Khi $n = 3$, các lời gọi đệ quy thực hiện thuật toán quay lui có thể vẽ như cây trong Hình 1-1.



Hình 1-1. Cây tìm kiếm quay lui trong bài toán liệt kê dãy nhị phân

1.3.3. Liệt kê các tập con có k phần tử

Để liệt kê các tập con k phần tử của tập $S = \{1, 2, \dots, n\}$ ta có thể đưa về liệt kê các cấu hình $x_1 \dots x_k$, ở đây $1 \leq x_1 < x_2 < \dots < x_k \leq n$.


Theo các nhận xét ở mục 1.2.3, giá trị cận dưới và cận trên của x_i là:

$$x_{i-1} + 1 \leq x_i \leq n - k + i \quad (1.1)$$

(Giả thiết rằng có thêm một số $x_0 = 0$ khi xét công thức (1.1) với $i = 1$)

Thuật toán quay lui sẽ xét tất cả các cách chọn x_1 từ 1 ($= x_0 + 1$) đến $n - k + 1$, với mỗi giá trị đó, xét tiếp tất cả các cách chọn x_2 từ $x_1 + 1$ đến $n - k + 2$, ... cứ như vậy khi chọn được đến x_k thì ta có một cấu hình cần liệt kê.

Dưới đây là chương trình liệt kê các tập con k phần tử bằng thuật toán quay lui với khuôn dạng Input/Output như quy định trong mục 1.2.3.

 SUBSETS_BT.PAS ✓ Thuật toán quay lui liệt kê các tập con k phần tử

```

{$MODE OBJFPC}
program SubSetEnumeration;
const
  max = 100;
var

```

```

x: array[0..max] of Integer;
n, k: Integer;

procedure PrintResult; //In ra tập con {x[1..k]}
var
  i: Integer;
begin
  Write('{}');
  for i := 1 to k do
    begin
      Write(x[i]);
      if i < k then Write(', ');
    end;
  WriteLn('{}');
end;

procedure Attempt(i: Integer); //Thử các cách chọn giá trị cho x[i]
var
  j: Integer;
begin
  for j := x[i - 1] + 1 to n - k + i do
    begin
      x[i] := j;
      if i = k then PrintResult
      else Attempt(i + 1);
    end;
end;

begin
  ReadLn(n, k);
  x[0] := 0;
  Attempt(1); //Khởi động thuật toán quay lui
end.

```

Về cơ bản, các chương trình cài đặt thuật toán quay lui chỉ khác nhau ở thủ tục *Attempt*. Ví dụ ở chương trình liệt kê dãy nhị phân, thủ tục này sẽ thử chọn các giá trị 0 hoặc 1 cho x_i ; còn ở chương trình liệt kê các tập con k phần tử, thủ tục này sẽ thử chọn x_i là một trong các giá trị nguyên từ cận dưới $x_{i-1} + 1$ tới cận trên $n - k + i$. Qua đó ta có thể thấy tính phổ dụng của thuật toán quay lui: mô hình cài đặt có thể thích hợp cho nhiều bài toán. Ở phương pháp sinh tuần tự, với mỗi bài toán lại phải có một thuật toán sinh cấu hình kế tiếp, điều đó làm cho việc cài đặt mỗi bài một khác, bên cạnh đó, không phải thuật toán sinh kế tiếp nào cũng dễ tìm ra và cài đặt được.

1.3.4. Liệt kê các chỉnh hợp không lặp chập k

Để liệt kê các chỉnh hợp không lặp chập k của tập $S = \{1, 2, \dots, n\}$ ta có thể đưa về liệt kê các cấu hình $x_{1\dots k}$, các $x_i \in S$ và khác nhau đôi một.

Thủ tục *Attempt(i)* – xét tất cả các khả năng chọn x_i – sẽ thử hết các giá trị từ 1 đến n chưa bị các phần tử đứng trước $x_{1\dots i-1}$ chọn. Muốn xem các giá trị nào chưa được chọn ta sử dụng kỹ thuật dùng mảng đánh dấu:

- Khởi tạo một mảng $Free[1 \dots n]$ mang kiểu logic boolean. Ở đây $Free[j]$ cho biết giá trị j có còn tự do hay đã bị chọn rồi. Ban đầu khởi tạo tất cả các phần tử mảng $Free[1 \dots n]$ là $True$ có nghĩa là các giá trị từ 1 đến n đều tự do.
- Tại bước chọn các giá trị có thể của x_i ta chỉ xét những giá trị j còn tự do ($Free[j] := True$).
- Trước khi gọi đệ quy $Attempt(i + 1)$ để thử chọn tiếp x_{i+1} : ta đặt giá trị j vừa gán cho x_i là “đã bị chọn” ($Free[j] := False$) để các thủ tục $Attempt(i + 1), Attempt(i + 2) \dots$ gọi sau này không chọn phải giá trị j đó nữa.
- Sau khi gọi đệ quy $Attempt(i + 1)$: có nghĩa là sắp tới ta sẽ thử gán một giá trị khác cho x_i thì ta sẽ đặt giá trị j vừa thử cho x_i thành “tự do” ($Free[j] := True$), bởi khi x_i đã nhận một giá trị khác rồi thì các phần tử đứng sau ($x_{i+1} \dots x_k$) hoàn toàn có thể nhận lại giá trị j đó.
- Tất nhiên ta chỉ cần làm thao tác đánh dấu/bỏ đánh dấu trong thủ tục $Attempt(i)$ có $i \neq k$, bởi khi $i = k$ thì tiếp theo chỉ có in kết quả chứ không cần phải chọn thêm phần tử nào nữa.

Input

Hai số nguyên dương n, k ($1 \leq k \leq n \leq 100$).

Output

Các chỉnh hợp không lặp chập k của tập $\{1, 2, \dots, n\}$

Sample Input	Sample Output
3 2	(1, 2) (1, 3) (2, 1) (2, 3) (3, 1) (3, 2)



ARRANGE_BT.PAS ✓ Thuật toán quay lui liệt kê các chỉnh hợp không lặp

```
{ $MODE OBJFPC }
program ArrangementEnumeration;
const
  max = 100;
var
  x: array[1..max] of Integer;
  Free: array[1..max] of Boolean;
  n, k: Integer;

procedure PrintResult; //Thủ tục in cấu hình tìm được
var
  i: Integer;
begin
  Write(' ');
  for i := 1 to k do
```

```

begin
  Write(x[i]);
  if i < k then Write(' ');
end;
WriteLn(' ');
end;

procedure Attempt(i: Integer); //Thử các cách chọn x[i]
var
  j: Integer;
begin
  for j := 1 to n do
    if Free[j] then //Chỉ xét những giá trị còn tự do
      begin
        x[i] := j;
        if i = k then PrintResult //Nếu đã chọn được đến x[k] thì in kết quả
        else
          begin
            Free[j] := False; //Đánh dấu: j đã bị chọn
            Attempt(i + 1); //Attempt(i + 1) sẽ chỉ xét những giá trị còn tự do gán cho x[i+1]
            Free[j] := True; //Bỏ đánh dấu, sắp tới sẽ thử một cách chọn khác của x[i]
          end;
        end;
      end;
    end;
  end;

begin
  ReadLn(n, k);
  FillChar(Free[1], n, True);
  Attempt(1); //Khởi động thuật toán quay lui
end.

```

Khi $k = n$ thì đây là chương trình liệt kê hoán vị.

1.3.5. Liệt kê các cách phân tích số

Cho một số nguyên dương n , hãy tìm tất cả các cách phân tích số n thành tổng của các số nguyên dương, các cách phân tích là hoán vị của nhau chỉ tính là 1 cách và chỉ được liệt kê một lần.

Ta sẽ dùng thuật toán quay lui để liệt kê các nghiệm, mỗi nghiệm tương ứng với một dãy x , để tránh sự trùng lặp khi liệt kê các cách phân tích, ta đưa thêm ràng buộc: dãy x phải có thứ tự không giảm: $x_1 \leq x_2 \leq \dots$

Thuật toán quay lui được cài đặt bằng thủ tục đệ quy $Attempt(i)$: thử các giá trị có thể nhận của x_i , mỗi khi thử xong một giá trị cho x_i , thủ tục sẽ gọi đệ quy $Attempt(i + 1)$ để thử các giá trị có thể cho x_{i+1} . Trước mỗi bước thử các giá trị cho x_i , ta lưu trữ $m = \sum_{j=1}^{i-1} x_j$ là tổng của tất cả các phần tử đứng trước x_i : $x_{1\dots i-1}$ và thử đánh giá miền giá trị mà x_i có thể nhận.

Rõ ràng giá trị nhỏ nhất mà x_i có thể nhận chính là x_{i-1} vì dãy x có thứ tự không giảm (Giả sử rằng có thêm một phần tử $x_0 = 1$, phần tử này không tham gia vào việc liệt kê cấu hình mà chỉ dùng để hợp thức hoá giá trị cận dưới của x_1)

Nếu x_i chưa phải là phần tử cuối cùng, tức là sẽ phải chọn tiếp ít nhất một phần tử $x_{i+1} \geq x_i$ nữa mà việc chọn thêm x_{i+1} không làm cho tổng vượt quá n . Ta có:

$$\begin{aligned}
 n &\geq \sum_{j=1}^{i-1} x_j + x_i + x_{i+1} \\
 &= m + x_i + x_{i+1} \\
 &\geq m + 2x_i
 \end{aligned}
 \tag{1.2}$$

Tức là nếu x_i chưa phải phần tử cuối cùng (cần gọi đệ quy chọn tiếp x_i) thì giá trị lớn nhất x_i có thể nhận là $\lfloor \frac{n-m}{2} \rfloor$, còn dĩ nhiên nếu x_i là phần tử cuối cùng thì bắt buộc x_i phải bằng $n - m$.

Vậy thì thủ tục $Attempt(i)$ sẽ gọi đệ quy $Attempt(i + 1)$ để tìm tiếp khi mà giá trị x_i được chọn còn cho phép chọn thêm một phần tử khác lớn hơn hoặc bằng nó mà không làm tổng vượt quá n : $x_i \leq \lfloor \frac{n-m}{2} \rfloor$. Ngược lại, thủ tục này sẽ in kết quả ngay nếu x_i mang giá trị đúng bằng số thiếu hụt của tổng $i - 1$ phần tử đầu so với n . Ví dụ đơn giản khi $n = 10$ thì thử $x_1 \in \{6,7,8,9\}$ là việc làm vô nghĩa vì như vậy cũng không ra nghiệm mà cũng không chọn tiếp x_2 được nữa.

Với giá trị khởi tạo $m := 0$ và $x_0 := 1$, thuật toán quay lui sẽ được khởi động bằng lời gọi $Attempt(1)$ và hoạt động theo cách sau:

- Với mỗi giá trị j : $x_{i-1} \leq j \leq \lfloor \frac{n-m}{2} \rfloor$, thử gán $x_i := j$, cập nhật $m := m + j$, sau đó gọi đệ quy tìm tiếp, sau khi đã thử xong các giá trị có thể cho x_{i+1} , biến m được phục hồi lại như cũ $m := m - j$ trước khi thử gán một giá trị khác cho x_i .
- Cuối cùng gán $x_i := n - m$ và in kết quả ra dãy $x_{1\dots i}$.


Input

Số nguyên dương $n \leq 100$

Output

Các cách phân tích số n .

Sample Input	Sample Output
6	6 = 1+1+1+1+1+1 6 = 1+1+1+1+2 6 = 1+1+1+3 6 = 1+1+2+2 6 = 1+1+4 6 = 1+2+3 6 = 1+5 6 = 2+2+2 6 = 2+4 6 = 3+3 6 = 6

 NUMBERPARTITION_BT.PAS ✓ Liệt kê các cách phân tích số

```

{$MODE OBJFPC}
program NumberPartitioning;
const
  max = 100;
var
  x: array[0..max] of Integer;
  n, m: Integer;

procedure Init; //Khởi tạo
begin
  m := 0;
  x[0] := 1;
end;

procedure PrintResult(k: Integer); //In kết quả ra dãy x[1..k]
var
  i: Integer;
begin
  Write(n, ' = ');
  for i := 1 to k - 1 do Write(x[i], '+');
  WriteLn(x[k]);
end;

procedure Attempt(i: Integer); //Thuật toán quay lui
var
  j: Integer;
begin
  for j := x[i - 1] to (n - m) div 2 do //Trường hợp còn chọn tiếp x[i+1]
  begin
    x[i] := j; //Thử đặt x[i]
    m := m + j; //Cập nhật tổng m
    Attempt(i + 1); //Chọn tiếp
    m := m - j; //Phục hồi tổng m
  end;
  x[i] := n - m; //Nếu x[i] là phần tử cuối thì nó bắt buộc phải là n-m
  PrintResult(i); //In kết quả
end;

begin
  ReadLn(n);
  Init;

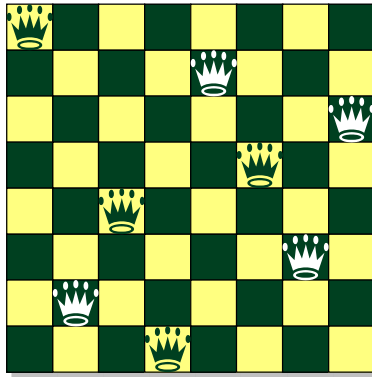
```

```
Attempt (1) ; //Khởi động thuật toán quay lui
end.
```

Bây giờ ta xét tiếp một ví dụ kinh điển của thuật toán quay lui...

1.3.6. Bài toán xếp hậu

Xét bàn cờ tổng quát kích thước $n \times n$. Một quân hậu trên bàn cờ có thể ăn được các quân khác nằm tại các ô cùng hàng, cùng cột hoặc cùng đường chéo. Hãy tìm các xếp n quân hậu trên bàn cờ sao cho không quân nào ăn quân nào. Ví dụ một cách xếp với $n = 8$ được chỉ ra trong Hình 1-2.



Hình 1-2. Một cách xếp 8 quân hậu lên bàn cờ 8×8

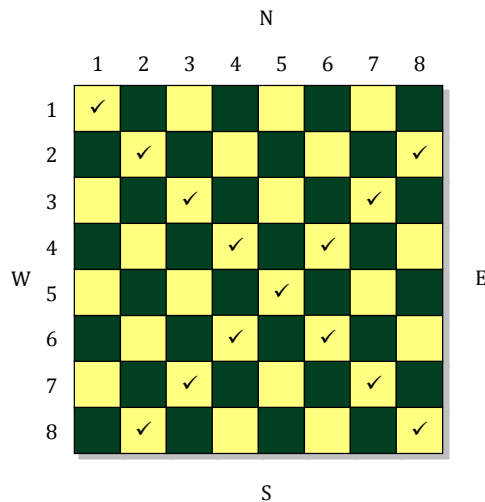
Nếu đánh số các hàng từ trên xuống dưới theo thứ tự từ 1 tới n , các cột từ trái qua phải theo thứ tự từ 1 tới n . Thì khi đặt n quân hậu lên bàn cờ, mỗi hàng phải có đúng 1 quân hậu (hậu ăn được ngang), ta gọi quân hậu sẽ đặt ở hàng 1 là quân hậu 1, quân hậu ở hàng 2 là quân hậu 2... quân hậu ở hàng n là quân hậu n . Vậy một nghiệm của bài toán sẽ được biết khi ta tìm ra được vị trí cột của những quân hậu.

Định hướng bàn cờ theo 4 hướng: Đông (Phải), Tây (Trái), Nam (Dưới), Bắc (Trên). Một quân hậu ở ô (x, y) (hàng x , cột y) sẽ khống chế.

- Toàn bộ hàng x
- Toàn bộ cột y
- Toàn bộ các ô (i, j) thỏa mãn đẳng thức $i + j = x + y$. Những ô này nằm trên một đường chéo theo hướng Đông Bắc-Tây Nam (ta gọi tắt là đường chéo phụ).
- Toàn bộ các ô (i, j) thỏa mãn đẳng thức $i - j = x - y$. Những ô này nằm trên một đường chéo Đông Nam-Tây Bắc (ta gọi tắt là đường chéo chính)

Từ những nhận xét đó, ta có ý tưởng đánh số các đường chéo trên bàn cờ.

- Với mỗi hằng số $k: 2 \leq k \leq 2n$. Tất cả các ô (i, j) trên bàn cờ thỏa mãn $i + j = k$ nằm trên một đường chéo phụ chỉ số k
- Với mỗi hằng số $l: 1 - n \leq l \leq n - 1$. Tất cả các ô (i, j) trên bàn cờ thỏa mãn $i - j = l$ nằm trên một đường chéo chính chỉ số l .



Hình 1-3. Đường chéo phụ mang chỉ số 10 và đường chéo chính mang chỉ số 0

Chúng ta sẽ sử dụng ba mảng logic để đánh dấu:

- Mảng $a_{1...n}$. $a_j = True$ nếu như cột j còn tự do, $a_j = False$ nếu như cột j đã bị một quân hậu khống chế.
- Mảng $b_{2...2n}$. $b_k = True$ nếu như đường chéo phụ thứ k còn tự do, $b_k = False$ nếu như đường chéo phụ thứ k đã bị một quân hậu khống chế.
- Mảng $c_{1-n...n-1}$. $c_l = True$ nếu như đường chéo chính thứ l còn tự do, $c_l = False$ nếu như đường chéo chính thứ l đã bị một quân hậu khống chế.

Ban đầu cả 3 mảng đánh dấu đều mang giá trị *True*. (Chưa có quân hậu nào trên bàn cờ, các cột và đường chéo đều tự do)

Thuật toán quay lui:

Xét tất cả các cột, thử đặt quân hậu 1 vào một cột, với mỗi cách đặt như vậy, xét tất cả các cách đặt quân hậu 2 không bị quân hậu 1 ăn, lại thử 1 cách đặt và xét tiếp các cách đặt quân hậu 3... Mỗi khi đặt được đến quân hậu n , ta in ra cách xếp hậu và dừng chương trình.

- Khi chọn vị trí cột j cho quân hậu thứ i , ta phải chọn ô (i, j) không bị các quân hậu đặt trước đó ăn, tức là phải chọn j thỏa mãn: cột j còn tự do: $a_j = True$, đường chéo phụ chỉ số $i + j$ còn tự do: $b_{i+j} = True$, đường chéo chính chỉ số $i - j$ còn tự do: $c_{i-j} = True$.
- Khi thử đặt được quân hậu vào ô (i, j) , nếu đó là quân hậu cuối cùng ($i = n$) thì ta có một nghiệm. Nếu không:
 - Trước khi gọi đệ quy tìm cách đặt quân hậu thứ $i + 1$, ta đánh dấu cột và 2 đường chéo bị quân hậu vừa đặt khống chế: $a_j = b_{i+j} = c_{i-j} := False$ để các lần gọi đệ quy tiếp sau chọn cách đặt các quân hậu kế tiếp sẽ không chọn vào những ô bị quân hậu vừa đặt khống chế.

- Sau khi gọi đệ quy tìm cách đặt quân hậu thứ $i + 1$, có nghĩa là sắp tới ta lại thử một cách đặt khác cho quân hậu i , ta bỏ đánh dấu cột và 2 đường chéo vừa bị quân hậu vừa thử đặt khống chế $a_j = b_{i+j} = c_{i-j} := True$ tức là cột và 2 đường chéo đó lại thành tự do, bởi khi đã đặt quân hậu i sang vị trí khác rồi thì trên cột j và 2 đường chéo đó hoàn toàn có thể đặt một quân hậu khác

Hãy xem lại trong các chương trình liệt kê chỉnh hợp không lặp và hoán vị về kỹ thuật đánh dấu. Ở đây chỉ khác với liệt kê hoán vị là: liệt kê hoán vị chỉ cần một mảng đánh dấu xem giá trị có tự do không, còn bài toán xếp hậu thì cần phải đánh dấu cả 3 thành phần: Cột, đường chéo phụ, đường chéo chính. Trường hợp đơn giản hơn: Yêu cầu liệt kê các cách đặt n quân xe lên bàn cờ $n \times n$ sao cho không quân nào ăn quân nào chính là bài toán liệt kê hoán vị.

Input

Số nguyên dương $n \leq 100$

Output

Một cách đặt các quân hậu lên bàn cờ $n \times n$

Sample Input	Sample Output
8	(1, 1) (2, 5) (3, 8) (4, 6) (5, 3) (6, 7) (7, 2) (8, 4)



NQUEENS_BT.PAS ✓ Thuật toán quay lui giải bài toán xếp hậu

```
{ $MODE OBJFPC }
program NQueens;
const
  max = 100;
var
  n: Integer;
  x: array[1..max] of Integer;
  a: array[1..max] of Boolean;
  b: array[2..2 * max] of Boolean;
  c: array[1 - max..max - 1] of Boolean;
  Found: Boolean;

procedure PrintResult; //In kết quả mỗi khi tìm ra nghiệm
var
  i: Integer;
begin
  for i := 1 to n do WriteLn('(', i, ', ', x[i], ') ');
  Found := True;
end;
```

```

//Kiểm tra ô (i, j) còn tự do hay đã bị một quân hậu khống chế?
function IsFree(i, j: Integer): Boolean;
begin
    Result := a[j] and b[i + j] and c[i - j];
end;

//Đánh dấu /bỏ đánh dấu một ô (i, j)
procedure SetFree(i, j: Integer; Enabled: Boolean);
begin
    a[j] := Enabled;
    b[i + j] := Enabled;
    c[i - j] := Enabled;
end;

procedure Attempt(i: Integer); //Thử các cách đặt quân hậu i vào hàng i
var
    j: Integer;
begin
    for j := 1 to n do //Xét tất cả các cột
        if IsFree(i, j) then //Tìm vị trí đặt chưa bị khống chế
            begin
                x[i] := j; //Thử đặt vào ô (i, j)
                if i = n then
                    begin
                        PrintResult; //Đặt đến con hậu n thì in ra 1 nghiệm
                        Exit;
                    end
                else
                    begin
                        SetFree(i, j, False); //Đánh dấu
                        Attempt(i + 1); //Thử các cách đặt quân hậu thứ i + 1
                        if Found then Exit;
                        SetFree(i, j, True); //Bỏ đánh dấu
                    end;
                end;
            end;
    end;
end;

begin
    ReadLn(n);
    //Đánh dấu tất cả các cột và đường chéo là tự do
    FillChar(a[1], n, True);
    FillChar(b[2], 2 * n - 1, True);
    FillChar(c[1 - n], 2 * n - 1, True);
    Found := False;
    Attempt(1); //Khởi động thuật toán quay lui
end.

```

Thuật toán dùng một biến *Found* làm cờ báo xem đã tìm ra nghiệm hay chưa, nếu *Found = True*, thuật toán quay lui sẽ ngưng ngay quá trình tìm kiếm.

Một sai lầm dễ mắc phải là chỉ đặt lệnh dừng thuật toán quay lui trong phép thử

```
if i = n then
  begin
    PrintResult;
    Exit;
  end;
```

Nếu làm như vậy lệnh Exit chỉ có tác dụng trong thủ tục $Attempt(n)$, muốn ngưng cả một dãy chuyền đệ quy, cần phải thoát liền một loạt các thủ tục đệ quy: $Attempt(n), Attempt(n - 1), \dots, Attempt(1)$. Đặt lệnh Exit vào sau lời gọi đệ quy chính là đặt lệnh Exit cho cả một dãy chuyền đệ quy mỗi khi tìm ra nghiệm.

Một số môi trường lập trình có lệnh dừng cả chương trình (như ở đoạn chương trình trên chúng ta có thể dùng lệnh Halt thay cho lệnh Exit). Nhưng nếu thuật toán quay lui chỉ là một phần trong chương trình, sau khi thực hiện thuật toán sẽ còn phải làm nhiều việc khác nữa, khi đó lệnh ngưng vô điều kiện cả chương trình ngay khi tìm ra nghiệm là không được phép. Cài đặt dãy Exit là một cách làm chính thống để ngưng dãy chuyền đệ quy.

1.4. Kỹ thuật nhánh cận

Có một lớp bài toán đặt ra trong thực tế yêu cầu tìm ra *một* nghiệm thoả mãn một số điều kiện nào đó, và nghiệm đó là *tốt nhất* theo một chỉ tiêu cụ thể, đó là lớp bài toán *tối ưu* (*optimization*). Nghiên cứu lời giải các lớp bài toán tối ưu thuộc về lĩnh vực quy hoạch toán học. Tuy nhiên cũng cần phải nói rằng trong nhiều trường hợp chúng ta chưa thể xây dựng một thuật toán nào thực sự hữu hiệu để giải bài toán, mà cho tới nay việc tìm nghiệm của chúng vẫn phải dựa trên mô hình *liệt kê* toàn bộ các cấu hình có thể và đánh giá, tìm ra cấu hình tốt nhất. Việc tìm phương án tối ưu theo cách này còn có tên gọi là *vét cạn* (*exhaustive search*). Chính nhờ kỹ thuật này cùng với sự phát triển của máy tính điện tử mà nhiều bài toán khó đã tìm thấy lời giải.

Việc liệt kê cấu hình có thể cài đặt bằng các phương pháp liệt kê: Sinh tuần tự và tìm kiếm quay lui. Dưới đây ta sẽ tìm hiểu kỹ hơn cơ chế của thuật toán quay lui để giới thiệu một phương pháp hạn chế không gian duyệt.

Mô hình thuật toán quay lui là tìm kiếm trên một cây phân cấp. Nếu giả thiết rằng mỗi nút nhánh của cây chỉ có 2 nút con thì cây có độ cao n sẽ có tới 2^n nút lá, con số này lớn hơn rất nhiều lần so với kích thước dữ liệu đầu vào n . Chính vì vậy mà nếu như ta có thao tác thừa trong việc chọn x_i thì sẽ phải trả giá rất lớn về chi phí thực thi thuật toán bởi quá trình tìm kiếm lòng vòng vô nghĩa trong các bước chọn kế tiếp x_{i+1}, x_{i+2}, \dots . Khi đó, một vấn đề đặt ra là trong quá trình liệt kê lời giải ta cần tận dụng những thông tin đã tìm được để *loại bỏ sớm những phương án chắc chắn không phải tối ưu*. Kỹ thuật đó gọi là kỹ thuật đánh giá nhánh cận (Branch-and-bound) trong tiến trình quay lui.

1.4.1. Mô hình kỹ thuật nhánh cận

Dựa trên mô hình thuật toán quay lui, ta xây dựng mô hình sau:

```

procedure Init;
begin
  «Khởi tạo một cấu hình bất kỳ best»;
end;

//Thủ tục này thử chọn cho x[i] tất cả các giá trị nó có thể nhận
procedure Attempt(i: Integer);
begin
  for «Mọi giá trị v có thể gán cho x[i]» do
    begin
      «Thử đặt x[i] := v»;
      if «Còn hi vọng tìm ra cấu hình tốt hơn best» then
        if «x[i] là phần tử cuối cùng trong cấu hình» then
          «Cập nhật best»
        else
          begin
            «Ghi nhận việc thử x[i] := v nếu cần»;
            Attempt(i + 1); //Gọi đệ quy, chọn tiếp x[i + 1]
            «Bỏ ghi nhận việc đã thử cho x[i] := v (nếu cần)»;
          end;
        end;
    end;
  end;
begin
  Init;
  Attempt(1);
  «Thông báo cấu hình tối ưu best»;
end.

```

Kỹ thuật nhánh cận thêm vào cho thuật toán quay lui khả năng đánh giá theo từng bước, nếu tại bước thứ i , giá trị thử gán cho x_i không có hi vọng tìm thấy cấu hình tốt hơn cấu hình *best* thì thử giá trị khác ngay mà không cần phải gọi đệ quy tìm tiếp hay ghi nhận kết quả nữa. Nghiệm của bài toán sẽ được làm tốt dần, bởi khi tìm ra một cấu hình mới tốt hơn *best*, ta sẽ cập nhật *best* bằng cấu hình mới vừa tìm được.

Dưới đây ta sẽ khảo sát một vài kỹ thuật đánh giá nhánh cận qua các bài toán cụ thể.

1.4.2. Đồ thị con đầy đủ cực đại

Bài toán tìm đồ thị con đầy đủ cực đại (Clique) là một bài toán có rất nhiều ứng dụng trong các mạng xã hội, tin sinh học, mạng truyền thông, nghiên cứu cấu trúc phân tử... Ta có thể phát biểu bài toán một cách hình thức như sau: Có n người và mỗi người có quen biết một số người khác. Giả sử quan hệ quen biết là quan hệ hai chiều, tức là nếu người i quen người j thì người j cũng quen người i và ngược lại. Vấn đề là hãy chọn ra một tập gồm nhiều người nhất trong số n người đã cho để hai người bất kỳ được chọn phải quen biết nhau.

Tuy đã có rất nhiều nghiên cứu về bài toán Clique nhưng người ta vẫn chưa tìm ra được thuật toán với độ phức tạp đa thức. Ta sẽ trình bày một cách giải bài toán Clique bằng thuật toán quay lui kết hợp với kỹ thuật nhánh cận.

□ Mô hình duyệt

Các quan hệ quen nhau được biểu diễn bởi ma trận $A = \{a_{ij}\}_{n \times n}$ trong đó $a_{ij} = \text{True}$ nếu như người i quen người j và $a_{ij} = \text{False}$ nếu như người i không quen người j . Theo giả thiết của bài toán, ma trận A là ma trận đối xứng: $a_{ij} = a_{ji} (\forall i, j)$.

Rõ ràng với một người bất kỳ thì có hai khả năng: người đó được chọn hoặc người đó không được chọn. Vì vậy một nghiệm của bài toán có thể biểu diễn bởi dãy $X = (x_1, x_2, \dots, x_n)$ trong đó $x_i = \text{True}$ nếu người thứ i được chọn và $x_i = \text{False}$ nếu người thứ i không được chọn. Gọi k là số người được chọn tương ứng với dãy X , tức là số vị trí $x_i = \text{True}$.

Phương án tối ưu được lưu trữ bởi mảng $best[1 \dots n]$, với $kbest$ là số người được chọn tương ứng với dãy $best$. Để đơn giản, ta khởi tạo mảng $best[1 \dots n]$ bởi giá trị False và $kbest := 0$, sau đó phương án $best$ và biến $kbest$ sẽ được thay bằng những phương án tốt hơn trong quá trình duyệt. Trên thực tế, phương án $best$ có thể khởi tạo bằng một thuật toán gần đúng.

Mô hình duyệt được thiết kế như mô hình liệt kê các dãy nhị phân bằng thuật toán quay lui: Thử hai giá trị True/False cho x_1 , với mỗi giá trị vừa thử cho x_1 lại thử hai giá trị của $x_2 \dots$

Gọi $deg[i]$ là số người quen của người i và $count[i]$ là số người quen của người i mà đã được chọn. Giá trị $deg[i]$ được xác định ngay từ đầu còn giá trị $count[i]$ sẽ được cập nhật ngay lập tức mỗi khi ta thử quyết định chọn hay không chọn một người j quen với i ($j < i$). Mảng $deg[1 \dots n]$ và $count[1 \dots n]$ được sử dụng trong hàm cận để hạn chế bớt không gian duyệt.

□ Hàm cận

Thuật toán quay lui được thực hiện đệ quy thông qua thủ tục $Attempt(i)$: Thử hai giá trị có thể gán cho x_i . Dây chuyền đệ quy được bắt đầu từ thủ tục $Attempt(1)$ và khi thủ tục $Attempt(i)$ được gọi thì ta đang có một phương án chọn trên tập những người từ 1 tới $i - 1$ và số người được chọn trong tập này là k .

Trong những người từ i tới n , chắc chắn nếu có chọn thêm thì ta chỉ được phép chọn những người j mà $count[j] \geq k$ và $deg[j] \leq kbest$. Điều này không khó để giải thích: $count[j] < k$ có nghĩa là người j không quen với ít nhất một người đã chọn; còn $deg[j] < kbest$ có nghĩa là nếu người j được chọn, phương án tìm được chắc chắn không thể có nhiều hơn $kbest$ người, không tốt hơn phương án $best$ hiện có.

Nhận xét trên là cơ sở để thiết lập hàm cận: Khi thủ tục $Attempt(i)$ được gọi, ta lọc ra những người trong phạm vi từ i tới n có $count[.] \geq k$ và $deg[.] > kbest$ và lập giả thuyết rằng trong trường hợp tốt nhất, tất cả những người này sẽ được chọn thêm. Giả thuyết này cho phép ta ước lượng cận trên của số người được chọn căn cứ vào dãy các quyết định $x[1 \dots i - 1]$ đã có trước đó. Nếu giá trị cận trên này vẫn $\leq kbest$, có thể kết luận ngay rằng dãy quyết định $x[1 \dots i - 1]$ không thể dẫn tới phương án tốt hơn phương án $best$ cho dù ta có thử hết những khả năng có thể của $x[i \dots n]$. Thủ tục $Attempt(i)$ sẽ không tiến hành thử gán giá trị cho x_i nữa mà thoát ngay, dây chuyền đệ quy lùi lại để thay đổi dãy quyết định $x[1 \dots i - 1]$.

Ngoài ra, thủ tục $Attempt(i)$ không nhất thiết phải thử hai giá trị True/False gán cho $x[i]$. Nếu như $count[i] < k$ hoặc $deg[i] \leq kbest$ thì chỉ cần thử $x[i] := False$ là đủ, vì trong trường hợp này nếu chọn người thứ i sẽ bị xung đột với những quyết định chọn trước hoặc không còn tiềm năng tìm ra phương án tốt hơn $best$.

❑ Cài đặt

Ta sẽ cài đặt bài toán Clique với khuôn dạng Input/Output như sau:

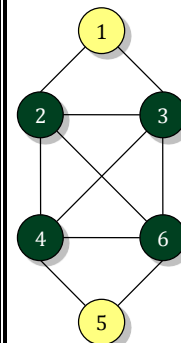
Input

- Dòng 1 chứa số người $n \leq 1000$ và số quan hệ quen biết m cách nhau ít nhất một dấu cách
- m dòng tiếp theo, mỗi dòng chứa hai số nguyên i, j cách nhau ít nhất một dấu cách cho biết về một quan hệ: hai người i, j quen nhau

Output

Phương án chọn ra nhiều người nhất để hai người bất kỳ đều quen nhau.

Sample Input	Sample Output
6 10	Number of guests: 4
1 2	Guests to be invited: 2, 3, 4, 6,
1 3	
2 3	
2 4	
2 6	
3 4	
3 6	
4 5	
4 6	
5 6	



📄 CLIQUE_BB.PAS ✓

```

{$MODE OBJFPC}
program Clique;
const
  maxN = 1000;
var
  a: array[1..maxN, 1..maxN] of Boolean;
  deg, count: array[1..maxN] of Integer;
  n: Integer;
  x, best: array[1..maxN] of Boolean;
  k, kbest: Integer;

procedure Enter; //Nhập dữ liệu và xây dựng ma trận quan hệ A
var
  m, i: Integer;
  u, v: Integer;
begin

```

```

ReadLn(n, m);
FillChar(a, SizeOf(a), False);
for i := 1 to m do
  begin
    ReadLn(u, v);
    a[u, v] := True;
    a[v, u] := True;
  end;
end;

procedure Init; //Khởi tạo
var
  u, v, temp: Integer;
  i, j: Integer;
begin
  //Trước hết tính các deg[1..n]: deg[i] = Số người quen người i
  FillDWord(deg[1], n, 0);
  for u:= 1 to n do
    for v := 1 + u to n do
      if a[u, v] then
        begin
          Inc(deg[u]); Inc(deg[v]);
        end;
  //Khởi tạo x và best là hai phương án có số người được chọn bằng 0
  FillChar(x[1], n, False); k := 0;
  FillChar(best[1], n, False); kbest := 0;
  //Đồng bộ mảng count[1..n] với phương án x
  FillDWord(count[1], n, 0);
end;

//Ước lượng cận trên của số người có thể chọn được dựa vào
function UpperBound(i: Integer): Integer;
var
  j: Integer;
begin
  Result := k;
  for j := i to n do
    if (deg[j] > kbest) and (count[j] >= k) then Inc(Result);
end;

procedure Attempt(i: Integer);
var
  j: Integer;
begin
  if UpperBound(i) <= kbest then Exit;
  if i = n + 1 then
    begin
      best := x;
      kbest := k;
      Exit;
    end;
  if (count[i] >= k) and (deg[i] >= kbest) then
    begin
      x[i] := True;
      Inc(k);
      for j := i + 1 to n do

```

```

        if a[i, j] then Inc(count[j]);
    Attempt(i + 1);
    x[i] := False;
    Dec(k);
    for j := i + 1 to n do
        if a[i, j] then Dec(count[j]);
    end;
    Attempt(i + 1);
end;

procedure PrintResult;
var
    i: Integer;
begin
    WriteLn('Number of guests: ', kbest);
    Write('Guests to be invited: ');
    for i := 1 to n do
        if best[i] then Write(i, ' ');
    WriteLn;
end;

begin
    Enter;
    Init;
    Attempt(1);
    PrintResult;
end.

```

1.4.3. Bài toán xếp ba lô

Bài toán xếp ba lô (Knapsack): Cho n sản phẩm, sản phẩm thứ i có trọng lượng là w_i và giá trị là v_i ($w_i, v_i \in \mathbb{R}^+$). Cho một balô có giới hạn trọng lượng là m , hãy chọn ra một số sản phẩm cho vào ba lô sao cho tổng trọng lượng của chúng không vượt quá m và tổng giá trị của chúng là lớn nhất có thể.

Knapsack là một bài toán nổi tiếng về độ khó: Hiện chưa có một lời giải hiệu quả cho nghiệm tối ưu trong trường hợp tổng quát. Những cố gắng để giải quyết bài toán Knapsack đã cho ra đời nhiều thuật toán gần đúng, hoặc những thuật toán tối ưu trong trường hợp đặt biệt (chẳng hạn thuật toán quy hoạch động khi $w_{1\dots n}$ và m là những số nguyên tương đối nhỏ).

Dưới đây ta sẽ xây dựng thuật toán quay lui và kỹ thuật nhánh cận để giải bài toán Knapsack.

□ Mô hình duyệt

Có hai khả năng cho mỗi sản phẩm: chọn hay không chọn. Vì vậy một cách chọn các sản phẩm xếp vào ba lô tương ứng với một dãy nhị phân độ dài n . Ta có thể biểu diễn nghiệm của bài toán dưới dạng một dãy $X = (x_1, x_2, \dots, x_n)$ trong đó $x_i = \text{True}$ nếu như sản phẩm i có được chọn. Mô hình duyệt sẽ được thiết kế tương tự như mô hình liệt kê các dãy nhị phân.

□ Lập hàm cận bằng cách “chơi sai luật”

Giả sử rằng ta có thể lấy một phần sản phẩm thay vì lấy toàn bộ sản phẩm, tức là có thể chia nhỏ một sản phẩm và định giá mỗi phần chia theo trọng lượng. Nếu sản phẩm i có giá trị v_i và trọng lượng w_i thì khi lấy một phần có trọng lượng $q \leq w_i$, phần này sẽ có giá trị là $\frac{q}{w_i} \times v_i$.

Với luật chọn được sửa đổi như vậy, ta có thể tiến hành một thuật toán tham lam để tìm phương án tối ưu: Với mỗi sản phẩm i , gọi tỉ số giá trị/khối lượng $\frac{v_i}{w_i}$ là *mật độ* của sản phẩm đó. Sắp xếp các sản phẩm theo thứ tự giảm dần của mật độ và đánh số lại các sản phẩm theo thứ tự đã sắp xếp:

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \dots \geq \frac{v_n}{w_n}$$

Bắt đầu với một ba lô rỗng, xét lần lượt các sản phẩm từ 1 tới n . Mỗi khi xét tới sản phẩm i , nếu có thể thêm toàn bộ sản phẩm i mà không vượt quá giới hạn trọng lượng của ba lô thì ta sẽ chọn toàn bộ sản phẩm i , nếu không, ta sẽ lấy một phần của sản phẩm i để đạt vừa đủ giới hạn trọng lượng của ba lô.

Ví dụ có 5 sản phẩm đã được sắp xếp giảm dần theo mật độ:

i	1	2	3	4	5
v_i	2	2	2	2	2
w_i	1	2	3	4	5

Với giới hạn trọng lượng là 8, ta sẽ lấy nguyên sản phẩm 1, nguyên sản phẩm 2, nguyên sản phẩm 3 và một nửa sản phẩm 4. Được đúng trọng lượng 8 và giá trị lấy được là $2 + 2 + 2 + 1 = 7$

Rõ ràng phương án chọn các sản phẩm như vậy là sai luật (phải lấy nguyên sản phẩm chứ không được lấy một phần), nhưng hãy thử xét lại thuật toán và giá trị lấy được, ta có nhận xét: Cho dù phương án chọn đúng luật có tốt như thế nào chăng nữa, tổng giá trị các sản phẩm được chọn không thể tốt hơn kết quả của phép chọn sai luật.

Nhận xét trên gợi ý cho ta viết một hàm $UpperBound(k, m)$: ước lượng xem nếu chọn trong các sản phẩm từ k tới n với giới hạn trọng lượng m thì tổng giá trị thu được không thể vượt quá bao nhiêu?. Giá trị hàm $UpperBound(k, m)$ được tính bằng phép chọn sai luật.

```

function UpperBound(k: Integer; m: Real): Real;
var
  i: Integer;
  q: Real;
begin
  Result := 0;
  for i := k to n do //Xét các sản phẩm từ k tới n
  begin
    if w[i] ≤ m then q := w[i] //Lấy toàn bộ sản phẩm
    else q := m; //Lấy một phần sản phẩm cho vừa đủ giới hạn trọng lượng
    Result := Result + q / w[i] * v[i]; //Cập nhật tổng giá trị
    m := m - q; //Cập nhật giới hạn trọng lượng mới
    if m = 0 then Break;
  end;
end;

```

Mỗi khi chuẩn bị ra quyết định chọn hay không chọn sản phẩm thứ k , thuật toán cần dựa vào tổng giá trị các sản phẩm đã quyết định chọn trước đó và giá trị hàm *UpperBound* để ước lượng cận trên của tổng giá trị có thể thu được. Nếu thấy không còn cơ may tìm ra phương án tốt hơn phương án đang được ghi nhận, quá trình quay lui sẽ thực hiện việc “tỉa nhánh”: không thử chọn trong các sản phẩm từ k tới n nữa bởi nếu muốn tìm ra phương án tốt hơn, cần phải thay đổi các quyết định chọn trên các sản phẩm từ 1 tới $k - 1$.

□ Đánh giá tương quan giữa các phần tử của cấu hình

Với mỗi sản phẩm, đôi khi ta không cần phải thử hai khả năng: chọn/không chọn. Một trong những cách làm là dựa vào những quyết định trên các sản phẩm trước đó để xác định sớm những sản phẩm chắc chắn không được chọn.

Giả sử trong số n sản phẩm đã cho có hai sản phẩm: sản phẩm A có trọng lượng 1 và giá trị 4, sản phẩm B có trọng lượng 2 và giá trị 3. Rõ ràng khi sắp xếp theo mật độ giảm dần thì sản phẩm A sẽ đứng trước sản phẩm B và sản phẩm A sẽ được thử trước, và khi đó nếu sản phẩm A đã không được chọn thì sau này không có lý do gì ta lại chọn sản phẩm B .

Tổng quát hơn, ta sẽ lập tức đưa quyết định không chọn sản phẩm k nếu trước đó ta đã không chọn sản phẩm i ($i < k$) có $w_i \leq w_k$ và $v_i \geq v_k$.

Nhận xét này cho ta thêm một tiêu chuẩn để hạn chế không gian duyệt. Tiêu chuẩn này có thể viết bằng hàm *Selectable*(j, k), ($j < k$): Cho biết có thể nào chọn sản phẩm k trong điều kiện ta đã quyết định chọn hay không chọn trên các sản phẩm từ 1 tới j :

```

function Selectable(j, k: Integer): Boolean;
var
  i: Integer;
begin
  for i := 1 to j do
    if not x[i] and (w[i] ≤ w[k]) and (v[i] ≥ v[k]) //Sản phẩm i không được chọn và i "không tồi hơn" k
      then Exit(False); //Kết luận q chắc chắn không được chọn
    Result := True;
  end;
end;

```

Hàm *Selectable* sẽ được dùng trong thủ tục quay lui, đồng thời tích hợp vào hàm *UpperBound* để có một đánh giá cận chặt hơn.

□ Cài đặt

Ta sẽ cài đặt bài toán xếp ba lô với khuôn dạng Input/Output như sau:

Input

- Dòng 1 chứa số nguyên dương $m \leq 100$ và số thực dương m cách nhau một dấu cách
- n dòng tiếp theo, dòng thứ i ghi hai số thực dương w_i, v_i cách nhau một dấu cách.

Output

Phương án chọn các sản phẩm có tổng trọng lượng $\leq m$ và tổng giá trị lớn nhất có thể.

Sample Input	Sample Output
5 14.0	Selected products:
<u>9.0 12.0</u>	- Product 3: Weight = 1.0; Value = 10.0
6.0 8.0	- Product 1: Weight = 9.0; Value = 12.0
<u>1.0 10.0</u>	- Product 5: Weight = 4.0; Value = 5.0
5.0 6.0	Total weight: 14.0
<u>4.0 5.0</u>	Total value : 27.0

KNAPSACK_BB.PAS ✓ Bài toán xếp balô

```

{$MODE OBJFPC}
program Knapsack;
const
  maxN = 100;
type
  TObj = record //Thông tin về một sản phẩm
    w, v: Real; //Trọng lượng và giá trị
    id: Integer; //Chỉ số
  end;
var
  obj: array[1..maxN] of TObj;
  x, best: array[1..maxN] of boolean;
  SumW, SumV: Real;
  MaxV: Real;
  n: Integer;
  m: Real;

procedure Enter; //Nhập dữ liệu

```

```

var
  i: Integer;
begin
  ReadLn(n, m);
  for i := 1 to n do
    with obj[i] do
      begin
        ReadLn(w, v);
        id := i;
      end;
    end;
end;

//Định nghĩa toán tử: sản phẩm x < sản phẩm y nếu mật độ x > mật độ y, toán tử này dùng để sắp xếp
operator < (const x, y: TObj): Boolean;
begin
  Result := x.v / x.w > y.v / y.w
end;

procedure Sort; //Thuật toán sắp xếp kiểu chèn, xếp các sản phẩm giảm dần theo mật độ
var
  i, j: Integer;
  temp: TObj;
begin
  for i := 2 to n do
    begin
      temp := obj[i]; j := i - 1;
      while (j > 0) and (temp < obj[j]) do
        begin
          obj[j + 1] := obj[j];
          Dec(j);
        end;
      obj[j + 1] := temp;
    end;
end;

procedure Init; //Khởi tạo
begin
  SumW := 0; //Một ba lô rỗng
  SumV := 0; //Tổng giá trị các phần tử được chọn
  MaxV := -1; //Tổng giá trị thu được trong phương án tối ưu best
end;

//Đánh giá xem có thể chọn sản phẩm k hay không khi đã quyết định xong với các sản phẩm 1..j
function Selectable(j, k: Integer): Boolean;
var
  i: Integer;
begin
  for i := 1 to j do
    if not x[i] and
      (obj[i].w <= obj[k].w) and (obj[i].v >= obj[k].v) then
      Exit(False); //Sản phẩm i đã không được chọn và không tồi hơn k, chắc chắn không chọn k
  Result := True;
end;

//Ước lượng giá trị cận trên của phép chọn
function UpperBound(k: Integer; m: Real): Real;

```

```

var
  i: Integer;
  q: Real;
begin
  Result := 0;
  for i := k to n do
    if Selectable(k - 1, i) then
      begin
        if obj[i].w <= m then q := obj[i].w //Lấy toàn bộ sản phẩm i
        else q := m; //Lấy một phần sản phẩm i
        Result := Result + q / obj[i].w * obj[i].v;
        m := m - q;
        if m = 0 then Break;
      end;
  end;

procedure Attempt(i: Integer); //Thuật toán quay lui
begin
  //Đánh giá xem có nên thử tiếp không, nếu không có hy vọng tìm ra nghiệm tốt hơn best thì thoát ngay
  if SumV + UpperBound(i, m - SumW) <= MaxV then
    Exit;
  if i = n + 1 then //Đã quyết định xong với n sản phẩm và tìm ra phương án x tốt hơn best
    begin //Cập nhật best, maxV và thoát ngay
      best := x;
      MaxV := SumV;
      Exit;
    end;
  if (SumW + obj[i].w <= m) and Selectable(i - 1, i) then //Sản phẩm i có thể chọn
    begin
      x[i] := True; //Thử chọn sản phẩm i
      SumW := SumW + obj[i].w; //Cập nhật tổng trọng lượng đang có trong ba lô
      SumV := SumV + obj[i].v; //Cập nhật tổng giá trị đang có trong ba lô
      Attempt(i + 1); //Thử sản phẩm kế tiếp
      SumW := SumW - obj[i].w; //Sau khi thử chọn xong, bỏ sản phẩm i khỏi ba lô
      SumV := SumV - obj[i].v; //phục hồi SumW và SumV như khi chưa chọn sản phẩm i
    end;
  x[i] := False; //Thử không chọn sản phẩm i
  Attempt(i + 1); //thử sản phẩm kế tiếp
end;

procedure PrintResult; //In kết quả
var
  i: Integer;
  TotalWeight: Real;
begin
  WriteLn('Selected products: ');
  TotalWeight := 0;
  for i := 1 to n do
    if best[i] then
      with obj[i] do
        begin
          Write('- Product ', id, ': ');
          WriteLn('Weight = ', w:1:1, ' ; Value = ', v:1:1);
          TotalWeight := TotalWeight + w;
        end;
  end;

```



```

WriteLn('Total weight: ', TotalWeight:1:1);
WriteLn('Total value : ', MaxV:1:1);
end;

begin
  Enter; //Nhập dữ liệu
  Sort; //Sắp xếp theo chiều giảm dần của mật độ
  Init; //Khởi tạo
  Attempt(1); //Khởi động thuật toán quay lui
  PrintResult; //In kết quả
end.

```

1.4.4. Dãy ABC

Cho trước một số nguyên dương $n \leq 1000$, hãy tìm một xâu chỉ gồm các ký tự 'A', 'B', 'C' thoả mãn 3 điều kiện:

- Có độ dài n .
- Hai đoạn con bất kỳ liền nhau đều khác nhau (đoạn con là một dãy ký tự liên tiếp của xâu).
- Có ít ký tự 'C' nhất.

□ Thuật toán 1: Ước lượng hàm cận

Ta sẽ dùng thuật toán quay lui để liệt kê các dãy n ký tự mà mỗi phần tử của dãy được chọn trong tập $\{A, B, C\}$. Giả sử cấu hình cần liệt kê có dạng $x_{1\dots n}$ thì:

Nếu dãy $x_{1\dots n}$ thoả mãn 2 đoạn con bất kỳ liền nhau đều khác nhau, thì trong 4 ký tự liên tiếp bất kỳ bao giờ cũng phải có ít nhất một ký tự 'C'. Như vậy với một đoạn gồm k ký tự liên tiếp của dãy $x_{1\dots n}$ thì số ký tự 'C' trong đoạn đó luôn $\geq \lfloor k/4 \rfloor$

Sau khi thử chọn $x_i \in \{A, B, C\}$, nếu ta đã có t_i ký tự 'C' trong đoạn $x_{1\dots i}$, thì cho dù các bước chọn tiếp sau làm tốt như thế nào chăng nữa, số ký tự 'C' phải chọn thêm không bao giờ ít hơn $\lfloor \frac{n-i}{4} \rfloor$. Tức là nếu theo phương án chọn x_i như thế này thì số ký tự 'C' trong dãy kết quả (khi chọn đến x_n) không thể ít hơn $t_i + \lfloor \frac{n-i}{4} \rfloor$. Ta dùng con số này để đánh giá nhánh cận, nếu nó nhiều hơn số ký tự 'C' trong cấu hình tốt nhất đang cho tới thời điểm hiện tại thì chắc chắn có làm tiếp cũng chỉ được một cấu hình tồi tệ hơn, ta bỏ qua ngay cách chọn này và thử phương án khác.

Tôi đã thử và thấy thuật toán này hoạt động khá nhanh với $n \leq 100$, tuy nhiên với những giá trị $n \geq 200$ thì vẫn không đủ kiên nhẫn để đợi ra kết quả. Dưới đây là một thuật toán khác tốt hơn, đi đôi với nó là một chiến lược chọn hàm cận khá hiệu quả, khi mà ta khó xác định hàm cận thật chặt bằng công thức tường minh.

□ Thuật toán 2: Lấy ngắn nuôi dài

Với mỗi độ dài m , ta gọi $f[m]$ là số ký tự 'C' trong xâu có độ dài m thoả mãn hai đoạn con bất kỳ liền nhau phải khác nhau và có ít ký tự 'C' nhất. Rõ ràng $f[0] = 0$, ta sẽ lập trình tính các $f[m]$ trong điều kiện các $f[0 \dots m - 1]$ đã biết.

Tương tự như thuật toán 1, giả sử cấu hình cần tìm có dạng $x_{1\dots m}$ thì sau khi thử chọn x_i , nếu ta đã có t_i ký tự 'C' trong đoạn $x_{1\dots i}$, thì cho dù các bước chọn tiếp sau làm tốt như thế nào chăng nữa, số ký tự 'C' phải chọn thêm không bao giờ ít hơn $f[m - i]$, tức là nếu chọn tiếp thì số ký tự 'C' không thể ít hơn $t_i + f[m - i]$. Ta dùng cận này kết hợp với thuật toán quay lui để tìm xâu tối ưu độ dài m cũng như để tính giá trị $f[m]$

Như vậy ta phải thực hiện thuật toán n lần với các độ dài xâu $m \in \{1, 2, \dots, n\}$, tuy nhiên lần thực hiện sau sẽ sử dụng những thông tin đã có của lần thực hiện trước để làm một hàm cận chặt hơn và thực hiện trong thời gian chấp nhận được.

□ Cài đặt

Input

Số nguyên dương n

Output

Xâu ABC cần tìm

Sample Input	Sample Output
10	Analyzing... f[1] = 0 f[2] = 0 f[3] = 0 f[4] = 1 f[5] = 1 f[6] = 1 f[7] = 1 f[8] = 2 f[9] = 2 f[10] = 2 The best string of 10 letters is: ABACBCBAB Number of 'C' letters: 2

📄 ABC_BB.PAS ✓ Dãy ABC

```
{ $MODE OBJFPC }
program ABC_STRING;
const
  max = 1000;
  modulus = 12345;
var
  n, m, MinC, CountC: Integer;
  Powers: array[0..max] of Integer;
```

```

f: array[0..max] of Integer;
x, Best: AnsiString;

procedure Init; //Với một độ dài m <= n, khởi tạo thuật toán quay lui
var
  i: Integer;
begin
  SetLength(x, m);
  MinC := m;
  CountC := 0;
  Powers[0] := 1;
  for i := 1 to m do
    Powers[i] := Powers[i - 1] * 3 mod modulus;
  f[0] := 0;
end;

//Đổi ký tự c ra một chữ số trong hệ cơ số 3
function Code(c: Char): Integer;
begin
  Result := Ord(c) - Ord('A');
end;

//Hàm Same(i, l) cho biết xâu gồm l ký tự kết thúc tại x[i] có trùng với xâu l ký tự liền trước nó không?
function Same(i, j, k: Integer): Boolean;
begin
  while k <= i do
    begin
      if x[k] <> x[j] then
        Exit(False);
      Inc(k); Inc(j);
    end;
  Result := True;
end;

//Hàm Check(i) cho biết x[i] có làm hỏng tính không lặp của dãy x[1..i] hay không. Thuật toán Rabin-Karp
function Check(i: Integer): Boolean;
var
  j, k: Integer;
  h: array[1..max] of Integer;
begin
  h[i] := Code(x[i]);
  for j := i - 1 downto 1 do
    begin
      h[j] := (Powers[i - j] * Code(x[j]) + h[j + 1]) mod modulus;
      if odd(i - j) then
        begin
          k := i - (i - j) div 2;
          if (h[k] * (Powers[k - j] + 1) mod modulus = h[j])
            and Same(i, j, k) then
            Exit(False);
        end;
    end;
  Result := True;
end;

//Giữ lại kết quả tốt hơn vừa tìm được
procedure UpdateSolution;

```

```

begin
  MinC := CountC;
  if m = n then
    Best := x;
end;

//Thuật toán quay lui
procedure Attempt(i: Integer); //Thử các giá trị có thể nhận của X[i]
var
  j: AnsiChar;
begin
  for j := 'A' to 'C' do //Xét tất cả các khả năng
    begin
      x[i] := j; //Thử đặt x[i]
      if Check(i) then //nếu giá trị đó vào không làm hỏng tính không lặp
        begin
          if j = 'C' then Inc(CountC); //Cập nhật số ký tự C cho tới bước này
          if CountC + f[m - i] < MinC then //Đánh giá nhánh cận
            if i = m then UpdateSolution //Cập nhật kết quả nếu đã đến lượt thử cuối
            else Attempt(i + 1); //Chưa đến lượt thử cuối thì thử tiếp
          if j = 'C' then Dec(CountC); //Phục hồi số ký tự C như cũ
        end;
      end;
    end;
end;

procedure PrintResult;
begin
  WriteLn('The best string of ', n, ' letters is:');
  WriteLn(Best);
  WriteLn('Number of 'C' letters: ', MinC);
end;

begin
  ReadLn(n);
  WriteLn('Analyzing...');
  for m := 1 to n do
    begin
      Init;
      Attempt(1);
      f[m] := MinC;
      WriteLn('f[' , m, '] = ', f[m]);
    end;
  WriteLn;
  PrintResult;
end.

```

1.4.5. Tóm tắt

Chúng ta đã khảo sát kỹ thuật nhánh cận áp dụng trong thuật toán quay lui để giải quyết một số bài toán tối ưu. Kỹ thuật này còn có thể áp dụng cho lớp các bài toán duyệt nói chung để hạn chế bớt không gian tìm kiếm.

Khi cài đặt thuật toán quay lui có đánh giá nhánh cận, cần có:

- Một hàm cận tốt để loại bỏ sớm những phương án chắc chắn không phải nghiệm
- Một thứ tự duyệt tốt để nhanh chóng đi tới nghiệm tối ưu

Có một số trường hợp mà khó có thể tìm ra một thứ tự duyệt tốt thì ta có thể áp dụng một thứ tự ngẫu nhiên của các giá trị cho mỗi bước thử và dùng một hàm chặn thời gian để chấp nhận ngay phương án tốt nhất đang có sau một khoảng thời gian nhất định và ngưng quá trình thử (ví dụ 1 giây). Một cách làm khác là ta sẽ chỉ duyệt tới một độ sâu nhất định, sau đó một thuật toán tham lam sẽ được áp dụng để tìm ra một nghiệm có thể không phải tối ưu nhưng tốt ở mức chấp nhận được. Chiến lược này có tên gọi “bé→duyệt, to→tham”.

Bài tập 1-1.

Hãy lập chương trình nhập vào hai số n và k , liệt kê các chỉnh hợp lặp chập k của tập $\{1, 2, \dots, n\}$.

Bài tập 1-2.

Hãy liệt kê các dãy nhị phân độ dài n mà trong đó cụm chữ số “01” xuất hiện đúng 2 lần.

Bài tập 1-3.

Nhập vào một danh sách n tên người. Liệt kê tất cả các cách chọn ra đúng k người trong số n người đó.

Bài tập 1-4.

Để liệt kê tất cả các tập con của tập $\{1, 2, \dots, n\}$ ta có thể dùng phương pháp liệt kê tập con như trên hoặc dùng phương pháp liệt kê tất cả các dãy nhị phân. Mỗi số 1 trong dãy nhị phân tương ứng với một phần tử được chọn trong tập. Ví dụ với tập $\{1, 2, 3, 4\}$ thì dãy nhị phân 1010 sẽ tương ứng với tập con $\{1, 3\}$. Hãy lập chương trình in ra tất cả các tập con của tập $\{1, 2, \dots, n\}$ theo hai phương pháp.

Bài tập 1-5.

Cần xếp n người một bàn tròn, hai cách xếp được gọi là khác nhau nếu tồn tại hai người ngồi cạnh nhau ở cách xếp này mà không ngồi cạnh nhau trong cách xếp kia. Hãy đếm và liệt kê tất cả các cách xếp.

Bài tập 1-6.

Người ta có thể dùng phương pháp sinh để liệt kê các chỉnh hợp không lặp chập k . Tuy nhiên có một cách khác là liệt kê tất cả các tập con k phần tử của tập hợp, sau đó in ra đủ $k!$ hoán vị của các phần tử trong mỗi tập hợp. Hãy viết chương trình liệt kê các chỉnh hợp không lặp chập k của tập $\{1, 2, \dots, n\}$ theo cả hai cách.

Bài tập 1-7.

Liệt kê tất cả các hoán vị chữ cái trong từ MISSISSIPPI theo thứ tự từ điển.

Bài tập 1-8.

Cho hai số nguyên dương l, n . Hãy liệt kê các xâu nhị phân độ dài n có tính chất, bất kỳ hai xâu con nào độ dài l liền nhau đều khác nhau.

Bài tập 1-9.

Với $n = 5, k = 3$ vẽ cây tìm kiếm quay lui của chương trình liệt kê tổ hợp chập k của tập $\{1, 2, \dots, n\}$

Bài tập 1-10.

Cho tập S gồm n số nguyên, hãy liệt kê tất cả các tập con k phần tử của tập S thỏa mãn: độ chênh lệch về giá trị giữa hai phần tử bất kỳ trong tập con đó không vượt quá t (t cho trước)

Bài tập 1-11.

Một dãy $x_1 \dots x_n$ gọi là một hoán vị hoàn toàn của tập $\{1, 2, \dots, n\}$ nếu nó là một hoán vị thoả mãn: $x_i \neq i, \forall i: 1 \leq i \leq n$. Hãy viết chương trình liệt kê tất cả các hoán vị hoàn toàn của tập $\{1, 2, \dots, n\}$

Bài tập 1-12.

Lập trình đếm số cách xếp n quân hậu lên bàn cờ $n \times n$ sao cho không quân nào ăn quân nào.

Bài tập 1-13.

Mã đi tuần: Cho bàn cờ tổng quát kích thước $n \times n$ và một quân Mã, hãy chỉ ra một hành trình của quân Mã xuất phát từ ô đang đứng đi qua tất cả các ô còn lại của bàn cờ, mỗi ô đúng 1 lần.

Bài tập 1-14.

Xét sơ đồ giao thông gồm n nút giao thông đánh số từ 1 tới n và m đoạn đường nối chúng, mỗi đoạn đường nối 2 nút giao thông. Hãy nhập dữ liệu về mạng lưới giao thông đó, nhập số hiệu hai nút giao thông s và t . Hãy in ra tất cả các cách đi từ s tới t mà mỗi cách đi không được qua nút giao thông nào quá một lần.

Bài tập 1-15.

Cho một số nguyên dương $n \leq 10000$, hãy tìm một hoán vị của dãy $(1, 2, \dots, 2n)$ sao cho tổng hai phần tử liên tiếp của dãy là số nguyên tố. Ví dụ với $n = 5$, ta có dãy $(1, 6, 5, 2, 3, 8, 9, 4, 7, 10)$

Bài tập 1-16.

Một dãy dấu ngoặc hợp lệ là một dãy các ký tự "(" và ")" được định nghĩa như sau:

- Dãy rỗng là một dãy dấu ngoặc hợp lệ độ sâu 0
- Nếu A là dãy dấu ngoặc hợp lệ độ sâu k thì (A) là dãy dấu ngoặc hợp lệ độ sâu $k + 1$
- Nếu A và B là hai dãy dấu ngoặc hợp lệ với độ sâu lần lượt là p và q thì AB là dãy dấu ngoặc hợp lệ độ sâu là $\max(p, q)$

Độ dài của một dãy dấu ngoặc là tổng số ký tự "(" và ")"

Ví dụ: Có 5 dãy dấu ngoặc hợp lệ độ dài 8 và độ sâu 3:

((OO))

((O)O)

((O))O

O(O))

O((O))

Bài toán đặt ra là khi cho biết trước hai số nguyên dương n, k và $1 \leq k \leq n \leq 10000$. Hãy liệt kê các dãy ngoặc hợp lệ có độ dài là $2n$ và độ sâu là k . Trong trường hợp có nhiều hơn 100 dãy thì chỉ cần đưa ra 100 dãy nhỏ nhất theo thứ tự từ điển.

Bài tập 1-17.

Có m người thợ và n công việc ($1 \leq m, n \leq 100$), mỗi thợ có khả năng làm một số công việc nào đó. Hãy chọn ra một tập ít nhất những người thợ sao cho bất kỳ công việc nào trong số công việc đã cho đều có người làm được trong số những người đã chọn.

Bài 2. Chia để trị và giải thuật đệ quy

2.1. Chia để trị

Ta nói một đối tượng là đệ quy nếu nó được định nghĩa qua một đối tượng khác cùng dạng với chính nó.

Ví dụ: Đặt hai chiếc gương cầu đối diện nhau. Trong chiếc gương thứ nhất chứa hình chiếc gương thứ hai. Chiếc gương thứ hai lại chứa hình chiếc gương thứ nhất nên tất nhiên nó chứa lại hình ảnh của chính nó trong chiếc gương thứ nhất... Ở một góc nhìn hợp lý, ta có thể thấy một dãy ảnh vô hạn của cả hai chiếc gương.

Một ví dụ khác là nếu người ta phát hình trực tiếp phát thanh viên ngồi bên máy vô tuyến truyền hình, trên màn hình của máy này lại có chính hình ảnh của phát thanh viên đó ngồi bên máy vô tuyến truyền hình và cứ như thế...

Trong toán học, ta cũng hay gặp các định nghĩa đệ quy:

- Giai thừa của n ($n!$): Nếu $n = 0$ thì $n! = 1$; nếu $n > 0$ thì $n! = n(n - 1)!$
- Ký hiệu số phần tử của một tập hợp hữu hạn S là $|S|$: Nếu $S = \emptyset$ thì $|S| = 0$; Nếu $S \neq \emptyset$ thì tất có một phần tử $x \in S$, khi đó $|S| = |S - \{x\}| + 1$. Đây là phương pháp định nghĩa tập các số tự nhiên.

Ta nói một bài toán P mang bản chất đệ quy nếu lời giải của một bài toán P có thể được thực hiện bằng lời giải của các bài toán P_1, P_2, \dots, P_n có dạng giống như P . Mới nghe thì có vẻ hơi lạ nhưng điểm mấu chốt cần lưu ý là: P_1, P_2, \dots, P_n tuy có dạng giống như P , nhưng theo một nghĩa nào đó, chúng phải “nhỏ” hơn P , để giải hơn P và việc giải chúng không cần dùng đến P .

Chia để trị (divide and conquer) là một phương pháp thiết kế giải thuật cho các bài toán mang bản chất đệ quy: Để giải một bài toán lớn, ta phân rã nó thành những bài toán con đồng dạng, và cứ tiến hành phân rã cho tới khi những bài toán con đủ nhỏ để có thể giải trực tiếp. Sau đó những nghiệm của các bài toán con này sẽ được phối hợp lại để được nghiệm của bài toán lớn hơn cho tới khi có được nghiệm bài toán ban đầu.

Khi nào một bài toán có thể tìm được thuật giải bằng phương pháp chia để trị?. Có thể tìm thấy câu trả lời qua việc giải đáp các câu hỏi sau:

- Có thể định nghĩa được bài toán dưới dạng phối hợp của những bài toán cùng loại nhưng nhỏ hơn hay không? Khái niệm “nhỏ hơn” là thế nào? (Xác định quy tắc phân rã bài toán)
- Trường hợp đặc biệt nào của bài toán có thể coi là đủ nhỏ để có thể giải trực tiếp được? (Xác định các bài toán cơ sở)

2.2. Giải thuật đệ quy

Các giải thuật đệ quy là hình ảnh trực quan nhất của phương pháp chia để trị. Trong các ngôn ngữ lập trình cấu trúc, các giải thuật đệ quy thường được cài đặt bằng các chương trình con đệ quy. Một chương trình con đệ quy gồm hai phần:

- Phần *neo* (*anchor*): Phần này được thực hiện khi mà công việc quá đơn giản, có thể giải trực tiếp chứ không cần phải nhờ đến một bài toán con nào cả.
- Phần *đệ quy* (*recursion*): Trong trường hợp bài toán chưa thể giải được bằng phần neo, ta xác định những bài toán con và gọi đệ quy giải những bài toán con đó. Khi đã có lời giải (đáp số) của những bài toán con rồi thì phối hợp chúng lại để giải bài toán đang quan tâm.

Phần đệ quy mô phỏng quá trình phân rã bài toán theo nguyên lý chia để trị. Phần neo tương ứng với những bài toán con đủ nhỏ có thể giải trực tiếp được, nó quyết định tính hữu hạn dừng của lời giải.

Sau đây là một vài ví dụ về giải thuật đệ quy.

2.2.1. Tính giai thừa

Định nghĩa giai thừa của một số tự nhiên n , ký hiệu $n!$, là tích của các số nguyên dương từ 1 tới n :

$$n! = \prod_{i=1}^n i = 1 \times 2 \times \dots \times n$$

Hoàn toàn có thể sử dụng một thuật toán lặp để tính $n!$, tuy nhiên nếu chúng ta thử nghĩ một theo một cách khác: Vì $n! = n \times (n - 1)!$ nên để tính $n!$ (bài toán lớn) ta đi tính $(n - 1)!$ (bài toán con) rồi lấy kết quả nhân với n .

Cách nghĩ này cho ta một định nghĩa quy nạp của hàm giai thừa, bài toán tính giai thừa của một số được đưa về bài toán tính giai thừa của một số khác nhỏ hơn.

```
function Factorial(n: Integer): Integer; //Nhận vào số tự nhiên n và trả về n!
begin
  if n = 0 then Result := 1 //Phần neo
  else Result := n * Factorial(n - 1); //Phần đệ quy
end;
```

Ở đây, phần neo định nghĩa kết quả hàm tại $n = 0$, còn phần đệ quy (ứng với $n > 0$) sẽ định nghĩa kết quả hàm qua giá trị của n và giai thừa của $n - 1$.

Ví dụ: Dùng hàm này để tính $3!$

$$\begin{array}{rcl}
 3! & = & 3 \times 2! \\
 & & \downarrow \\
 2! & = & 2 \times 1! \\
 & & \downarrow \\
 1! & = & 1 \times 0! \\
 & & \downarrow \\
 & & 0! = 1
 \end{array}$$

2.2.2. Đổi cơ số

Để biểu diễn một giá trị số $x \in \mathbb{N}$ trong hệ nhị phân: $x = \overline{x_d x_{d-1} \dots x_1 x_0}_{(2)}$, ta cần tìm dãy các chữ số nhị phân $x_0, x_1, \dots, x_d \in \{0,1\}$ để:

$$x = \sum_{i=0}^d x_i 2^i = x_d \times 2^d + x_{d-1} \times 2^{d-1} + \dots + x_1 \times 2 + x_0$$

Có nhiều thuật toán lập để tìm biểu diễn nhị phân của x , tuy nhiên chúng ta có thể sử dụng phương pháp chia để trị để thiết kế một giải thuật đệ quy khá ngắn gọn.

Ta có nhận xét là x_0 (chữ số hàng đơn vị) chính bằng số dư của phép chia x cho 2 ($x \bmod 2$) và nếu loại bỏ x_0 khỏi biểu diễn nhị phân của x ta sẽ được số:

$$\overline{x_d x_{d-1} \dots x_1}_{(2)} = x \operatorname{div} 2$$

Vậy để tìm biểu diễn nhị phân của x (bài toán lớn), ta có thể tìm biểu diễn nhị phân của số $x \operatorname{div} 2$ (bài toán nhỏ) rồi viết thêm giá trị $x \bmod 2$ vào sau biểu diễn đó. Ngoài ra khi $x = 0$ hay $x = 1$, biểu diễn nhị phân của x chính là x nên có thể coi đây là những bài toán đủ nhỏ có thể giải trực tiếp được. Toàn bộ thuật toán có thể viết bằng một thủ tục đệ quy *Convert*(x) như sau:

```
procedure Convert(x: Integer);
begin
  if x ≥ 2 then Convert(x div 2);
  Output ← x mod 2;
end;
```

2.2.3. Dãy số Fibonacci

Dãy số Fibonacci bắt nguồn từ bài toán cổ về việc sinh sản của các cặp thỏ. Bài toán đặt ra 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 (ab) (cặp ban đầu)

Giữa tháng thứ 2: 1 cặp (ab) (cặp ban đầu vẫn chưa đẻ)

Giữa tháng thứ 3: 2 cặp (AB)(cd) (cặp ban đầu đẻ ra thêm 1 cặp con)

Giữa tháng thứ 4: 3 cặp (AB)(cd)(ef) (cặp ban đầu tiếp tục đẻ)

Giữa tháng thứ 5: 5 cặp (AB)(CD)(ef)(gh)(ik) (cả cặp (AB) và (CD) cùng đẻ)

Bây giờ, ta xét tới việc tính số cặp thỏ ở tháng thứ n : $f(n)$. Nếu ta đang ở tháng $n - 1$ và tính số thỏ ở tháng n thì:

Số tháng tới = Số hiện có + Số được sinh ra trong tháng tới

Mặt khác, với tất cả các cặp thỏ ≥ 1 tháng tuổi thì sang tháng sau, chúng đều ≥ 2 tháng tuổi và đều sẽ sinh. Tức là:

Số được sinh ra trong tháng tới = Số tháng trước

Vậy:

Số tháng tới = Số hiện có + Số tháng trước

$$f(n) = f(n - 1) + f(n - 2)$$

Vậy có thể tính được $f(n)$ theo công thức sau:

$$f(n) = \begin{cases} 1, & \text{nếu } n \leq 2 \\ f(n - 1) + f(n - 2), & \text{nếu } n > 2 \end{cases}$$

```
function f(n: Integer): Integer; //Tính số cặp thỏ ở tháng thứ n
begin
  if n <= 2 then Result := 1 //Phần neo
  else Result := f(n - 1) + f(n - 2); //Phần đệ quy
end;
```

2.3. Hiệu lực của chia để trị và đệ quy

Chia để trị là một cách thức tiếp cận bài toán. Có rất nhiều bài toán quen thuộc có thể giải bằng đệ quy thay vì lời giải lặp nhờ cách tiếp cận này. Ngoài những bài toán kể trên, có thể đưa ra một số ví dụ khác:

- Bài toán tìm giá trị lớn nhất trong một dãy số: Để tìm giá trị lớn nhất của dãy (a_1, a_2, \dots, a_n) , nếu dãy chỉ có một phần tử thì đó chính là giá trị lớn nhất, nếu không ta tìm giá trị lớn nhất của dãy gồm $\lfloor n/2 \rfloor$ phần tử đầu và giá trị lớn nhất của dãy gồm $\lfloor n/2 \rfloor$ phần tử cuối, sau đó chọn ra giá trị lớn nhất trong hai giá trị này. Phương pháp này tỏ ra thích hợp khi cài đặt thuật toán trên các máy song song: Việc tìm giá trị lớn nhất trong hai nửa dãy sẽ được thực hiện độc lập và đồng thời trên hai bộ xử lý khác nhau, làm giảm thời gian thực hiện trên máy. Tương tự như vậy, thuật toán tìm kiếm tuần tự cũng có thể viết bằng đệ quy.
- Thuật toán tìm kiếm nhị phân cũng có thể viết bằng đệ quy: Đưa việc tìm kiếm một giá trị trên dãy đã sắp xếp về việc tìm kiếm giá trị đó trên một dãy con có độ dài bằng một nửa.
- Thuật toán QuickSort đã có mô hình chuẩn viết bằng đệ quy, nhưng không chỉ có QuickSort, một loạt các thuật toán sắp xếp khác cũng có thể cài đặt bằng đệ quy, ví dụ thuật toán Merge Sort: Để sắp xếp một dãy khóa, ta sẽ sắp xếp riêng dãy các phần tử mang chỉ số lẻ và dãy các phần tử mang chỉ số chẵn rồi trộn chúng lại. Hay thuật toán Insertion

Sort: Để sắp xếp một dãy khóa, ta lấy ra một phần tử bất kỳ, sắp xếp các phần tử còn lại và chèn phần tử đã lấy ra vào vị trí đúng của nó...

- Để liệt kê các hoán vị của dãy số $(1, 2, \dots, n)$ ta lấy lần lượt từng phần tử trong dãy ra ghép với $(n - 1)!$ hoán vị của các phần tử còn lại.

Cần phải nhấn mạnh rằng phương pháp chia để trị là một cách tiếp cận bài toán và tìm lời giải đệ quy, nhưng nguyên lý này còn đóng vai trò chủ đạo trong việc thiết kế nhiều thuật toán khác nữa.

Ngoài ra, không được lạm dụng chia để trị và đệ quy. Có những trường hợp lời giải đệ quy tỏ ra ngắn gọn và hiệu quả (ví dụ như mô hình cài đặt thuật toán QuickSort) nhưng cũng có những trường hợp giải thuật đệ quy không nhanh hơn hay đơn giản hơn giải thuật lặp (tính giai thừa); thậm chí còn có những trường hợp mà lạm dụng đệ quy sẽ làm phức tạp hóa vấn đề, kéo theo một quá trình tính toán cồng kềnh như giải thuật tính số Fibonacci ở trên (có thể chứng minh bằng quy nạp là số phép cộng trong giải thuật đệ quy để tính số Fibonacci thứ n đúng bằng $f(n) - 1$ trong khi giải thuật lặp chỉ cần không quá n phép cộng).

Vì bài toán giải bằng phương pháp chia để trị mang bản chất đệ quy nên để chứng minh tính đúng đắn và phân tích thời gian thực hiện giải thuật, người ta thường sử dụng các công cụ quy nạp toán học. Nói riêng về việc phân tích thời gian thực hiện giải thuật, còn có một công cụ rất mạnh là định lý Master có nội dung như sau.

Giả sử rằng ta có một bài toán kích thước n và một thuật toán chia để trị. Gọi $T(n)$ là thời gian thực hiện giải thuật đó. Nếu thuật toán phân rã bài toán lớn ra thành a bài toán con, mỗi bài toán con có kích thước n/b , sau đó giải độc lập a bài toán con và phối hợp nghiệm lại thì thời gian thực hiện giải thuật sẽ là

$$T(n) = aT(n/b) + f(n)$$

Ở đây ta ký hiệu $f(n)$ là thời gian phân rã bài toán lớn, phối hợp nghiệm của các bài toán con...nói chung là các chi phí thời gian khác ngoài việc giải các bài toán con.

Định lý Master nói rằng nếu $a \geq 1$ và $b > 1$, (n/b có thể là $\lfloor n/b \rfloor$ hay $\lceil n/b \rceil$ không quan trọng), khi đó:

- Nếu $f(n) = O(n^{\log_b a - \epsilon})$ với hằng số $\epsilon > 0$, thì $T(n) = \Theta(n^{\log_b a})$
- Nếu $f(n) = \Theta(n^{\log_b a})$ thì $T(n) = \Theta(n^{\log_b a} \lg n)$
- Nếu $f(n) = \Omega(n^{\log_b a + \epsilon})$ với hằng số $\epsilon > 0$ và $af(n/b) \leq cf(n)$ với hằng số $c < 1$ và với mọi giá trị n đủ lớn thì $T(n) = \Theta(f(n))$

Tuy việc chứng minh định lý này khá phức tạp nhưng chúng ta cần phải nhớ để áp dụng được nhanh. Ta sẽ xét tiếp một vài bài toán kinh điển áp dụng phương pháp chia để trị và đệ quy

2.3.1. Tính lũy thừa

Bài toán đặt ra là cho hai số nguyên dương x, n . Hãy tính giá trị x^n .

□ Thuật toán 1

Ta có thể tính trực tiếp bằng thuật toán lặp*: Viết một hàm $Power(x, n)$ để tính x^n

```
function Power(x, n: Integer): Integer;  
var  
  i: Integer;  
begin  
  Result := 1;  
  for i := 1 to n do Result := Result * x;  
end;
```

Nếu coi phép nhân (*) là phép toán tích cực thì có thể thấy rằng thuật toán tính trực tiếp này cần n phép nhân. Vậy thời gian thực hiện giải thuật là $\Theta(n)$.

□ Thuật toán 2

Xét một thuật toán chia để trị dựa vào tính chất sau của x^n

$$x^n = \begin{cases} 1, & \text{nếu } n = 0 \\ x \times x^{n-1}, & \text{nếu } n > 0 \end{cases}$$

```
function Power(x, n: Integer): Integer;  
begin  
  if n = 0 then Result := 1  
  else Result := x * Power(x, n - 1);  
end;
```

Bài toán tính x^n được đưa về bài toán tính x^{n-1} nếu $n > 0$. Xét về thời gian thực hiện giải thuật, thuật toán cần thực hiện tất cả n phép nhân (phép toán tích cực) nên thời gian thực hiện cũng là $\Theta(n)$.

Thuật toán này không có cải thiện nào về tốc độ, lại tổn bộ nhớ hơn (bộ nhớ chứa tham số truyền cho chương trình con khi gọi đệ quy).

□ Thuật toán 3

Ta xét một thuật toán chia để trị khác:

* Một số công cụ lập trình có cung cấp sẵn hàm mũ. Như trong Free Pascal, ta có thể tính $x^n = Exp(Ln(x) * n)$ hay sử dụng thư viện Math để dùng các hàm $Power(x, n)$ hoặc $IntPower(x, n)$. Tuy nhiên cách làm này có hạn chế là không thể tùy biến được nếu chúng ta thực hiện tính toán trên số lớn, khi mà kết quả x^n buộc phải biểu diễn bằng mảng hoặc xâu ký tự...

$$x^n = \begin{cases} 1, & \text{nếu } n = 0 \\ (x^{n/2})^2, & \text{nếu } n > 0 \text{ và } n \text{ chẵn} \\ (x^{\lfloor n/2 \rfloor})^2 \times x, & \text{nếu } n > 0 \text{ và } n \text{ lẻ} \end{cases}$$

```
function Power(x, n: Integer): Integer;
begin
  if n = 0 then Result := 1
  else
    begin
      Result := Power(x, n div 2);
      Result := Result * Result;
      if n mod 2 = 1 then Result := Result * x;
    end;
  end;
end;
```

Việc tính x^n được quy về việc tính $x^{\lfloor n/2 \rfloor}$ rồi đem kết quả bình phương lên (thêm 1 phép nhân), sau đó giữ nguyên kết quả hoặc nhân thêm kết quả với x tùy theo n chẵn hay n lẻ.

Nếu gọi $T(n)$ là thời gian thực hiện hàm $Power(x, n)$ thì ngoài lệnh gọi đệ quy, các lệnh khác trong hàm $Power(x, n)$ tuy có tổng thời gian thực hiện không phải là hằng số nhưng bị chặn (trên và dưới) bởi hằng số. Vậy:

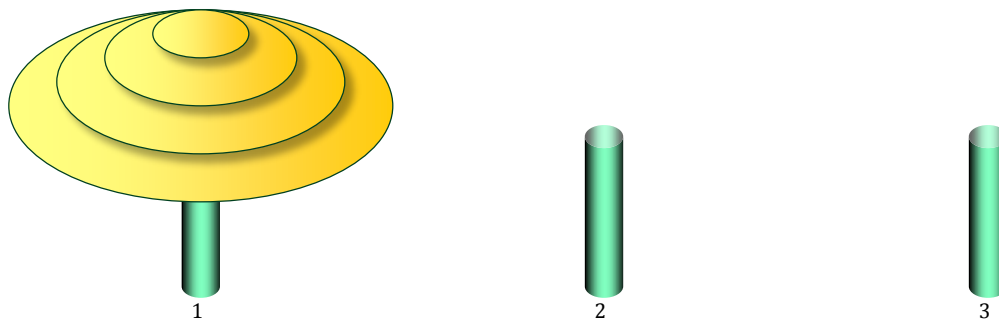
$$T(n) = T(\lfloor n/2 \rfloor) + \Theta(1)$$

Áp dụng định lý Master, ta có $T(n) = \Theta(\lg n)$. Thuật toán thứ ba tỏ ra nhanh hơn hai thuật toán trước. Ví dụ cụ thể, để tính 2^{12} , chúng ta chỉ mất 6 phép nhân so với 12 phép nhân của hai cách tính trước.

2.3.2. Tháp Hà Nội

□ Bài toán

Đây là một bài toán mang tính chất một trò chơi, tương truyền rằng tại ngôi đền Benares có ba cái cọc kim cương. Khi khai sinh ra thế giới, thượng đế đặt 64 cái đĩa bằng vàng chồng lên nhau theo thứ tự giảm dần của đường kính tính từ dưới lên, đĩa to nhất được đặt trên một chiếc cọc.



Hình 2-1. Tháp Hà Nội

Các nhà sư lần lượt chuyển các đĩa sang cọc khác theo luật:

- Khi di chuyển một đĩa, phải đặt nó vào vị trí ở một trong ba cọc đã cho
- Mỗi lần chỉ có thể chuyển một đĩa và phải là đĩa ở trên cùng của chồng đĩa
- Tại một vị trí, đĩa nào mới chuyển đến sẽ phải đặt lên trên cùng
- Đĩa lớn hơn không bao giờ được phép đặt lên trên đĩa nhỏ hơn (hay nói cách khác: một đĩa chỉ được đặt trên cọc hoặc đặt trên một đĩa lớn hơn)

Ngày tận thế sẽ đến khi toàn bộ chồng đĩa được chuyển sang một cọc khác.

Trong trường hợp có 2 đĩa, cách làm có thể mô tả như sau: Chuyển đĩa nhỏ sang cọc 3, đĩa lớn sang cọc 2 rồi chuyển đĩa nhỏ từ cọc 3 sang cọc 2.

Những người mới bắt đầu có thể giải quyết bài toán một cách dễ dàng khi số đĩa là ít, nhưng họ sẽ gặp rất nhiều khó khăn khi số các đĩa nhiều hơn. Tuy nhiên, với tư duy quy nạp toán học và một máy tính thì công việc trở nên khá dễ dàng:

□ Thuật toán chia để trị

Giả sử chúng ta có n đĩa.

Nếu $n = 1$ thì ta chuyển đĩa duy nhất đó từ cọc 1 sang cọc 2 là xong.

Nếu ta có phương pháp chuyển được $n - 1$ đĩa từ cọc 1 sang cọc 2, thì tổng quát, cách chuyển $n - 1$ đĩa từ cọc x sang cọc y ($1 \leq x, y \leq 3$) cũng tương tự.

Giả sử rằng ta có phương pháp chuyển được $n - 1$ đĩa giữa hai cọc bất kỳ. Để chuyển n đĩa từ cọc x sang cọc y , ta gọi cọc còn lại là $z (= 6 - x - y)$. Coi đĩa to nhất là ... cọc, chuyển $n - 1$ đĩa còn lại từ cọc x sang cọc z , sau đó chuyển đĩa to nhất đó sang cọc y và cuối cùng lại coi đĩa to nhất đó là cọc, chuyển $n - 1$ đĩa còn lại đang ở cọc z sang cọc y chồng lên đĩa to nhất.

Như vậy để chuyển n đĩa (bài toán lớn), ta quy về hai phép chuyển $n - 1$ đĩa (bài toán nhỏ) và một phép chuyển 1 đĩa (bài toán cơ sở). Cách làm đó được thể hiện trong thủ tục đệ quy dưới đây:

```
procedure Move(n, x, y: Integer); //Thủ tục chuyển n đĩa từ cọc x sang cọc y
begin
  if n = 1 then
    Output ← Chuyển 1 đĩa từ x sang y
  else //Để chuyển n > 1 đĩa từ cọc x sang cọc y, ta chia làm 3 công đoạn
    begin
      Move(n - 1, x, 6 - x - y); //Chuyển n - 1 đĩa từ cọc x sang cọc trung gian
      Move(1, x, y); //Chuyển đĩa to nhất từ x sang y
      Move(n - 1, 6 - x - y, y); //Chuyển n - 1 đĩa từ cọc trung gian sang cọc y
    end;
end;
```

Việc chuyển n đĩa bây giờ trở nên rất đơn giản qua một lệnh gọi $Move(n, 1, 2)$

Có thể chứng minh số phép chuyển đĩa để giải bài toán Tháp Hà Nội với n đĩa là $2^n - 1$ bằng quy nạp:

Rõ ràng là tính chất này đúng với $n = 1$, bởi ta cần $2^1 - 1 = 1$ phép chuyển để thực hiện yêu cầu.

Với $n > 1$; giả sử (quy nạp) rằng để chuyển $n - 1$ đĩa giữa hai cọc ta cần $2^{n-1} - 1$ phép chuyển, khi đó để chuyển n đĩa từ cọc x sang cọc y , nhìn vào giải thuật đệ quy ta có thể thấy rằng trong trường hợp này nó cần $(2^{n-1} - 1) + 1 + (2^{n-1} - 1) = 2^n - 1$ phép chuyển. Tính chất được chứng minh đúng với n .

Vậy thì công thức này sẽ đúng với mọi n .

2.3.3. Nhân đa thức

□ Bài toán

Cho hai đa thức $A(x) = \sum_{i=0}^m a_i x^i$ và $B(x) = \sum_{j=0}^n b_j x^j$. Bài toán đặt ra là tìm đa thức $C(x) = A(x)B(x) = \sum_{k=0}^{m+n} c_k x^k$.

Một đa thức hoàn toàn xác định nếu ta biết được giá trị các hệ số của nó. Như trong bài toán này, công việc chính là đi tìm các giá trị c_k :

$$c_k = \sum_{i+j=k} a_i b_j, \forall k: 0 \leq k \leq m+n$$

□ Phương pháp tính trực tiếp

Để tìm đa thức $C(x)$ một cách trực tiếp, có thể sử dụng thuật toán sau:

```
for k := 0 to m + n do c[k] := 0; //Khởi tạo các hệ số đa thức C bằng 0
//Xét mọi cặp hệ số a[i], b[j], cộng dồn tích a[i] * b[j] vào c[i + j]
for i := 0 to m do
  for j := 0 to n do
    c[i + j] := c[i + j] + a[i] * b[j];
```

Dễ thấy rằng thời gian thực hiện giải thuật nhân đa thức trực tiếp là $\Theta(mn)$. Dưới đây chúng ta sẽ tiếp cận bài toán theo phương pháp chia để trị và giới thiệu một thuật toán mới.

□ Thuật toán chia để trị

Đa thức $A(x)$ có bậc m và đa thức $B(x)$ có bậc n . Nếu một trong hai đa thức này có bậc 0 thì bài toán trở thành nhân một đa thức với một số, có thể coi đây là trường hợp đơn giản và có thể tính trực tiếp được. Nếu hai đa thức này đều có bậc lớn hơn 0, không giảm tính tổng quát, ta có thể coi bậc của hai đa thức này bằng nhau và là số lẻ, bởi vì việc tăng bậc của một đa thức và gán hệ số bậc cao nhất của nó bằng 0 không làm ảnh hưởng tới kết quả tính tích hai đa thức.

Bây giờ ta có hai đa thức $A(x)$ và $B(x)$ cùng bậc $n = 2k - 1$:

$$A(x) = \sum_{i=0}^{2k-1} a_i x^i; B(x) = \sum_{i=0}^{2k-1} b_i x^i$$

Xét đa thức $A(x)$:

$$\begin{aligned} A(x) &= a_{(2k-1)}x^{2k-1} + \dots + a_kx^k + a_{k-1}x^{k-1} + \dots + a_0 \\ &= x^k \underbrace{(a_{2k-1}x^{k-1} + \dots + a_k)}_{A_h(x)} + \underbrace{(a_{k-1}x^{k-1} + \dots + a_0)}_{A_l(x)} \end{aligned} \quad (2.1)$$

Vậy nếu chia dãy hệ số của $A(x)$ làm hai nửa bằng nhau, nửa những hệ số cao tương ứng với một đa thức $A_h(x)$, và nửa những hệ số thấp tương ứng với một đa thức $A_l(x)$. Thì $A_h(x)$ và $A_l(x)$ lần lượt là thương và dư của phép chia đa thức $A(x)$ cho x^k :

$$A(x) = x^k A_h(x) + A_l(x) \quad (2.2)$$

Tương tự như vậy ta có thể phân tích $B(x)$ thành:

$$B(x) = x^k B_h(x) + B_l(x) \quad (2.3)$$

Các đa thức $A_h(x)$, $A_l(x)$, $B_h(x)$ và $B_l(x)$ đều là đa thức bậc $k - 1$. Xét tích $A(x)B(x)$:

$$\begin{aligned} A(x)B(x) &= (x^k A_h(x) + A_l(x))(x^k B_h(x) + B_l(x)) \\ &= x^{2k} \underbrace{(A_h(x)B_h(x))}_{P(x)} + x^k \underbrace{(A_h(x)B_l(x) + A_l(x)B_h(x))}_{R(x)} + \underbrace{A_l(x)B_l(x)}_{Q(x)} \end{aligned} \quad (2.4)$$

Để tính $A(x)B(x)$, ta quy về việc tìm ba đa thức $P(x)$, $Q(x)$, $R(x)$. Việc tính $P(x)$ cũng như $Q(x)$, đều cần một phép nhân đa thức bậc $k - 1$. Sau khi tính được $P(x)$ và $Q(x)$ thì $R(x)$ cũng có thể tính mà chỉ dùng một phép nhân đa thức bậc $k - 1$ theo cách:

Dùng một phép nhân đa thức bậc $k - 1$ để tính:

$$S(x) = (A_h(x) + A_l(x))(B_h(x) + B_l(x)) \quad (2.5)$$

Sau đó tính

$$R(x) = S(x) - P(x) - Q(x) \quad (2.6)$$

Bạn có thể kiểm chứng đẳng thức $S(x) = P(x) + Q(x) + R(x)$ để suy ra tính đúng đắn của công thức tính.

Xét công thức (2.4), nếu gọi $T(n)$ là thời gian thực hiện phép nhân hai đa thức bậc n thì có thể nhận thấy rằng ngoài thời gian thực hiện ba phép nhân đa thức kể trên, các phép toán khác (tính tổng đa thức, nhân đa thức với đơn thức) có thời gian thực hiện $\Theta(n)$. Vậy

$$T(n) = 3T(\lfloor n/2 \rfloor) + \Theta(n) \quad (2.7)$$

Áp dụng **Error! Reference source not found.**, trường hợp 1, ta có $T(n) = \Theta(n^{\lg_2 3}) \approx \Theta(n^{1,585})$

Điều này chỉ ra rằng thuật toán chia để trị tỏ ra tốt hơn thuật toán tính trực tiếp.

❑ Cài đặt

Chúng ta sẽ cài đặt thuật toán nhân hai đa thức $A(x) = a_m x^m + a_{m-1} x^{m-1} + \dots + a_0$ và $B(x) = b_n x^n + b_{n-1} x^{n-1} \dots + b_0$ với khuôn dạng nhập xuất dữ liệu như sau:

Input

- Dòng 1 chứa hai số tự nhiên m, n lần lượt là bậc của đa thức $A(x)$ và đa thức $B(x)$
- Dòng 2 chứa $m + 1$ số thực a_m, a_{m-1}, \dots, a_0 là các hệ số của $A(x)$
- Dòng 3 chứa $n + 1$ số thực b_n, b_{n-1}, \dots, b_0 là các hệ số của $B(x)$

Output

Các hệ số của đa thức $C(x)$ từ hệ số bậc cao nhất đến hệ số bậc thấp nhất

Sample Input	Sample Output	
1 2 2.0 1.0 1.0 2.0 3.0	2.0 5.0 8.0 3.0	$A(x) = 2x + 1$ $B(x) = x^2 + 2x + 3$ $C(x) = 2x^3 + 5x^2 + 8x + 3$

📄 POLYNOMIALMULTIPLICATION_DC.PAS ✓ Nhân đa thức

```
{ $MODE OBJFPC }
program PolynomialMultiplication;
uses Math;
type
  TPolynomial = array of Real; // Đa thức được biểu diễn bằng mảng động các hệ số
var
  a, b, c: TPolynomial;
  m, n: Integer;

// Nếu hai đa thức p, q khác bậc, thêm vài hạng tử bậc cao nhất có hệ số 0 để bậc của chúng bằng nhau
procedure Equalize(var p, q: TPolynomial);
var
  lenP, lenQ, len: Integer;
begin
  lenP := Length(p);
  lenQ := Length(q);
  len := Max(lenP, lenQ);
  SetLength(p, len);
  SetLength(q, len);
  if lenP < len then // p bị tăng bậc
    FillChar(p[lenP], (len - lenP) * SizeOf(Real), 0); // Đặt p[lenP...len - 1] := 0
  if lenQ < Length(q) then // q bị tăng bậc
    FillChar(q[lenQ], (len - lenQ) * SizeOf(Real), 0); // Đặt q[lenQ...len - 1] := 0;
end;

procedure Enter; // Nhập dữ liệu
var
  i: Integer;
begin
  ReadLn(m, n);
  SetLength(a, m + 1);
  SetLength(b, n + 1);
  for i := m downto 0 do Read(a[i]);
```

```

    ReadLn;
    for i := n downto 0 do Read(b[i]);
    Equalize(a, b); //Làm bậc của hai đa thức bằng nhau
end;

//Đa thức p có bậc len - 1 được phân ra làm hai đa thức pL, pH
procedure DivMod(const p: TPolynomial; out pL, pH: TPolynomial);
var
    len, sublen: Integer;
begin
    len := Length(p); //len là số hệ số của p
    sublen := (len + 1) div 2; //Tính sublen = Ceil(len / 2)
    pL := Copy(p, 0, sublen); //Đưa phần hệ số thấp sang pL
    pH := Copy(p, sublen, len - sublen); //Đưa phần hệ số cao sang pH
    Equalize(pL, pH); //Nếu len lẻ, pH có thể có bậc nhỏ hơn pL → làm cho chúng cùng bậc
end;

//Định nghĩa toán tử gán: gán đa thức bằng một số thực v
operator := (v: Real): TPolynomial;
begin
    SetLength(Result, 1);
    Result[0] := v;
end;

//Tính tổng hai đa thức
operator +(const a, b: TPolynomial): TPolynomial;
var
    i: Integer;
begin
    SetLength(Result, Length(a));
    for i := 0 to High(Result) do
        Result[i] := a[i] + b[i];
    end;

//Tính hiệu hai đa thức
operator -(const a, b: TPolynomial): TPolynomial;
var
    i: Integer;
begin
    SetLength(Result, Length(a));
    for i := 0 to High(Result) do
        Result[i] := a[i] - b[i];
    end;

//Tính tích hai đa thức
operator *(const a, b: TPolynomial): TPolynomial;
var
    aH, aL, bH, bL: TPolynomial;
    P, Q, R: TPolynomial;
    i, k: Integer;
begin
    if High(a) = 0 then //Trường hợp cơ sở: Hai đa thức bậc 0
    begin
        Result := a[0] * b[0];
        Exit;
    end;
    //Tách mỗi đa thức thành 2 đa thức

```

```

DivMod(a, aL, aH);
DivMod(b, bL, bH);
k := Length(aH);
P := aH * bH;
Q := aL * bL;
R := (aH + aL) * (bH + bL) - (aH * bH + aL * bL);
//P, Q, R đều có bậc 2k-2, Kết quả có bậc = 4k-2
SetLength(Result, 4 * k - 1);
//Trước hết cộng P * x^(2k) vào Result ↔ Điền các hệ số của P vào Result[2k..4k-2]
for i := 0 to High(P) do
    Result[i + 2 * k] := P[i];
//Tiếp theo cộng Q vào Result ↔ Điền các hệ số của Q vào Result[0..2k-2]
for i := 0 to High(Q) do
    Result[i] := Q[i];
Result[2 * k - 1] := 0; //Còn duy nhất một hệ số của Result chưa khởi tạo, đặt = 0
//Cuối cùng cộng R * x^k vào Result
for i := 0 to High(R) do
    Result[i + k] := Result[i + k] + R[i];
end;

procedure PrintResult;
var
    i, d: Integer;
begin
    //Loại bỏ những hạng tử cao nhất bằng 0 từ đa thức kết quả
    d := High(c);
    while (d > 0) and (c[d] = 0) do Dec(d);
    SetLength(c, d + 1);
    //In kết quả
    for i := High(c) downto 0 do Write(c[i]:1:1, ' ');
    WriteLn;
end;


begin
    Enter;
    c := a * b;
    PrintResult;
end.

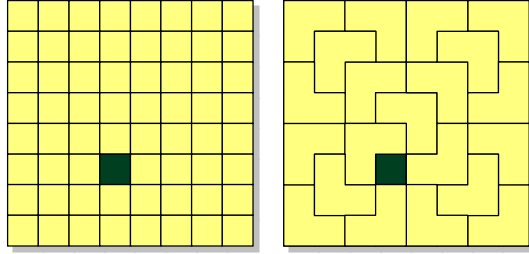
```

Bài tập 2-1.

Khi phải làm việc với các số lớn vượt quá phạm vi biểu diễn của các kiểu dữ liệu chuẩn, người ta có thể biểu diễn các số lớn bằng mảng các chữ số: $n = \overline{n_d n_{d-1} \dots n_0} = \sum_{i=0}^d n_i 10^i$ ($\forall i: 0 \leq n_i \leq 9$), và cài đặt các phép toán số học trên số lớn để thực hiện các phép toán này như với các kiểu dữ liệu chuẩn. Thuật toán nhân hai số lớn [26] có cách làm tương tự như thuật toán nhân hai đa thức. Hãy cài đặt thuật toán này để nhân hai số lớn.

Bài tập 2-2.

Cho một bảng ô vuông kích thước $2^k \times 2^k$ ô vuông đơn vị, trên đó ta bỏ đi một ô. Hãy tìm cách lát kín các ô còn lại của bảng bằng các mảnh ghép dạng  sao cho không có hai mảnh nào chồng nhau. Ví dụ với $k = 3$



Bài tập 2-3.

Xét phép nhân hai ma trận $A_{m \times n}$ và $B_{n \times p}$ để được ma trận $C_{m \times p}$:

$$c_{ik} = \sum_{j=1}^n a_{ij} \times b_{jk}, (1 \leq i \leq m, 1 \leq k \leq p)$$

Ma trận C có thể tính trực tiếp bằng thuật toán:

```

for i := 1 to m do
  for k := 1 to p do
    begin
      c[i, k] := 0;
      for j := 1 to n do c[i, k] := c[i, k] + a[i, j] * b[j, k];
    end;
  end;
end;

```

Thuật toán tính trực tiếp có thời gian thực hiện $\Theta(mnp)$.

Xét thuật toán chia để trị: Thêm những hàng 0 và cột 0 vào ma trận để ba ma trận A, B, C đều có kích thước $2^s \times 2^s$. Chia mỗi ma trận làm 4 phần bằng nhau, mỗi phần là một ma trận con kích thước $2^{s-1} \times 2^{s-1}$:

$$A = \begin{bmatrix} A^{(11)} & A^{(12)} \\ A^{(21)} & A^{(22)} \end{bmatrix}; B = \begin{bmatrix} B^{(11)} & B^{(12)} \\ B^{(21)} & B^{(22)} \end{bmatrix}; C = \begin{bmatrix} C^{(11)} & C^{(12)} \\ C^{(21)} & C^{(22)} \end{bmatrix}$$

khi đó nếu $C = A \times B$ thì:

$$\begin{aligned} C^{(11)} &= A^{(11)}B^{(11)} + A^{(12)}B^{(21)} \\ C^{(12)} &= A^{(11)}B^{(12)} + A^{(12)}B^{(22)} \\ C^{(21)} &= A^{(21)}B^{(11)} + A^{(22)}B^{(21)} \\ C^{(22)} &= A^{(21)}B^{(12)} + A^{(22)}B^{(22)} \end{aligned}$$

Để tính ma trận C theo cách này, chúng ta cần 8 phép nhân ma trận con. Nếu đặt kích thước của mỗi ma trận là $n = 2^s$ thì thời gian thực hiện giải thuật có thể biểu diễn bằng hàm:

$$T(n) = 8T(n/2) + \Theta(n^2)$$

Áp dụng định lý Master, trường hợp 1, ta có $T(n) = \Theta(n^3)$, không có gì tốt hơn thuật toán tính trực tiếp.

Tuy nhiên các ma trận con $C^{(11)}, C^{(12)}, C^{(21)}, C^{(22)}$ có thể tính chỉ cần 7 phép nhân ma trận con bằng thuật toán Strassen [40]. Các phép nhân ma trận con cần thực hiện là:

$$M_1 = (A^{(11)} + A^{(22)})(B^{(11)} + B^{(22)})$$

$$\begin{aligned}
M_2 &= (A^{(21)} + A^{(22)})B^{(11)} \\
M_3 &= A^{(11)}(B^{(12)} - B^{(22)}) \\
M_4 &= A^{(22)}(B^{(21)} - B^{(11)}) \\
M_5 &= (A^{(11)} + A^{(12)})B^{(22)} \\
M_6 &= (A^{(21)} - A^{(11)})(B^{(11)} + B^{(12)}) \\
M_7 &= (A^{(12)} - A^{(22)})(B^{(21)} + B^{(22)})
\end{aligned}$$

Khi đó:

$$\begin{aligned}
C^{(11)} &= M_1 + M_4 - M_5 + M_7 \\
C^{(12)} &= M_3 + M_5 \\
C^{(21)} &= M_2 + M_4 \\
C^{(22)} &= M_1 - M_2 + M_3 + M_6
\end{aligned}$$

Xét về thời gian thực hiện giải thuật Strassen, ta có:

$$T(n) = 7T(n/2) + \Theta(n^2)$$

Áp dụng định lý Master, trường hợp 1, ta có $T(n) = \Theta(n^{\lg 7}) \approx \Theta(n^{2,807})$.

Hãy cài đặt thuật toán Strassen tính tích hai ma trận.

Tự đọc thêm về thuật toán Coppersmith-Winograd [7], hiện đang là thuật toán nhanh nhất hiện nay để nhân hai ma trận trong thời gian $O(n^{2,376})$

Bài tập 2-4.

Trên mặt phẳng cho n hình tròn, hãy tính diện tích miền mặt phẳng bị n hình tròn này chiếm chỗ.

Gợi ý: Ta có thể coi các hình tròn đã cho không trùng nhau tuy có thể giao nhau. Xét một hình chữ nhật R chứa tất cả các hình tròn, ta cần tính phần diện tích của R bị các hình tròn chiếm chỗ. Nếu R có giao với chỉ một hình tròn, ta chỉ cần đo diện tích phần giao. Nếu R có giao với nhiều hơn một hình tròn, ta chia R làm bốn hình chữ nhật con ở bốn góc và tính tổng diện tích phần giao của bốn hình chữ nhật con này với các hình tròn bằng đệ quy.

Bài tập 2-5. (Cấp điểm gần nhất)

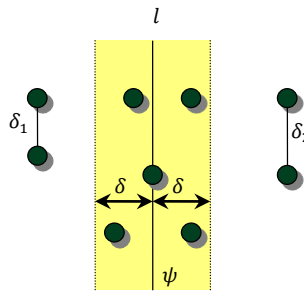
Trên mặt phẳng với hệ tọa độ trục chuẩn Oxy cho n điểm phân biệt: $P_1(x_1, y_1), P_2(x_2, y_2), \dots, P_n(x_n, y_n)$. Hãy tìm hai điểm gần nhau nhất.

Khoảng cách giữa hai điểm $P_i(x_i, y_i)$ và $P_j(x_j, y_j)$ là khoảng cách Euclid:

$$|P_i P_j| = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Để tìm cặp điểm gần nhất, chúng ta có thuật toán $\Theta(n^2)$: thử tất cả các bộ đôi và đo khoảng cách. Tuy nhiên có một thuật toán tốt hơn dựa trên chiến lược chia để trị:

- Phân hoạch tập các điểm làm hai tập rời nhau S_L và S_R , một tập gồm $\lfloor n/2 \rfloor$ điểm và một tập gồm $\lceil n/2 \rceil$ điểm thỏa mãn: Mọi điểm thuộc S_L đều có hoành độ \leq mọi điểm thuộc S_R . Điều này có thể thực hiện bằng cách tìm một đường thẳng l song song với trục tung, đặt tại hoành độ là trung vị của các hoành độ, các điểm nằm bên trái l đưa vào S_L , các điểm nằm bên phải l vào S_R , còn các điểm nằm trên l sẽ được chia vào S_L và S_R để giữ cho lực lượng của hai tập hơn kém nhau không quá 1.
- Lần lượt giải bài toán tìm cặp điểm gần nhất trên S_L và S_R , gọi δ_L và δ_R lần lượt là khoảng cách ngắn nhất tìm được trên hai tập. Đặt $\delta = \min(\delta_L, \delta_R)$.
- Nhận xét rằng nếu trong n điểm đã cho có một cặp điểm có khoảng cách ngắn hơn δ thì hai điểm đó phải có một điểm thuộc S_L và một điểm thuộc S_R . Tức là cả hai điểm phải nằm trong một dải ψ giới hạn bởi hai đường thẳng song song cách l một khoảng δ (Hình 2-2). Việc còn lại là tìm cặp điểm gần nhất trong dải này và cực tiểu hóa δ nếu cặp điểm tìm được có khoảng cách ngắn hơn cặp điểm đang có.



Hình 2-2. Thuật toán chia để trị tìm cặp điểm gần nhất

Dễ thấy rằng để hai điểm có khoảng cách nhỏ hơn δ thì điều kiện cần là độ chênh lệch tung độ giữa hai điểm phải nhỏ hơn δ .

Hãy chứng minh rằng nếu $P \in \psi$ thì có không quá 8 điểm khác thuộc ψ có độ chênh lệch tung độ nhỏ hơn δ so với P .

Hãy chỉ ra rằng nếu ban đầu các điểm được sắp xếp theo thứ tự tăng dần của tung độ thì có thể tìm thuật toán $O(n)$ để tìm cặp điểm gần nhất trong ψ nếu cặp điểm đó có khoảng cách nhỏ hơn δ . (gợi ý: chiếu các điểm của ψ lên trục tung, với mỗi điểm, từ ảnh chiếu của nó xét 8 ảnh lân cận và đo khoảng cách từ điểm đang xét tới 8 điểm tương ứng).

Hãy tìm thuật toán $O(n \lg n)$ và lập trình thuật toán đó để giải bài toán tìm cặp điểm gần nhất trong không gian hai chiều.

Hãy tìm thuật toán $O(n(\lg n)^{d-1})$ để giải bài toán tìm cặp điểm gần nhất trong không gian Euclid d chiều.

Bài 3. Quy hoạch động

Trong khoa học tính toán, quy hoạch động là một phương pháp hiệu quả để giải các bài toán tối ưu mang bản chất đệ quy.

Tên gọi "*Quy hoạch động*" – (*Dynamic Programming*) được Richard Bellman đề xuất vào những năm 1940 để mô tả một phương pháp giải các bài toán mà người giải cần đưa ra lần lượt một loạt các quyết định tối ưu. Tới năm 1953, Bellman chỉnh lại tên gọi này theo nghĩa mới và đưa ra nguyên lý giải quyết các bài toán bằng phương pháp quy hoạch động.

Không có một thuật toán tổng quát để giải tất cả các bài toán quy hoạch động. Mục đích của bài này là cung cấp một cách tiếp cận mới trong việc giải quyết các bài toán tối ưu mang bản chất đệ quy, đồng thời đưa ra các ví dụ để người đọc hình thành các kỹ năng trong việc tiếp cận và giải quyết các bài toán quy hoạch động.

3.1. Công thức truy hồi

3.1.1. Bài toán ví dụ

Chúng ta sẽ tìm hiểu về công thức truy hồi và phương pháp giải công thức truy hồi qua một ví dụ:

Cho số tự nhiên $n \leq 400$. Hãy cho biết có bao nhiêu cách phân tích số n thành tổng của dãy các số nguyên dương, các cách phân tích là hoán vị của nhau chỉ tính là một cách.

Ví dụ: $n = 5$ có 7 cách phân tích:

$$1. 5 = 1 + 1 + 1 + 1 + 1$$

$$2. 5 = 1 + 1 + 1 + 2$$

$$3. 5 = 1 + 1 + 3$$

$$4. 5 = 1 + 2 + 2$$

$$5. 5 = 1 + 4$$

$$6. 5 = 2 + 3$$

$$7. 5 = 5$$

Ta coi trường hợp $n = 0$ cũng có 1 cách phân tích thành tổng các số nguyên dương (0 là tổng của dãy rỗng)

Để giải bài toán này, mục 1.3.5 đã giới thiệu phương pháp liệt kê tất cả các cách phân tích và đếm số cấu hình. Bây giờ hãy thử nghĩ xem, có cách nào *tính ngay ra số lượng các cách phân tích mà không cần phải liệt kê* hay không. Bởi vì khi số cách phân tích tương đối lớn, phương pháp liệt kê tỏ ra khá chậm. (ví dụ chỉ với $n = 100$ đã có 190.569.292 cách phân tích).

3.1.2. Đặt công thức truy hồi

Nếu gọi $f[m, v]$ là số cách phân tích số v thành tổng các số nguyên dương $\leq m$. Khi đó các cách phân tích số v thành tổng một dãy các số nguyên dương $\leq m$ có thể chia làm hai loại:

- Loại 1: Không chứa số m trong phép phân tích, khi đó số cách phân tích loại này chính là số cách phân tích số v thành tổng các số nguyên dương $\leq m - 1$, tức là bằng $f[m - 1, v]$.
- Loại 2: Có chứa ít nhất một số m trong phép phân tích. Khi đó nếu trong các cách phân tích loại này ta bỏ đi số m đó thì ta sẽ được các cách phân tích số $v - m$ thành tổng các số nguyên dương $\leq m$. Có nghĩa là về mặt số lượng, số các cách phân tích loại này bằng $f[m, v - m]$

Trong trường hợp $m > v$ thì rõ ràng chỉ có các cách phân tích loại 1, còn trong trường hợp $m \leq v$ thì sẽ có cả các cách phân tích loại 1 và loại 2. Vì thế:

$$f[m, v] = \begin{cases} f[m - 1, v], & \text{nếu } m > v \\ f[m - 1, v] + f[m, v - m], & \text{nếu } m \leq v \end{cases} \quad (3.1)$$

Ta có công thức xây dựng $f[m, v]$ từ $f[m - 1, v]$ và $f[m, v - m]$. Công thức này có tên gọi là *công thức truy hồi (recurrence)* đưa việc tính $f[m, v]$ về việc tính các $f[m', v']$ với dữ liệu nhỏ hơn. Tất nhiên cuối cùng ta sẽ quan tâm đến $f[n, n]$: Số các cách phân tích n thành tổng các số nguyên dương $\leq n$.

Ví dụ với $n = 5$, các giá trị $f[m, v]$ có thể cho bởi bảng:

f	0	1	2	3	4	5	v
0	1	0	0	0	0	0	
1	1	1	1	1	1	1	
2	1	1	2	2	3	3	
3	1	1	2	3	4	5	
4	1	1	2	3	5	6	
5	1	1	2	3	5	7	
	m						

Từ công thức tính, ta thấy rằng $f[m, v]$ được tính qua giá trị của một phần tử ở hàng trên ($f[m - 1, v]$) và một phần tử ở cùng hàng, bên trái: ($f[m, v - m]$). Ví dụ $f[5, 5]$ sẽ được tính bằng $f[4, 5] + f[5, 0]$, hay $f[3, 5]$ sẽ được tính bằng $f[2, 5] + f[3, 2]$. Chính vì vậy để tính $f[m, v]$ thì $f[m - 1, v]$ và $f[m, v - m]$ phải được tính trước. Suy ra thứ tự hợp lý để tính các phần tử trong bảng thì ta sẽ phải tính lần lượt các hàng từ trên xuống và trên mỗi hàng thì tính theo thứ tự từ trái qua phải.

Điều đó có nghĩa là ban đầu ta phải tính hàng 0 của bảng. Theo định nghĩa $f[0, v]$ là số dãy có các phần tử là số nguyên dương ≤ 0 mà tổng bằng v , theo quy ước của đầu bài thì $f[0, 0] = 1$ và dễ thấy rằng $f[0, v] = 0, \forall v > 0$.

Bây giờ giải thuật đếm trở nên rất đơn giản:

- Khởi tạo hàng 0 của bảng $f: f[0,0] := 1; f[0,v] := 0, \forall v > 0$
- Dùng công thức truy hồi tính ra tất cả các phần tử của bảng f
- Cuối cùng cho biết $f[n,n]$ là số cách phân tích cần tìm

Chúng ta sẽ cài đặt chương trình đếm số cách phân tích số với khuôn dạng nhập/xuất dữ liệu như sau:

Input

Số tự nhiên $n \leq 400$

Output

Số cách phân tích số n

Sample Input	Sample Output
400	6727090051741041926 Analyses



NUMBERPARTITIONING1_DP.PAS ✓ Đếm số cách phân tích số

```
{ $MODE OBJFPC }
program Counting1;
const
  max = 400;
var
  f: array[0..max, 0..max] of Int64;
  n, m, v: Integer;
begin
  Readln(n);
  //Khởi tạo hàng 0 của bảng
  FillByte(f[0, 1], n * SizeOf(Int64), 0);
  f[0, 0] := 1;
  //Tính bảng f
  for m := 1 to n do
    for v := 0 to n do
      if v < m then f[m, v] := f[m - 1, v]
      else f[m, v] := f[m - 1, v] + f[m, v - m];
  Writeln(f[n, n], ' Analyses');
end.
```

3.1.3. Cải tiến thứ nhất

Cách làm trên có thể tóm tắt lại như sau: Khởi tạo hàng 0 của bảng, sau đó dùng hàng 0 tính hàng 1, dùng hàng 1 tính hàng 2, v.v... tới khi tính được hết hàng n . Có thể nhận thấy rằng khi đã tính xong hàng thứ m thì việc lưu trữ các hàng từ 0 tới $m - 1$ là không cần thiết bởi vì việc tính hàng $+1$ chỉ phụ thuộc các giá trị lưu trữ trên hàng m . Do đó quá trình tính toán chỉ cần lưu trữ hai hàng của mảng f : hàng x tương ứng với hàng vừa được tính của bảng f và hàng y tương ứng với hàng sắp tính của bảng f : Đầu tiên hàng x được gán các giá trị tương ứng trên hàng 0 của bảng f ; sau đó, công thức truy hồi sẽ được áp dụng để tính hàng y từ hàng x . Tiếp theo, hai hàng x và y sẽ đổi vai trò cho nhau và công thức truy hồi tiếp tục dùng hàng x tính

hàng y , hàng y sau khi tính sẽ gồm các giá trị tương ứng trên hàng 2 của bảng f , v.v... Vậy ta có một cách cài đặt cải tiến sau:

NUMBERPARTITIONING2_DP.PAS ✓ Đếm số cách phân tích số

```
{ $MODE OBJFPC }
program Counting2;
const
  max = 400;
var
  f: array[0..1, 0..max] of Int64;
  x, y: Integer;
  n, m, v: Integer;
begin
  Readln(n);
  FillByte(f[0, 1], n * SizeOf(Int64), 0);
  f[0, 0] := 1;
  x := 0; y := 1; //x: hàng đã có, y: hàng đang tính
  for m := 1 to n do
    begin //Dùng hàng x tính hàng y
      for v := 0 to n do
        if v < m then f[y, v] := f[x, v]
        else f[y, v] := f[x, v] + f[y, v - m];
        x := 1 - x; y := 1 - y; //Đảo vai trò x và y, tính xoay lại
      end;
      Writeln(f[x, n], ' Analyses');
    end.
end.
```

3.1.4. Cải tiến thứ hai

Ta vẫn còn cách tốt hơn nữa, tại mỗi bước, ta chỉ cần lưu lại một hàng của bảng f như mảng một chiều, sau đó áp dụng công thức truy hồi lên mảng đó tính lại chính nó, để sau khi tính, mảng một chiều sẽ chứa các giá trị trên hàng kế tiếp của bảng f .

NUMBERPARTITIONING3_DP.PAS ✓ Đếm số cách phân tích số

```
{ $MODE OBJFPC }
program Counting3;
const
  max = 400;
var
  f: array[0..max] of Int64;
  n, m, v: Integer;
begin
  Readln(n);
  FillByte(f[1], n * SizeOf(Int64), 0);
  f[0] := 1;
  for m := 1 to n do
    for v := m to n do //Chỉ cần tính lại các f[v] với v ≥ m
      Inc(f[v], f[v - m]);
    Writeln(f[n], ' Analyses');
  end.
end.
```

3.1.5. Cài đặt đệ quy

Xem lại công thức truy hồi tính $f[m, v] = f[m - 1, v] + f[m, v - m]$, ta nhận thấy rằng để tính $f[m, v]$ ta phải biết được chính xác $f[m - 1, v]$ và $f[m, v - m]$. Như vậy việc xác định thứ tự tính các phần tử trong bảng f (phần tử nào tính trước, phần tử nào tính sau) là quan trọng. Trong trường hợp thứ tự này khó xác định, ta có thể tính dựa trên một hàm đệ quy mà không cần phải quan tâm tới thứ tự tính toán. Trước khi trình bày phương pháp này, ta sẽ thử viết một hàm đệ quy theo cách quen thuộc để giải công thức truy hồi:

NUMBERPARTITIONING4_RECUR.PAS ✓ Đếm số cách phân tích số

```
{ $MODE OBJFPC }
program Counting4;
var
  n: Integer;

function Getf(m, v: Integer): Int64;
begin
  if m = 0 then //phần neo
    if v = 0 then Result := 1
    else Result := 0
  else //phần đệ quy
    if v < m then Result := Getf(m - 1, v)
    else Result := Getf(m - 1, v) + Getf(m, v - m);
end;

begin
  Readln(n);
  Writeln(Getf(n, n), ' Analyses');
end.
```

Phương pháp cài đặt này tỏ ra khá chậm vì với mỗi giá trị của m và v , hàm $Getf(m, v)$ có thể bị tính nhiều lần (bài sau sẽ giải thích rõ hơn điều này). Ta có thể cải tiến bằng cách kết hợp hàm đệ quy $Getf(m, v)$ với một mảng hai chiều f . Ban đầu các phần tử của f được coi là “chưa biết” (bằng cách gán một giá trị đặc biệt). Hàm $Getf(m, v)$ trước hết sẽ tra cứu tới $f[m, v]$, nếu $f[m, v]$ chưa biết thì hàm $Getf(m, v)$ sẽ gọi đệ quy để tính giá trị của $f[m, v]$ rồi dùng giá trị này gán cho kết quả hàm, còn nếu $f[m, v]$ đã biết thì hàm này chỉ việc gán kết quả hàm là $f[m, v]$ mà không cần gọi đệ quy để tính toán nữa.

NUMBERPARTITIONING5_DP.PAS ✓ Đếm số cách phân tích số

```
{ $MODE OBJFPC }
program Counting5;
const
  max = 400;
var
  n: Integer;
  f: array[0..max, 0..max] of Int64;

function Getf(m, v: Integer): Int64; //Tính f[m, v]
begin
  if f[m, v] = -1 then //Nếu f[m, v] chưa được tính thì đi tính f[m, v]
```

```

if m = 0 then //phần neo
  if v = 0 then f[m, v] := 1
  else f[m, v] := 0
else //phần đệ quy
  if v < m then f[m, v] := Getf(m - 1, v)
  else f[m, v] := Getf(m - 1, v) + Getf(m, v - m);
Result := f[m, v]; //Gán kết quả hàm bằng f[m, v]
end;

begin
  Readln(n);
  FillByte(f, SizeOf(f), $FF); //Các phần tử của f được gán giá trị đặc biệt (-1)
  Writeln(Getf(n, n), ' Analyses');
end.

```

Việc sử dụng phương pháp đệ quy để giải công thức truy hồi là một kỹ thuật đáng lưu ý, vì khi gặp một công thức truy hồi phức tạp, khó xác định thứ tự tính toán thì phương pháp này tỏ ra rất hiệu quả, hơn thế nữa nó làm rõ hơn bản chất đệ quy của công thức truy hồi.

Ngoài ra, nếu nhập vào $n = 5$ và in ra bảng f sau khi tính, ta sẽ được bảng sau:

f	0	1	2	3	4	5
0	1	0	0	0	0	0
1	1	1	1	1	1	1
2	1	1	2	2	-1	3
3	1	1	2	-1	-1	5
4	1	1	-1	-1	-1	6
5	1	-1	-1	-1	-1	7

Nhìn vào bảng kết quả f với $n = 5$, ta thấy còn rất nhiều phần tử mang giá trị -1 (chưa được tính), điều này chỉ ra rằng cài đặt đệ quy còn cho phép chúng ta bỏ qua không cần tính những phần tử không tham gia vào việc tính $f[n, n]$ và làm tăng tốc đáng kể quá trình tính toán.

3.1.6. Tóm tắt kỹ thuật giải công thức truy hồi

Công thức truy hồi có vai trò quan trọng trong bài toán đếm, chẳng hạn đếm số cấu hình thỏa mãn điều kiện nào đó, hoặc tìm tương ứng giữa một số thứ tự và một cấu hình.

Ví dụ trong bài này là thuộc dạng đếm số cấu hình thỏa mãn điều kiện nào đó. Tuy nhiên nếu đặt lại bài toán: Nếu đem tất cả các dãy số nguyên dương không giảm có tổng bằng n sắp xếp theo thứ tự từ điển thì dãy thứ p là dãy nào?, hoặc cho một dãy số nguyên dương không giảm có tổng bằng n , hỏi dãy đó đứng thứ mấy?. Lúc này ta sẽ có bài toán tìm tương ứng giữa một số thứ tự và một cấu hình - Một dạng bài toán có thể giải quyết hiệu quả bằng công thức truy hồi.

Ta cũng đã khảo sát một vài kỹ thuật phổ biến để giải công thức truy hồi: Phương pháp tính trực tiếp bảng giá trị, phương pháp tính luân phiên (chỉ phải lưu trữ các phần tử tích cực), phương pháp tự cập nhật giá trị, và phương pháp đệ quy kết hợp với bảng lưu trữ. Những phương pháp này sẽ đóng vai trò quan trọng trong việc cài đặt chương trình giải công thức truy hồi - Hạt nhân của bài toán quy hoạch động.

3.1.7. ★ Nói thêm về bài toán phân tích số

Bài toán *phân tích số* (*integer partitioning*) là một bài toán quan trọng trong lĩnh vực số học và vật lý. Chúng ta không đi sâu vào chi tiết hai lĩnh vực này mà chỉ nói tới một kết quả đã được chứng minh bằng lý thuyết số học về các *số ngũ giác* (*pentagonal numbers*):

Những số ngũ giác là những số nguyên có dạng $p(k) = \frac{k(3k-1)}{2}$ với $k \in \mathbb{Z}$. Bằng cách cho k nhận lần lượt các giá trị trong dãy $0, +1, -1, +2, -2, +3, -3, \dots$ ta có thể liệt kê các số ngũ giác theo thứ tự tăng dần là:

$$0, 1, 2, 5, 7, 12, 15, \dots$$

Gọi $f(n)$ là số cách phân tích số n thành tổng của các số nguyên dương (không tính hoán vị). Quy ước $f(0) = 1$ và $f(n) = 0$ với $\forall n < 0$, khi đó:

$$\begin{aligned} f(n) &= \sum_{k=1}^{\infty} (-1)^{k+1} (f(n - p(k)) + f(n - p(-k))) \\ &= f(n - 1) + f(n - 2) - f(n - 5) - f(n - 7) + \dots \end{aligned} \quad (3.2)$$

Chúng ta có thể sử dụng công thức truy hồi (3.2) để đếm số cách phân tích số trong thời gian $O(n\sqrt{n})$ thay vì $\Theta(n^2)$ nếu dùng công thức (3.1) như các chương trình trước. Trong chương trình dưới đây, để tránh lỗi tràn số khi phép cộng có thể cho kết quả vượt quá phạm vi biểu diễn số nguyên 64 bit, chúng ta tính $f(n)$ theo cách:

$$\begin{aligned} SumA &:= \sum_{k=1}^{\infty} (-1)^{k+1} f(n - p(k)) \\ SumB &:= \sum_{k=1}^{\infty} (-1)^{k+1} f(n - p(-k)) \\ f(n) &:= SumA + SumB \end{aligned} \quad (3.3)$$

NUMBERPARTITIONING6_DP.PAS ✓ Đếm số cách phân tích số

```
{ $MODE OBJFPC }
program Counting6;
const
  max = 400;
var
  n, i, k, s: Integer;
  f: array[0..max] of Int64;
  SumA, SumB: Int64;

function p(k: Integer): Integer; //Số ngũ giác p(k)
begin
  Result := k * (3 * k - 1) shr 1;
end;

begin
  Readln(n);
```

```

f[0] := 1;
for i := 1 to n do
  begin
    SumA := 0;
    SumB := 0;
    s := 1;
    k := 1;
    while p(k) <= i do
      begin
        Inc(SumA, s * f[i - p(k)]);
        if p(-k) <= i then
          Inc(SumB, s * f[i - p(-k)]);
        Inc(k);
        s := -s; //Đảo dấu hạng tử
      end;
    f[i] := SumA + SumB;
  end;
  Writeln(f[n], ' Analyses');
end.

```

3.2. Phương pháp quy hoạch động

3.2.1. Bài toán quy hoạch động

Trong toán học và khoa học máy tính, *quy hoạch động* (*dynamic programming*) là một phương pháp hiệu quả giải những bài toán tối ưu có ba tính chất sau đây:

- Bài toán lớn có thể phân rã thành những bài toán con đồng dạng, những bài toán con đó có thể phân rã thành những bài toán nhỏ hơn nữa ...(*recursive form*).
- Lời giải tối ưu của các bài toán con có thể sử dụng để tìm ra lời giải tối ưu của bài toán lớn (*optimal substructure*)
- Hai bài toán con trong quá trình phân rã có thể có chung một số bài toán con khác (*overlapping subproblems*).

Tính chất thứ nhất và thứ hai là điều kiện cần của một bài toán quy hoạch động. Tính chất thứ ba nêu lên đặc điểm của một bài toán mà cách giải bằng phương pháp quy hoạch động hiệu quả hơn hẳn so với phương pháp giải đệ quy thông thường.

Với những bài toán có hai tính chất đầu tiên, chúng ta thường nghĩ đến các thuật toán chia để trị và đệ quy: Để giải quyết một bài toán lớn, ta chia nó ra thành nhiều bài toán con đồng dạng và giải quyết độc lập các bài toán con đó.

Khác với thuật toán đệ quy, phương pháp quy hoạch động thêm vào cơ chế lưu trữ nghiệm hay một phần nghiệm của mỗi bài toán khi giải xong nhằm mục đích *sử dụng lại*, hạn chế những thao tác thừa trong quá trình tính toán.

Chúng ta xét một ví dụ đơn giản: Dãy Fibonacci là dãy vô hạn các số nguyên dương f_1, f_2, \dots được định nghĩa bằng công thức truy hồi sau:

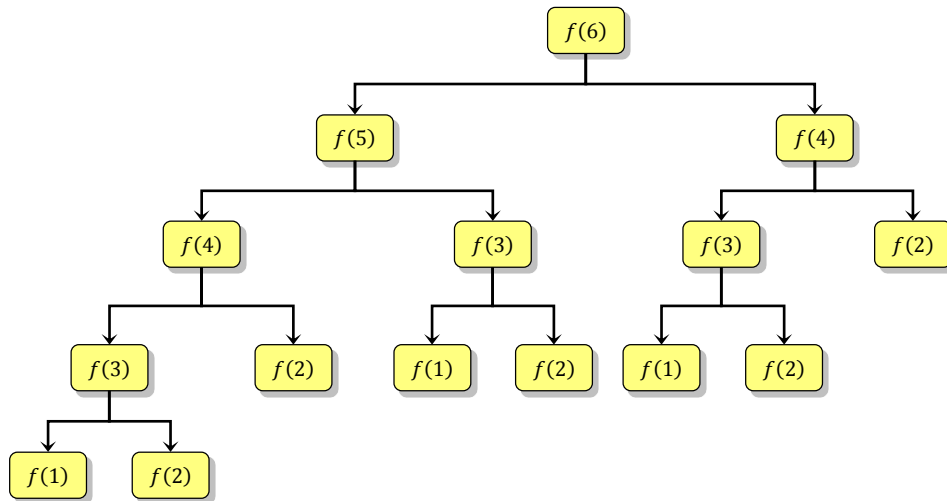
$$f_i = \begin{cases} 1, & \text{nếu } i \leq 2 \\ f_{i-1} + f_{i-2}, & \text{nếu } i > 2 \end{cases}$$

Hãy tính f_6 .

Xét đoạn chương trình sau:

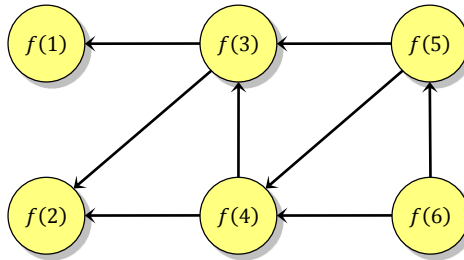
```
program TopDown;  
  
function f(i: Integer): Integer;  
begin  
  if i ≤ 2 then Result := 1  
  else Result := f(i - 1) + f(i - 2);  
end;  
  
begin  
  Output ← f(6);  
end.
```

Trong cách giải này, hàm đệ quy $f(i)$ được dùng để tính số Fibonacci thứ i . Chương trình chính gọi $f(6)$, nó sẽ gọi tiếp $f(5)$ và $f(4)$ để tính ... Quá trình tính toán có thể vẽ như cây trong Hình 3-1.



Hình 3-1. Hàm đệ quy tính số Fibonacci

Ta thấy rằng để tính $f(6)$, máy phải tính một lần $f(5)$, hai lần $f(4)$, ba lần $f(3)$, năm lần $f(2)$ và ba lần $f(1)$. Sự thiếu hiệu quả có thể giải thích bởi tính chất: Hai bài toán trong quá trình phân rã có thể có chung một số bài toán con. Ví dụ nếu coi lời gọi hàm $f(i)$ tương ứng với bài toán tính số Fibonacci thứ i , thì bài toán $f(4)$ là bài toán con chung của cả $f(5)$ và $f(6)$. Điều đó chỉ ra rằng nếu tính $f(5)$ và $f(6)$ một cách độc lập, bài toán $f(4)$ sẽ phải giải hai lần.



Hình 3-2. Tính chất tập các bài toán con gối nhau (overlapping subproblems) khi tính số Fibonacci.

Để khắc phục những nhược điểm trong việc giải độc lập các bài toán con, mỗi khi giải xong một bài toán trong quá trình phân rã, chúng ta sẽ *lưu trữ lời giải* của nó nhằm mục đích *sử dụng lại*. Chẳng hạn với bài toán tính số Fibonacci thứ 6, ta có thể dùng một mảng $f_{1..6}$ để lưu trữ các giá trị số Fibonacci, khởi tạo sẵn giá trị cho f_1 và f_2 bằng 1, từ đó tính tiếp lần lượt f_3, f_4, f_5, f_6 , đảm bảo mỗi giá trị Fibonacci chỉ phải tính một lần:

```

program BottomUp;
var
  f: array[1..6] of Integer;
  i: Integer;
begin
  f[1] := 1; f[2] := 1;
  for i := 3 to 6 do
    f[i] := f[i - 1] + f[i - 2];
  Output ← f[6];
end.

```

Kỹ thuật lưu trữ lời giải của các bài toán con nhằm mục đích sử dụng lại được gọi là *memoization**, nếu tới một bước nào đó mà lời giải một bài toán con không còn cần thiết nữa, chúng ta có thể bỏ lời giải đó đi khỏi không gian lưu trữ để tiết kiệm bộ nhớ. Ví dụ khi ta đang cần tính f_6 thì chỉ cần biết giá trị f_5 và f_4 là đủ nên việc lưu trữ các giá trị $f_{1..3}$ là vô nghĩa. Những kỹ thuật này chúng ta đã bàn đến trong mục 3.1.3 và 3.1.4. Cụ thể bài toán tính số Fibonacci, ta có thể cài đặt dùng hai biến luân phiên:

* Từ này không có trong từ điển tiếng Anh, có thể coi như từ đồng nghĩa với *memorization*, gốc từ: *memo* (Bản ghi nhớ)

```

program BottomUp;
var
  a, b: Integer;
  i: Integer;
begin
  a := 1; b := 1;
  for i := 3 to 6 do
    begin
      b := a + b; //b := f[i]
      a := b - a; //a := f[i-1]
    end;
  Output ← b;
end.

```

Cách giải này bắt đầu từ việc giải bài toán nhỏ nhất, lưu trữ nghiệm và từ đó giải quyết những bài toán lớn hơn...cách tiếp cận này gọi là cách tiếp cận từ dưới lên (bottom-up). Cách tiếp cận từ dưới lên không cần có chương trình con đệ quy, vì vậy bớt đi một số đáng kể lời gọi chương trình con và tiết kiệm được bộ nhớ chứa tham số và biến địa phương của chương trình con đệ quy. Tuy nhiên với cách tiếp cận từ dưới lên, chúng ta phải xác định rõ thứ tự giải các bài toán để khi bắt đầu giải một bài toán nào đó thì tất cả các bài toán con của nó phải được giải sẵn từ trước, điều này đôi khi tương đối phức tạp (xác định thứ tự tô-pô trên tập các bài toán phân rã). Trong trường hợp khó (hoặc không thể) xác định thứ tự hợp lý giải quyết các bài toán con, người ta sử dụng cách tiếp cận từ trên xuống (top-down) kết hợp với kỹ thuật lưu trữ nghiệm (memoization), kỹ thuật này chúng ta cũng đã bàn đến trong mục 3.1.5. Với bài toán tính số Fibonacci, ta có thể cài đặt bằng hàm đệ quy như sau:

```

program TopDownWithMemoization;
var
  f: array[1..6] of Integer;
  i: Integer;

function Getf(i: Integer): Integer;
begin
  if f[i] = 0 then //f[i] chưa được tính thì mới đi tính f[i]
    if i ≤ 2 then f[i] := 1
    else f[i] := Getf(i - 1) + Getf(i - 2);
  Result := f[i]; //Gán kết quả hàm bằng f[i]
end;

begin
  for i := 1 to 6 do f[i] := 0;
  Output ← Getf(6);
end.

```

Trước khi đi vào phân tích cách tiếp cận một bài toán quy hoạch động, ta làm quen với một số thuật ngữ sau đây:

- Bài toán giải theo phương pháp quy hoạch động gọi là *bài toán quy hoạch động*.

- Công thức phối hợp nghiệm của các bài toán con để có nghiệm của bài toán lớn gọi là *công thức truy hồi* của quy hoạch động.
- Tập các bài toán nhỏ nhất có ngay lời giải để từ đó giải quyết các bài toán lớn hơn gọi là *cơ sở quy hoạch động*.
- Không gian lưu trữ lời giải các bài toán con để tìm cách phối hợp chúng gọi là *bảng phương án của quy hoạch động*.

3.2.2. Cách giải một bài toán quy hoạch động

Việc giải một bài toán quy hoạch động phải qua khá nhiều bước, từ những phân tích ở trên, ta có thể hình dung:

Quy hoạch động = Chia để trị + Cơ chế lưu trữ nghiệm để sử dụng lại

(Dynamic Programming = Divide and Conquer + Memoization)

Không có một “thuật toán” nào cho chúng ta biết một bài toán có thể giải bằng phương pháp quy hoạch động hay không? Khi gặp một bài toán cụ thể, chúng ta phải phân tích bài toán, sau đó kiểm chứng ba tính chất của một bài toán quy hoạch động trước khi tìm lời giải bằng phương pháp quy hoạch động.

Nếu như bài toán đặt ra đúng là một bài toán quy hoạch động thì việc đầu tiên phải làm là phân tích xem một bài toán lớn có thể phân rã thành những bài toán con đồng dạng như thế nào, sau đó xây dựng cách tìm nghiệm của bài toán lớn trong điều kiện chúng ta đã biết nghiệm của những bài toán con - *tìm công thức truy hồi*. Đây là công đoạn khó nhất vì chẳng có phương pháp tổng quát nào xây dựng công thức truy hồi cả, tất cả đều dựa vào kinh nghiệm và độ nhạy cảm của người lập trình khi đọc và phân tích bài toán.

Sau khi xây dựng công thức truy hồi, ta phải phân tích xem tại mỗi bước tính nghiệm của một bài toán, chúng ta *cần lưu trữ bao nhiêu bài toán con* và với mỗi bài toán con thì *cần lưu trữ toàn bộ nghiệm hay một phần nghiệm*. Từ đó xác định lượng bộ nhớ cần thiết để lưu trữ, nếu lượng bộ nhớ cần huy động vượt quá dung lượng cho phép, chúng ta cần tìm một công thức khác hoặc thậm chí, một cách giải khác không phải bằng quy hoạch động.

Một điều không kém phần quan trọng là phải chỉ ra được *bài toán nào cần phải giải trước bài toán nào*, hoặc chí ít là chỉ ra được khái niệm bài toán này nhỏ hơn bài toán kia nghĩa là thế nào. Nếu không, ta có thể rơi vào quá trình lòng vòng: Giải bài toán P_1 cần phải giải bài toán P_2 , giải bài toán P_2 lại cần phải giải bài toán P_1 (công thức truy hồi trong trường hợp này là không thể giải được). Cũng nhờ chỉ rõ quan hệ “nhỏ hơn” giữa các bài toán mà ta có thể xác định được một tập các bài toán nhỏ nhất có thể giải trực tiếp, nghiệm của chúng sau đó được dùng làm cơ sở giải những bài toán lớn hơn.

Trong đa số các bài toán quy hoạch động, nếu bạn tìm được đúng công thức truy hồi và xác định đúng tập các bài toán cơ sở thì coi như phần lớn công việc đã xong. Việc tiếp theo là cài đặt chương trình giải công thức truy hồi:

- Nếu giải công thức truy hồi theo cách tiếp cận từ dưới lên, trước hết cần giải tất cả các bài toán cơ sở, lưu trữ nghiệm vào bảng phương án, sau đó dùng công thức truy hồi tính nghiệm của những bài toán lớn hơn và lưu trữ vào bảng phương án cho tới khi bài toán ban đầu được giải. Cách này yêu cầu phải xác định được chính xác thứ tự giải các bài toán (từ nhỏ đến lớn). Ưu điểm của nó là cho phép dễ dàng loại bỏ nghiệm những bài toán con không dùng đến nữa để tiết kiệm bộ nhớ lưu trữ cho bảng phương án.
- Nếu giải công thức truy hồi theo cách tiếp cận từ trên xuống, bạn cần phải có cơ chế đánh dấu bài toán nào đã được giải, bài toán nào chưa. Nếu bài toán chưa được giải, nó sẽ được phân rã thành các bài toán con và giải bằng đệ quy. Cách này không yêu cầu phải xác định thứ tự giải các bài toán nhưng cũng chính vì vậy, khó khăn sẽ gặp phải nếu muốn loại bỏ bớt nghiệm của những bài toán con không dùng đến nữa để tiết kiệm bộ nhớ lưu trữ cho bảng phương án.

Công đoạn cuối cùng là chỉ ra nghiệm của bài toán ban đầu. Nếu bảng phương án lưu trữ toàn bộ nghiệm của các bài toán thì chỉ việc đưa ra nghiệm của bài toán ban đầu, nếu bảng phương án chỉ lưu trữ một phần nghiệm, thì trong quá trình tính công thức truy hồi, ta để lại một “manh mối” nào đó (gọi là vết) để khi kết thúc quá trình giải, ta có thể truy vết tìm ra toàn bộ nghiệm.

Chúng ta sẽ khảo sát một số bài toán quy hoạch động kinh điển để hình dung được những kỹ thuật cơ bản trong việc phân tích bài toán quy hoạch động, tìm công thức truy hồi cũng như lập trình giải các bài toán quy hoạch động.

3.3. Một số bài toán quy hoạch động

3.3.1. Dãy con đơn điệu tăng dài nhất

Cho dãy số nguyên $A = (a_1, a_2, \dots, a_n)$. 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. Tức là tìm một số k lớn nhất và dãy chỉ số $i_1 < i_2 < \dots < i_k$ sao cho $a_{i_1} < a_{i_2} < \dots < a_{i_k}$.

Input

- Dòng 1 chứa số n ($n \leq 10^6$)
- Dòng 2 chứa n số nguyên a_1, a_2, \dots, a_n ($\forall i: |a_i| \leq 10^6$)

Output

Dãy con đơn điệu tăng dài nhất của dãy $A = (a_1, a_2, \dots, a_n)$

Sample Input	Sample Output
12	Length = 8
1 2 3 8 9 4 5 6 2 3 9 10	a[1] = 1
	a[2] = 2
	a[3] = 3
	a[6] = 4
	a[7] = 5
	a[8] = 6
	a[11] = 9
	a[12] = 10

□ Thuật toán

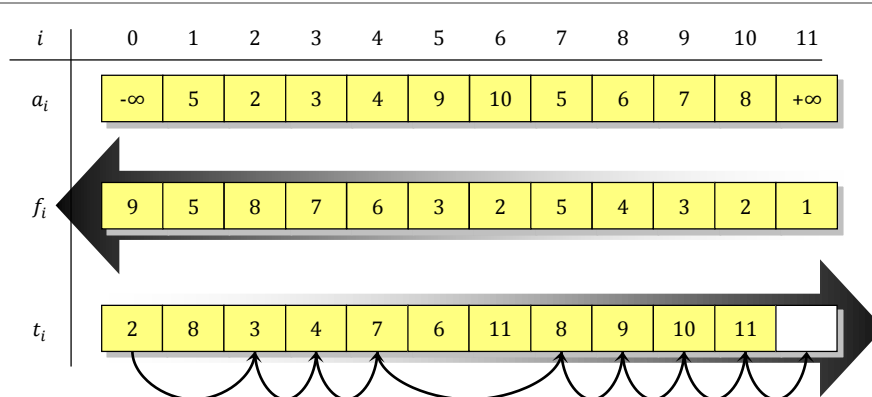
Bổ sung vào A hai phần tử: $a_0 = -\infty$ và $a_{n+1} = +\infty$. Khi đó dãy con đơn điệu tăng dài nhất của dãy A chắc chắn sẽ bắt đầu từ a_0 và kết thúc ở a_{n+1} . Với mỗi giá trị i ($0 \leq i \leq n+1$), gọi $f[i]$ là độ dài dãy con đơn điệu tăng dài nhất bắt đầu tại a_i .

Ta sẽ xây dựng cách tính các giá trị $f[i]$ bằng công thức truy hồi. Theo định nghĩa, $f[i]$ là độ dài dãy con đơn điệu tăng dài nhất bắt đầu tại a_i , dãy con này 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 nào đó đứng sau a_i . Ta sẽ chọn dãy nào để ghép thêm a_i vào đầu?, tất nhiên ta 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 $f[i]$ được tính như sau: Xét tất cả các chỉ số j từ $i+1$ đến $n+1$ mà $a_j > a_i$, chọn ra chỉ số j_{max} có $f[j_{max}]$ lớn nhất và đặt $f[i] := f[j_{max}] + 1$.

Kỹ thuật lưu vết sẽ được thực hiện song song với quá trình tính toán. Mỗi khi tính $f[i] := f[j_{max}] + 1$, ta đặt $t[i] := j_{max}$ để ghi nhận rằng dãy con dài nhất bắt đầu tại a_i sẽ có phần tử kế tiếp là $a_{j_{max}}$:

$$\begin{aligned} f[i] &:= \max\{f[j]: i < j \leq n+1 \text{ và } a_j > a_i\} + 1 \\ t[i] &:= \arg \max\{f[j]: i < j \leq n+1 \text{ và } a_j > a_i\} \end{aligned} \quad (3.4)$$

Từ công thức truy hồi tính các giá trị $f[.]$, ta nhận thấy rằng để tính $f[i]$ thì các giá trị $f[j]$ với $j > i$ phải được tính trước. Tức là thứ tự tính toán đúng phải là tính từ cuối mảng f về đầu mảng f . Với thứ tự tính toán như vậy, cơ sở quy hoạch động (bài toán nhỏ nhất) sẽ là phép tính $f[n+1]$, theo định nghĩa thì dãy con đơn điệu tăng dài nhất bắt đầu tại $a_{n+1} = +\infty$ chỉ có một phần tử là chính nó nên $f[n+1] = 1$.



Hình 3-3. Giải công thức truy hồi và truy vết

Sau khi tính xong các giá trị $f[.]$ và $t[.]$ việc chỉ ra dãy con đơn điệu tăng dài nhất được thực hiện bằng cách truy vết theo chiều ngược lại, từ đầu mảng t về cuối mảng t , cụ thể là ta bắt đầu từ phần tử $t[0]$:

$i_1 = t[0]$ chính là chỉ số phần tử đầu tiên được chọn (a_{i_1}).

$i_2 = t[i_1]$ là chỉ số phần tử thứ hai được chọn (a_{i_2}).

$i_3 = t[i_2]$ là chỉ số phần tử thứ ba được chọn (a_{i_3}).

...

❑ Cài đặt

Dưới đây ta sẽ cài đặt chương trình giải bài toán tìm dãy con đơn điệu tăng dài nhất theo những phân tích trên.

📄 LIS.PAS ✓ Tìm dãy con đơn điệu tăng dài nhất

```
{ $MODE OBJFPC }
program LongestIncreasingSubsequence;
const
  max = 1000000;
  maxV = 1000000;
var
  a, f, t: array[0..max + 1] of Integer;
  n: Integer;

procedure Enter; //Nhập dữ liệu
var
  i: Integer;
begin
  ReadLn(n);
  for i := 1 to n do Read(a[i]);
end;

procedure Optimize; //Quy hoạch động
var
  i, j, jmax: Integer;
```

```

begin
  a[0] := -maxV - 1; a[n + 1] := maxV + 1;
  f[n + 1] := 1; //Đặt cơ sở quy hoạch động
  for i := n downto 0 do //Giải công thức truy hồi
    begin //Tính f[i]
      //Tìm jmax > j thoả mãn a[jmax] > a[j] và f[jmax] lớn nhất
      jmax := n + 1;
      for j := i + 1 to n do
        if (a[j] > a[i]) and (f[j] > f[jmax]) then jmax := j;
      f[i] := f[jmax] + 1; //Lưu trữ nghiệm - memoization
      t[i] := jmax; //Lưu vết
    end;
  end;

  procedure PrintResult; //In kết quả
  var
    i: Integer;
  begin
    WriteLn('Length = ', f[0] - 2); //Dãy kết quả phải bỏ đi hai phần tử a[0] và a[n + 1]
    i := t[0]; //Truy vết bắt đầu từ t[0]
    while i <> n + 1 do
      begin
        WriteLn('a[' , i , '] = ', a[i]);
        i := t[i]; //Xét phần tử kế tiếp
      end;
    end;

  begin
    Enter;
    Optimize;
    PrintResult;
  end.

```

□ Cải tiến

Nhắc lại công thức truy hồi tính các $f[i]$ là:

$$f[i] = \begin{cases} 1, & \text{nếu } i = n + 1 \\ \max \{f[j]: (j > i) \text{ và } (a_j > a_i)\} + 1, & \text{nếu } 0 \leq i \leq n \end{cases} \quad (3.5)$$

Để giải công thức truy hồi bằng cách cài đặt trên, ta sẽ cần thời gian $\Omega(n^2)$ để tính mảng f . Với kích thước dữ liệu lớn ($n \approx 10^6$) thì chương trình này chạy rất chậm. Ta cần một giải pháp tốt hơn cho bài toán này.

Quan sát sau đây sẽ cho ta một thuật toán với độ phức tạp tính toán là $O(n \lg m)$ với n là độ dài dãy A và m là độ dài dãy con đơn điệu tăng dài nhất tìm được.

Xét đoạn cuối dãy A bắt đầu từ chỉ số i : $(a_i, a_{i+1}, \dots, a_{n+1} = +\infty)$. Gọi:

- m là độ dài dãy con tăng dài nhất trong đoạn này.
- Với mỗi độ dài λ ($1 \leq \lambda \leq m$), gọi s_λ là chỉ số của phần tử a_{s_λ} thoả mãn: Tồn tại dãy đơn điệu tăng độ dài λ bắt đầu từ a_{s_λ} . Nếu có nhiều phần tử trong dãy A cùng thoả mãn điều kiện này thì ta chọn a_{s_λ} là phần tử lớn nhất trong số những phần tử đó.

- Với mỗi chỉ số j ($i \leq j \leq n$), gọi t_j là chỉ số phần tử liền sau phần tử a_j trong dãy con đơn điệu tăng dài nhất bắt đầu tại a_j .

Chúng ta sẽ tính m , các giá trị $s[\lambda]$ và các giá trị t_j bằng phương pháp quy hoạch động theo i .

Trường hợp cơ sở: Rõ ràng khi $i = n + 1$, đoạn cuối dãy A bắt đầu từ chỉ số i chỉ có một phần tử là $a_{n+1} = +\infty$ nên trong trường hợp này, $m = 1, s_1 = n + 1$.

Nhận xét rằng nếu đã biết được các giá trị $m, s_{(\cdot)}$ và $t_{(\cdot)}$ tương ứng với một chỉ số i nào đó thì các giá trị $s_{(\cdot)}$ có tính chất:

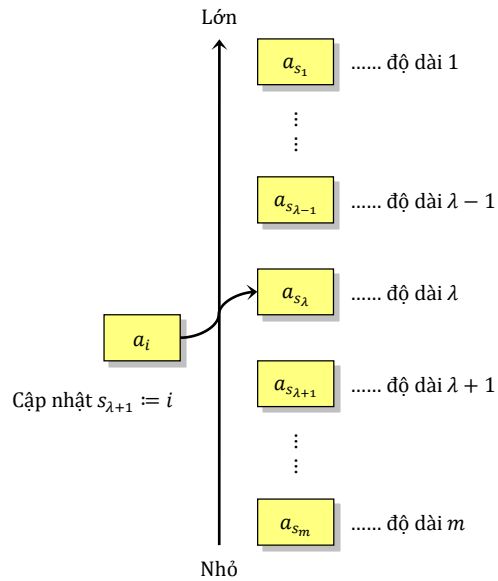
$$a_{s_m} < a_{s_{m-1}} < \dots < a_{s_1} \quad (3.6)$$

Thật vậy, vì dãy con tăng dài nhất bắt đầu tại a_{s_λ} có độ dài λ , dãy con này nếu xét từ phần tử thứ hai trở đi sẽ tạo thành một dãy con tăng độ dài $\lambda - 1$. Phần tử thứ hai này chắc chắn nhỏ hơn $a_{s_{\lambda-1}}$ (theo cách xây dựng $s_{\lambda-1}$). Vậy ta có $a_{s_\lambda} < a_{s_{\lambda-1}} (\forall \lambda)$.

Để tính các giá trị $m, s_{(\cdot)}$ và $t_{(\cdot)}$ tương ứng với chỉ số $i = C$, trong trường hợp đã biết các giá trị $m, s_{(\cdot)}$ và $t_{(\cdot)}$ tương ứng với chỉ số $i = C + 1$, ta có thể làm như sau:

- Dùng thuật toán tìm kiếm nhị phân trên dãy $a_{s_m} < a_{s_{m-1}} < \dots < a_{s_1}$ để tìm giá trị λ lớn nhất thỏa mãn $a_i < a_{s_\lambda}$. Đến đây ta xác định được độ dài dãy con tăng dài nhất bắt đầu tại a_i là $\lambda + 1$, lưu lại vết $t_i = s_\lambda$. Có thể hiểu thao tác này là đem a_i nối vào đầu một dãy con tăng độ dài λ để được một dãy con tăng độ dài $\lambda + 1$.
- Cập nhật lại $m_{\text{mới}} := \max(m_{\text{cũ}}, \lambda + 1)$ và $s_{\lambda+1} := i$.

Bản chất của thuật toán là tại mỗi bước ta lưu thông tin về các dãy con tăng độ dài $1, 2, \dots, m$. Nếu có nhiều dãy con tăng cùng một độ dài, ta chỉ quan tâm tới dãy có phần tử bắt đầu lớn nhất (để dễ nối thêm một phần tử khác vào đầu dãy). Hiện tại $a_{s_{\lambda+1}}$ đang là phần tử bắt đầu của một dãy con tăng độ dài $\lambda + 1$. Sau khi nối a_i vào đầu một dãy con tăng độ dài λ thì a_i cũng là phần tử bắt đầu của một dãy con tăng độ dài $\lambda + 1$. Xét về mặt giá trị, do tính chất của thuật toán tìm kiếm nhị phân ở trên, ta có $a_i \geq a_{s_{\lambda+1}}$, tức là dãy bắt đầu tại a_i "dễ" nối thêm phần tử khác vào đầu hơn nên ta cập nhật lại $s_{\lambda+1} := i$. (Hình 3-4)



Hình 3-4

LIS2.PAS ✓ Tìm dãy con đơn điệu tăng dài nhất (cải tiến)

```

{$MODE OBJFPC}
program LongestIncreasingSubsequence;
const
  max = 1000000;
  maxV = 1000000;
var
  a, t: array[0..max + 1] of Integer;
  s: array[1..max + 2] of Integer;
  n, m: Integer;

procedure Enter; //Nhập dữ liệu
var
  i: Integer;
begin
  ReadLn(n);
  for i := 1 to n do Read(a[i]);
end;

procedure Init; //Khởi tạo
begin
  a[0] := -maxV - 1; a[n + 1] := maxV + 1; //Thêm vào 2 phần tử
  m := 1; //Độ dài dãy con tăng dài nhất bắt đầu từ a[n + 1]
  s[1] := n + 1; //Tính s[\lambda] với \lambda = 1
end;

//Thuật toán tìm kiếm nhị phân nhận vào một phần tử a[i] = v,
//và tìm trong dãy a[s[1]] > a[s[2]] > ... > a[s[m]], trả về chỉ số \lambda lớn nhất thoả mãn a[s[\lambda]] > v
function BinarySearch(v: Integer): Integer;
var
  Low, Middle, High: Integer;

```

```

begin
  Low := 2; High := m;
  while Low <= High do
    begin
      Middle := (Low + High) div 2;
      if a[s[Middle]] > v then Low := Middle + 1
      else High := Middle - 1;
    end;
  Result := High;
end;

procedure Optimize; //Quy hoạch động
var
  i, lambda: Integer;
begin
  for i := n downto 0 do //Giải công thức truy hồi
    begin
      lambda := BinarySearch(a[i]);
      t[i] := s[lambda]; //Lưu vết
      if lambda + 1 > m then //Cập nhật m là độ dài dãy con tăng dài nhất cho tới thời điểm này
        m := lambda + 1;
      s[lambda + 1] := i;
    end;
  end;

procedure PrintResult; //In kết quả
var
  i: Integer;
begin
  WriteLn('Length = ', m - 2); //Dãy kết quả phải bỏ đi hai phần tử a[0] và a[n + 1]
  i := t[0]; //Truy vết bắt đầu từ t[0]
  while i <> n + 1 do
    begin
      WriteLn('a[', i, '] = ', a[i]);
      i := t[i]; //Xét phần tử kế tiếp
    end;
  end;

begin
  Enter;
  Init;
  Optimize;
  PrintResult;
end.

```

3.3.2. Bài toán xếp ba lô

Cho n sản phẩm, sản phẩm thứ i có trọng lượng là w_i và giá trị là v_i ($w_i, v_i \in \mathbb{Z}^+$). Cho một balô có giới hạn trọng lượng là m , hãy chọn ra một số sản phẩm cho vào balô sao cho tổng trọng lượng của chúng không vượt quá m và tổng giá trị của chúng là lớn nhất có thể.

Input

- Dòng 1 chứa hai số nguyên dương n, m ($n \leq 1000; m \leq 10000$)
- n dòng tiếp theo, dòng thứ i chứa hai số nguyên dương w_i, v_i cách nhau ít nhất một dấu cách ($w_i, v_i \leq 10000$)

Output

Phương án chọn các sản phẩm có tổng trọng lượng $\leq m$ và tổng giá trị lớn nhất có thể.

Sample Input	Sample Output
5 11	Selected objects:
3 30	1. Object 5: Weight = 4; value = 40
4 40	2. Object 2: Weight = 4; value = 40
5 40	3. Object 1: Weight = 3; value = 30
9 100	Total weight: 11
4 40	Total value : 110

□ Thuật toán

Bài toán Knapsack tuy chưa có thuật toán giải hiệu quả trong trường hợp tổng quát nhưng với ràng buộc dữ liệu cụ thể như ở bài này thì có thuật toán quy hoạch động khá hiệu quả để giải, dưới đây chúng ta sẽ trình bày cách làm đó.

Gọi $f[i, j]$ là giá trị lớn nhất có thể có bằng cách chọn trong các sản phẩm $\{1, 2, \dots, i\}$ với giới hạn trọng lượng j . Mục đích của chúng ta là đi tìm $f[n, m]$: Giá trị lớn nhất có thể có bằng cách chọn trong n sản phẩm đã cho với giới hạn trọng lượng m .

Với giới hạn trọng lượng j , việc chọn trong số các sản phẩm $\{1, 2, \dots, i\}$ để có giá trị lớn nhất sẽ là một trong hai khả năng:

- Nếu không chọn sản phẩm thứ i , thì $f[i, j]$ là giá trị lớn nhất có thể có bằng cách chọn trong số các sản phẩm $\{1, 2, \dots, i - 1\}$ với giới hạn trọng lượng bằng j . Tức là $f[i, j] = f[i - 1, j]$
- Nếu có chọn sản phẩm thứ i (tất nhiên chỉ xét tới trường hợp này khi mà $w_i \leq j$), thì $f[i, j]$ bằng giá trị sản phẩm thứ i ($= v_i$) cộng với giá trị lớn nhất có thể có được bằng cách chọn trong số các sản phẩm $\{1, 2, \dots, i - 1\}$ với giới hạn trọng lượng $j - w_i$ ($= f[i - 1, j - w_i]$). Tức là về mặt giá trị thu được, $f[i, j] = f[i - 1, j - w_i] + v_i$.

Theo cách xây dựng $f[i, j]$, ta suy ra công thức truy hồi:

$$f[i, j] = \begin{cases} f[i - 1, j], & \text{nếu } j < w_i \\ \max\{f[i - 1, j], f[i - 1, j - w_i] + v_i\}, & \text{nếu } j \geq w_i \end{cases} \quad (3.7)$$

Từ công thức truy hồi, ta thấy rằng để tính các phần tử trên hàng i của bảng f thì ta cần phải biết các phần tử trên hàng liền trước đó (hàng $i - 1$). Vậy nếu ta biết được hàng 0 của bảng f thì có thể tính ra mọi phần tử trong bảng. Từ đó suy ra cơ sở quy hoạch động: $f[0, j]$ theo định nghĩa là giá trị lớn nhất có thể bằng cách chọn trong số 0 sản phẩm, nên hiển nhiên $f[0, j] = 0$.

Tính xong bảng phương án thì ta quan tâm đến $f[n, m]$, đó chính là giá trị lớn nhất thu được khi chọn trong cả n sản phẩm với giới hạn trọng lượng m . Việc còn lại là chỉ ra phương án chọn các sản phẩm.

- Nếu $f[n, m] = f[n - 1, m]$ thì tức là phương án tối ưu không chọn sản phẩm n , ta truy tiếp $f[n - 1, m]$.

- Còn nếu $f[n, m] \neq f[n - 1, m]$ thì ta thông báo rằng phép chọn tối ưu có chọn sản phẩm n và truy tiếp $f[n - 1, m - w_n]$.

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

□ Cài đặt

Rõ ràng ta có thể giải công thức truy hồi theo kiểu từ dưới lên: Khởi tạo hàng 0 của bảng phương án f toàn số 0, sau đó dùng công thức truy hồi tính lần lượt từ hàng 1 tới hàng n . Tuy nhiên trong chương trình này chúng ta sẽ giải công thức truy hồi theo kiểu từ trên xuống (sử dụng hàm đệ quy kết hợp với kỹ thuật lưu trữ nghiệm) vì cách giải này không phải đi tính toàn bộ bảng phương án.

KNAPSACK_DP.PAS ✓ Bài toán xếp ba lô

```
{ $MODE OBJFPC }
program Knapsack;
const
  maxN = 1000;
  maxM = 10000;
var
  w, v: array[1..maxN] of Integer;
  f: array[0..maxN, 0..maxM] of Integer;
  n, m: Integer;
  SumW, SumV: Integer;

procedure Enter; //Nhập dữ liệu
var
  i: Integer;
begin
  ReadLn(n, m);
  for i := 1 to n do ReadLn(w[i], v[i]);
end;

function Getf(i, j: Integer): Integer; //Tính f[i, j]
begin
  if f[i, j] = -1 then //Nếu f[i, j] chưa được tính thì mới đi tính
    if i = 0 then f[i, j] := 0 //Cơ sở quy hoạch động
    else
      if (j < w[i]) or
        (Getf(i - 1, j) > Getf(i - 1, j - w[i]) + v[i]) then
        f[i, j] := Getf(i - 1, j) //Không chọn sản phẩm i thì tốt hơn
      else
        f[i, j] := Getf(i - 1, j - w[i]) + v[i]; //Chọn sản phẩm i thì tốt hơn
  Result := f[i, j]; //Trả về f[i, j] trong kết quả hàm
end;

procedure PrintResult; //In kết quả
var
  Count: Integer;
begin
  WriteLn('Selected objects:');
  Count := 0;
  SumW := 0;
```

```

while n <> 0 do //Truy vết từ f[n, m]
begin
  if Getf(n, m) <> Getf(n - 1, m) then //Phương án tối ưu có chọn sản phẩm n
  begin
    Inc(Count);
    Write(Count, '. Object ', n, ': ');
    WriteLn('Weight = ', w[n], '; value = ', v[n]);
    Inc(SumW, w[n]);
    Dec(m, w[n]); //Chọn sản phẩm n rồi thì giới hạn trọng lượng giảm đi w[n]
  end;
  Dec(n);
end;
WriteLn('Total weight: ', SumW);
WriteLn('Total value : ', SumV);
end;

begin
  Enter;
  FillByte(f, SizeOf(f), $FF); //Khởi tạo các phần tử của f là -1 (chưa được tính)
  SumV := Getf(n, m);
  PrintResult;
end.

```

Nếu thêm vào một đoạn chương trình để in ra bảng f sau khi giải công thức truy hồi thì với dữ liệu như trong ví dụ, bảng f sẽ là:

f	0	1	2	3	4	5	6	7	8	9	10	11
0	0	-1	0	0	0	-1	0	0	0	-1	-1	0
1	-1	-1	0	30	-1	-1	30	30	-1	-1	-1	30
2	-1	-1	0	-1	-1	-1	40	70	-1	-1	-1	70
3	-1	-1	0	-1	-1	-1	-1	70	-1	-1	-1	80
4	-1	-1	-1	-1	-1	-1	-1	70	-1	-1	-1	100
5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	110

Ta thấy còn rất nhiều giá trị -1 trong bảng, nghĩa là có rất nhiều phần tử trong bảng f không tham gia vào quá trình tính $f[5,11]$. Điều đó chỉ ra rằng cách giải công thức truy hồi từ trên xuống trong trường hợp này sẽ hiệu quả hơn cách giải từ dưới lên vì nó không cần tính toàn bộ bảng phương án của quy hoạch động.

3.3.3. Biến đổi xâu

Với xâu ký tự $X = x_1x_2\dots x_m$, xét 3 phép biến đổi:

- $Insert(i, c)$: Chèn ký tự c vào sau vị trí i của xâu X .
- $Delete(i)$: Xoá ký tự tại vị trí i của xâu X .
- $Replace(i, c)$: Thay ký tự tại vị trí i của xâu X bởi ký tự c .

Yêu cầu: Cho hai xâu ký tự $X = x_1x_2\dots x_m$ và $Y = y_1y_2\dots y_n$ ($m, n \leq 1000$) chỉ gồm các chữ cái in hoa, hãy tìm một số ít nhất các phép biến đổi kể trên để biến xâu X thành xâu Y .

Input

- Dòng 1 chứa xâu X

- Dòng 2 chứa chuỗi Y

Output

Cách biến đổi X thành Y

Sample Input	Sample Output
ACGGTAG CCTAAG	Number of alignments: 4 X = ACGGTAG Insert(5, A) X = ACGGTAAG Delete(4) X = ACGTAAG Delete(3) X = ACTAAG Replace(1, C) X = CCTAAG

Bài toán này (*String alignment**) có ứng dụng khá nhiều trong Tin-sinh học để phân tích và so sánh các yếu tố di truyền (DNA, RNA, proteins, chromosomes, genes, ...). Dữ liệu di truyền thường có dạng chuỗi, chẳng hạn một phân tử DNA có thể biểu diễn dưới dạng chuỗi các ký tự $\{A, C, G, T\}$ hay một phân tử RNA có thể biểu diễn bằng chuỗi các ký tự $\{A, C, G, U\}$. Để trả lời câu hỏi hai mẫu dữ liệu di truyền (mã hoá dưới dạng chuỗi ký tự) giống nhau đến mức nào, người ta có thể dùng số lượng các phép biến đổi kể trên để đo mức độ khác biệt giữa hai mẫu. Tùy vào từng yêu cầu thực tế, người ta còn có thể thêm vào nhiều phép biến đổi nữa, hoặc gán trọng số cho từng vị trí biến đổi (vì các vị trí trong chuỗi có thể có tầm quan trọng khác nhau).

Có thể kể ra vài ứng dụng cụ thể của bài toán giống hệt. Một ứng dụng là xác định chủng loại của một virus chưa biết. Chẳng hạn khi phát hiện một virus mới gây bệnh mà con người chưa kịp hình thành khả năng miễn dịch, người ta sẽ đối sánh mẫu RNA của virus đó với tất cả các mẫu RNA của các virus đã có trong cơ sở dữ liệu, từ đó tìm ra các loại virus tương tự, xác định chủng loại của virus và điều chế vaccine phòng bệnh. Một ứng dụng khác là phân tích các chuỗi DNA để xác định mối liên quan giữa các loài, từ đó xây dựng cây tiến hoá từ một loài tổ tiên nào đó. Ngoài ứng dụng trong sinh học phân tử, bài toán giống hệt còn có nhiều ứng dụng khác trong khoa học máy tính, lý thuyết mã, nhận dạng tiếng nói, v.v...

□ Thuật toán

Ta trở lại bài toán với hai chuỗi ký tự $X = x_1x_2\dots x_m$ và $Y = y_1y_2\dots y_n$. Để biến đổi chuỗi X thành chuỗi Y , chúng ta sẽ phân tích một bài toán con: Tìm một số ít nhất các phép biến đổi để biến i ký tự đầu tiên của chuỗi X thành j ký tự đầu tiên của chuỗi Y . Cụ thể, ta gọi $f[i, j]$ là số phép biến đổi tối thiểu để biến chuỗi $x_1x_2\dots x_i$ thành chuỗi $y_1y_2\dots y_j$. Khi đó:

* Tạm dịch: Giống hệt

- Nếu $x_i = y_j$ thì vấn đề trở thành biến xâu $x_1x_2 \dots x_{i-1}$ thành xâu $y_1y_2 \dots y_{j-1}$. Tức là nếu xét về số lượng các phép biến đổi, trong trường hợp này $f[i, j] = f[i - 1, j - 1]$.
- Nếu $x_i \neq y_j$. Ta có ba lựa chọn:
 - Hoặc chèn ký tự y_j vào cuối xâu $x_1x_2 \dots x_i$ rồi biến xâu $x_1x_2 \dots x_i$ thành xâu $y_1y_2 \dots y_{j-1}$. Tức là nếu theo sự lựa chọn này, $f[i, j] = f[i, j - 1] + 1$
 - Hoặc xóa ký tự cuối của xâu $x_1x_2 \dots x_i$ rồi biến xâu $x_1x_2 \dots x_{i-1}$ thành xâu $y_1y_2 \dots y_j$. Tức là nếu theo sự lựa chọn này, $f[i, j] = f[i - 1, j] + 1$
 - Hoặc thay ký tự cuối của xâu $x_1x_2 \dots x_i$ bởi ký tự y_j rồi biến xâu $x_1x_2 \dots x_{i-1}$ thành xâu $y_1y_2 \dots y_{j-1}$. Tức là nếu theo sự lựa chọn này, $f[i, j] = f[i - 1, j - 1] + 1$

Vì chúng ta cần $f[i, j] \rightarrow \min$ nên suy ra công thức truy hồi:

$$f[i, j] = \begin{cases} f[i - 1, j - 1], & \text{nếu } x_i = y_j \\ 1 + \min \begin{cases} f[i, j - 1] \\ f[i - 1, j] \\ f[i - 1, j - 1] \end{cases}, & \text{nếu } x_i \neq y_j \end{cases} \quad (3.8)$$

Từ công thức truy hồi, ta thấy rằng nếu biểu diễn f dưới dạng ma trận thì phần tử ở ô (i, j) ($f[i, j]$) được tính dựa vào phần tử ở ô nằm bên trái: $(i, j - 1)$, ô nằm phía trên: $(i - 1, j)$ và ô nằm ở góc trái trên $(i - 1, j - 1)$. Từ đó suy ra tập các bài toán cơ sở:

- $f[0, j]$ là số phép biến đổi biến xâu rỗng thành j ký tự đầu của xâu Y , nó cần tối thiểu j phép chèn: $f[0, j] = j$
- $f[i, 0]$ là số phép biến đổi biến xâu gồm i ký tự đầu của X thành xâu rỗng, nó cần tối thiểu i phép xóa: $f[i, 0] = i$

Vậy đầu tiên bảng phương án $f[0 \dots m, 0 \dots n]$ được khởi tạo hàng 0 và cột 0 là cơ sở quy hoạch động. Từ đó dùng công thức truy hồi tính ra tất cả các phần tử bảng f , sau khi tính xong thì $f[m, n]$ cho ta biết số phép biến đổi tối thiểu để biến xâu $X = x_1x_2 \dots x_m$ thành $Y = y_1y_2 \dots y_n$. Việc cuối cùng là chỉ ra cụ thể cách biến đổi tối ưu, ta bắt đầu truy vết từ $f[m, n]$, việc truy vết chính là dò ngược lại xem $f[m, n]$ được tìm như thế nào.

- Nếu $x_m = y_n$ thì ta tìm cách biến $x_1x_2 \dots x_{m-1}$ thành $y_1y_2 \dots y_{n-1}$, tức là truy tiếp $f[m - 1, n - 1]$
- Nếu $x_m \neq y_n$ thì xét ba trường hợp
 - Nếu $f[m, n] = f[m, n - 1] + 1$ thì ta thực hiện phép *Insert*(m, y_n) để chèn y_n vào cuối xâu X rồi tìm cách biến đổi $x_1x_2 \dots x_m$ thành $y_1y_2 \dots y_{n-1}$ (truy tiếp $f[m, n - 1]$)
 - Nếu $f[m, n] = f[m - 1, n] + 1$ thì ta thực hiện phép *Delete*(m) để xóa ký tự cuối xâu X rồi tìm cách biến đổi $x_1x_2 \dots x_{m-1}$ thành $y_1y_2 \dots y_n$ (truy tiếp $f[m - 1, n]$)
 - Nếu $f[m, n] = f[m - 1, n - 1] + 1$ thì ta thực hiện phép *Replace*(m, y_n) để thay ký tự cuối xâu X bởi y_n rồi tìm cách biến đổi $x_1x_2 \dots x_{m-1}$ thành $y_1y_2 \dots y_{n-1}$ (truy tiếp $f[m - 1, n - 1]$)

Ta đưa về việc truy vết với m, n nhỏ hơn và cứ lặp lại cho tới khi quy về bài toán cơ sở: $m = 0$ hoặc $n = 0$.

Ví dụ với $X = \text{'ACGGTAG'}$ và $Y = \text{'CCTAAG'}$, việc truy vết trên bảng phương án f được chỉ ra trong Hình 3-5.

		C C T A A G						
f		0	1	2	3	4	5	6
0		0	1	2	3	4	5	6
A	1	1	1	2	3	3	4	5
C	2	2	1	1	2	3	4	5
G	3	3	2	2	2	3	4	4
G	4	4	3	3	3	3	4	4
T	5	5	4	4	3	4	4	5
A	6	6	5	5	4	3	4	5
G	7	7	6	6	5	4	4	4

Hình 3-5. Truy vết tìm cách biến đổi xâu

❑ Cài đặt

📄 STRINGALIGNMENT_DP.PAS ✓ Biến đổi xâu

```
{ $MODE OBJFPC }
program StringAlignment;
const
  max = 1000;
type
  TStrOperator = (soDoNothing, soInsert, soDelete, soReplace);
var
  x, y: AnsiString;
  f: array[0..max, 0..max] of Integer;
  m, n: Integer;

procedure Enter; //Nhập dữ liệu
begin
  ReadLn(x);
  ReadLn(y);
  m := Length(x); n := Length(y);
end;

function Min3(x, y, z: Integer): Integer; //Tính min của 3 số
begin
  if x < y then Result := x else Result := y;
  if z < Result then Result := z;
end;

procedure Optimize; //Giải công thức truy hồi
var
  i, j: Integer;
begin
  //Lưu cơ sở quy hoạch động
  for j := 0 to n do f[0, j] := j;
  for i := 1 to m do f[i, 0] := i;
  //Dùng công thức truy hồi tính toàn bộ bảng phương án
```



```

for i := 1 to m do
  for j := 1 to n do
    if x[i] = y[j] then
      f[i, j] := f[i - 1, j - 1]
    else
      f[i, j] := Min3(f[i, j - 1], f[i - 1, j - 1], f[i - 1, j]) + 1;
    end;
  end;
end;

```

//Thực hiện thao tác op với tham số vị trí p và ký tự c trên chuỗi X

```

procedure Perform(op: TStrOperator; p: Integer = 0; c: Char = #0);

```

```

begin

```

```

  case op of

```

```

    soInsert: //Phép chèn

```

```

      begin

```

```

        WriteLn('Insert(', p, ', ', c, ')');

```

```

        Insert(c, x, p + 1);

```

```

      end;

```

```

    soDelete: //Phép xoá

```

```

      begin

```

```

        WriteLn('Delete(', p, ')');

```

```

        Delete(x, p, 1);

```

```

      end;

```

```

    soReplace: //Phép thay

```

```

      begin

```

```

        WriteLn('Replace(', p, ', ', c, ')');

```

```

        x[p] := c;

```

```

      end;

```

```

  end;

```

```

  WriteLn('X = ', x); //In ra chuỗi X sau khi biến đổi

```

```

end;

```

```

procedure PrintResult; //In kết quả

```

```

begin

```

```

  WriteLn('Number of alignments: ', f[m, n]);

```

```

  Perform(soDoNothing); //In ra chuỗi X ban đầu

```

```

  while (m <> 0) and (n <> 0) do //Làm tới khi m = n = 0

```

```

    if x[m] = y[n] then //Hai ký tự cuối giống nhau, không làm gì cả, truy tiếp f[m - 1, n - 1]

```

```

      begin

```

```

        Dec(m); Dec(n);

```

```

      end

```

```

    else //Tại vị trí m của chuỗi X cần có một phép biến đổi

```

```

      if f[m, n] = f[m, n - 1] + 1 then //Nếu là phép chèn

```

```

        begin

```

```

          Perform(soInsert, m, y[n]);

```

```

          Dec(n); //Truy sang phải: f[m, n - 1]

```

```

        end

```

```

      else

```

```

        if f[m, n] = f[m - 1, n] + 1 then //Nếu là phép xoá

```

```

          begin

```

```

            Perform(soDelete, m);

```

```

            Dec(m); //Truy lên trên: f[m - 1, n]

```

```

          end

```

```

        else //Không phải chèn, không phải xoá thì phải là phép thay thế

```

```

          begin

```

```

            Perform(soReplace, m, y[n]);

```

```

            Dec(m); Dec(n); //Truy chéo lên trên: f[m - 1, n - 1]

```

```

        end;
while m > 0 do
    begin
        Perform(soDelete, m);
        Dec(m);
    end;
while n > 0 do
    begin
        Perform(soInsert, 0, y[n]);
        Dec(n);
    end;
end;

begin
    Enter;
    Optimize;
    PrintResult;
end.

```

3.3.4. Phân hoạch tam giác

Trên mặt phẳng với hệ tọa độ trục chuẩn Oxy , cho một đa giác lồi $P_1P_2\dots P_n$, trong đó tọa độ đỉnh P_i là (x_i, y_i) . Một bộ $n - 3$ đường chéo đôi một không cắt nhau sẽ chia đa giác đã cho thành $n - 2$ tam giác, ta gọi bộ $n - 3$ đường chéo đó là một phép tam giác phân của đa giác lồi ban đầu.

Bài toán đặt ra là hãy tìm một phép tam giác phân có trọng số (tổng độ dài các đường chéo) nhỏ nhất.

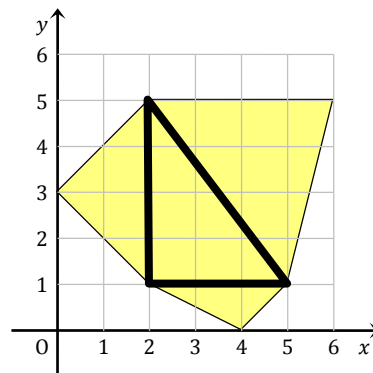
Input

- Dòng 1 chứa số n là số đỉnh của đa giác ($3 \leq n \leq 200$)
- n dòng tiếp theo, dòng thứ i chứa tọa độ của đỉnh P_i : gồm hai số thực x_i, y_i

Output

Phép tam giác phân nhỏ nhất.

Sample Input	Sample Output
6	Minimum triangulation: 12.00
4.0 0.0	P[2] - P[6] = 3.00
5.0 1.0	P[2] - P[4] = 5.00
6.0 5.0	P[4] - P[6] = 4.00
2.0 5.0	
0.0 3.0	
2.0 1.0	



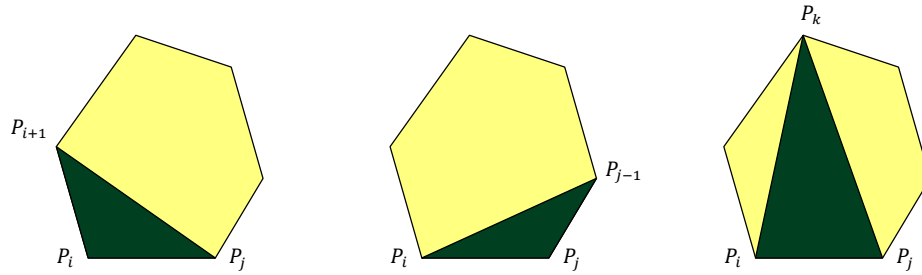
□ Thuật toán

Ký hiệu $d(i, j)$ là khoảng cách giữa hai điểm P_i và P_j :

$$d(i, j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \quad (3.9)$$

Gọi $f[i, j]$ là trọng số phép tam giác phân nhỏ nhất để chia đa giác $P_i P_{i+1} \dots P_j$, khi đó $f[1, n]$ chính là trọng số phép tam giác phân nhỏ nhất để chia đa giác ban đầu.

Bằng quan sát hình học của một phép tam giác phân trên đa giác $P_i P_{i+1} \dots P_j$, ta nhận thấy rằng trong bất kỳ phép tam giác phân nào, luôn tồn tại một và chỉ một tam giác chứa cạnh $P_i P_j$. Tam giác chứa cạnh $P_i P_j$ sẽ có một đỉnh là P_i , một đỉnh là P_j và một đỉnh P_k nào đó ($i < k < j$). Xét một phép tam giác phân nhỏ nhất để chia đa giác $P_i P_{i+1} \dots P_j$, ta quan tâm tới tam giác chứa cạnh $P_i P_j$: $\Delta P_i P_j P_k$, có ba khả năng xảy ra (Hình 3-6):



Hình 3-6. Ba trường hợp của tam giác chứa cạnh $P_i P_j$

Khả năng thứ nhất, đỉnh $P_k \equiv P_{i+1}$, khi đó đường chéo $P_{i+1} P_j$ sẽ thuộc phép tam giác phân đang xét, những đường chéo còn lại sẽ phải tạo thành một phép tam giác phân nhỏ nhất để chia đa giác lồi $P_{i+1} P_{i+2} \dots P_j$. Tức là trong trường hợp này:

$$f[i, j] = f^{(k=i+1)}[i, j] = d(i+1, j) + f[i+1, j] \quad (3.10)$$

Khả năng thứ hai, đỉnh $P_k \equiv P_{j-1}$, khi đó đường chéo $P_i P_{j-1}$ sẽ thuộc phép tam giác phân đang xét, những đường chéo còn lại sẽ phải tạo thành một phép tam giác phân nhỏ nhất để chia đa giác lồi $P_i P_{i+1} \dots P_{j-1}$. Tức là trong trường hợp này:

$$f[i, j] = f^{(k=j-1)}[i, j] = d(i, j-1) + f[i, j-1] \quad (3.11)$$

Khả năng thứ ba, đỉnh P_k không trùng P_{i+1} và cũng không trùng P_{j-1} , khi đó cả hai đường chéo $P_i P_k$ và $P_j P_k$ đều thuộc phép tam giác phân, những đường chéo còn lại sẽ tạo thành hai phép tam giác phân nhỏ nhất tương ứng với đa giác $P_i P_{i+1} \dots P_k$ và đa giác $P_k P_{k+1} \dots P_j$. Tức là trong trường hợp này:

$$f[i, j] = f^{(k)}[i, j] = d(i, k) + d(k, j) + f[i, k] + f[k, j] \quad (3.12)$$

Vì $f[i, j]$ là trọng số của phép tam giác phân nhỏ nhất trên đa giác $P_i P_{i+1} \dots P_j$ nên nó sẽ phải là trọng số nhỏ nhất của phép tam giác phân trên tất cả các khả năng chọn đỉnh P_k tức là:

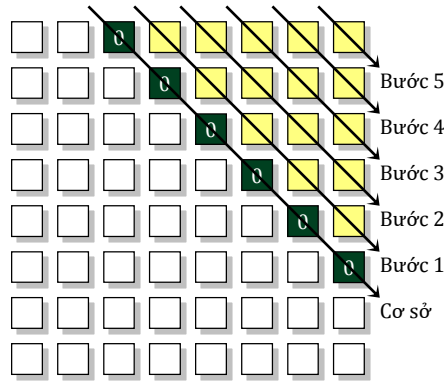
$$f[i, j] = \min_{k:i < k < j} \{f^{(k)}[i, j]\} \quad (3.13)$$

Trong đó

$$f^{(k)}[i, j] = \begin{cases} d(i+1, j) + f[i+1, j]; & \text{nếu } k = i+1 \\ d(i+1, j) + f[i, j-1]; & \text{nếu } k = j-1 \\ d(i, k) + d(k, j) + f[i, k] + f[k, j]; & \text{nếu } i+1 < k < j-1 \end{cases} \quad (3.14)$$

Ta xây dựng xong công thức truy hồi tính $f[i, j]$. Từ công thức truy hồi, ta nhận thấy rằng $f[i, j]$ được tính qua các giá trị $f[i, k]$ và $f[k, j]$ với $i < k < j$, điều đó chỉ ra rằng hiệu số $j - i$ chính là thứ tự lớn-nhỏ của các bài toán: Các giá trị $f[i, j]$ với $j - i$ lớn sẽ được tính sau các giá trị $f[i', j']$ có hiệu số $j' - i'$ nhỏ hơn. Thứ tự lớn-nhỏ này cho phép tìm ra cơ sở quy hoạch động: Các giá trị $f[i, i+2]$ là trọng số của phép tam giác phân nhỏ nhất để chia đa giác lõm gồm 3 đỉnh (tức là tam giác), trong trường hợp này chúng ta không cần chia gì cả và trọng số phép tam giác phân là $f[i, i+2] = 0$.

Việc giải công thức truy hồi sẽ được thực hiện theo trình tự: Đầu tiên bảng phương án f được điền cơ sở quy hoạch động: Các phần tử trên đường chéo $f[i, i+2]$ được đặt bằng 0, dùng công thức truy hồi tính tiếp các phần tử trên đường chéo $f[i, i+3]$, rồi tính tiếp đường chéo $f[i, i+4]$... Cứ như vậy cho tới khi tính được phần tử $f[1, n]$.



Hình 3-7. Quy trình tính bảng phương án

Song song với quá trình giải công thức truy hồi, tại mỗi bước tính $f[i, j] := \min_{k:i < k < j} \{f^{(k)}[i, j]\}$, ta lưu lại $trace[i, j]$ là chỉ số k mà tại đó $f[i, j]$ đạt cực tiểu, tức là:

$$trace[i, j] := \arg \min_{k:i < k < j} \{f^{(k)}[i, j]\} \quad (3.15)$$

Sau khi quá trình tính toán kết thúc, để in ra phép tam giác phân nhỏ nhất đối với đa giác $P_i P_{i+1} \dots P_j$, ta sẽ dựa vào phần tử $k = trace[i, j]$ để đưa vấn đề về việc in ra phép tam giác

phân đối với hai đa giác $P_i P_{i+1} \dots P_k$ và $P_k P_{k+1} \dots P_j$. Điều này có thể được thực hiện đơn giản bằng một thủ tục đệ quy.

□ Cài đặt

TRIANGULATION_DP.PAS ✓ Phân hoạch tam giác

```
{ $MODE OBJFPC }
program Triangulation;
const
  max = 200;
var
  x, y: array[1..max] of Real;
  f: array[1..max, 1..max] of Real;
  trace: array[1..max, 1..max] of Integer;
  n: Integer;

procedure Enter; //Nhập dữ liệu
var
  i: Integer;
begin
  ReadLn(n);
  for i := 1 to n do ReadLn(x[i], y[i]);
end;

function d(i, j: Integer): Real; //Hàm đo khoảng cách P[i] - P[j]
begin
  Result := Sqrt(Sqr(x[i] - x[j]) + Sqr(y[i] - y[j]));
end;

//Hàm cực tiểu hoá target := Min(target, value)
function Minimize(var target: Real; value: Real): Boolean;
begin
  Result := value < target; //Trả về True nếu target được cập nhật
  if Result then target := value;
end;

procedure Optimize; //Giải công thức truy hồi
var
  m, i, j, k: Integer;
begin
  for i := 1 to n - 2 do
    f[i, i + 2] := 0;
    for m := 3 to n - 1 do
      for i := 1 to n - m do
        begin //Tính f[i, i+m]
          j := i + m;
          //Trường hợp k = i+1
          f[i, j] := d(i + 1, j) + f[i + 1, j];
          trace[i, j] := i + 1;
          //Trường hợp i+1 < k < j-1
          for k := i + 2 to j - 2 do
            if Minimize(f[i, j],
              d(i, k) + d(k, j) + f[i, k] + f[k, j]) then
              trace[i, j] := k;
          //Trường hợp k = j-1
```

```

        if Minimize(f[i, j], d(i, j - 1) + f[i, j - 1]) then
            trace[i, j] := j - 1;
        end;
    end;

//Với i + 1 < j, thủ tục này in ra đường chéo P[i] - P[j]
procedure WriteDiagonal(i, j: Integer);
begin
    if i + 1 < j then
        WriteLn('P[' , i , ' ] - P[' , j , ' ] = ' , d(i, j):0:2);
    end;

//Truy vết bằng đệ quy, in ra cách tam giác phân đa giác P[i..j]
procedure TraceRecursively(i, j: Integer);
var
    k: Integer;
begin
    if j <= i + 2 then Exit; //Đa giác có ≤ 3 đỉnh thì không cần phân hoạch
    k := trace[i, j]; //Lấy vết
    WriteDiagonal(i, k);
    WriteDiagonal(k, j);
    TraceRecursively(i, k); //In ra cách phân hoạch đa giác P[i..k]
    TraceRecursively(k, j); //In ra cách phân hoạch đa giác P[k..j]
end;

procedure PrintResult; //In kết quả
begin
    WriteLn('Minimum triangulation: ' , f[1, n]:0:2);
    TraceRecursively(1, n);
end;

begin
    Enter;
    Optimize;
    PrintResult;
end.

```

3.3.5. Phép nhân dãy ma trận

Với ma trận $A = \{a_{ij}\}$ kích thước $p \times q$ và ma trận $B = \{b_{jk}\}$ kích thước $q \times r$. Người ta có phép nhân hai ma trận đó để được ma trận $C = \{c_{ik}\}$ kích thước $p \times r$. Mỗi phần tử của ma trận C được tính theo công thức:

$$c_{ik} = \sum_{j=1}^q a_{ij} \times b_{jk}; (1 \leq i \leq p; 1 \leq k \leq r) \quad (3.16)$$

Ví dụ với A là ma trận kích thước 3×4 , B là ma trận kích thước 4×5 thì C sẽ là ma trận kích thước 3×5 (Hình 3-8).

				B																																
				<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>1</td><td>0</td><td>2</td><td style="background-color: yellow;">4</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td><td style="background-color: yellow;">5</td><td>1</td></tr> <tr><td>3</td><td>0</td><td>1</td><td style="background-color: yellow;">6</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td style="background-color: yellow;">1</td><td>1</td></tr> </table>	1	0	2	4	0	0	1	0	5	1	3	0	1	6	1	1	1	1	1	1												
1	0	2	4	0																																
0	1	0	5	1																																
3	0	1	6	1																																
1	1	1	1	1																																
A					C																															
<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td style="background-color: yellow;">5</td><td style="background-color: yellow;">6</td><td style="background-color: yellow;">7</td><td style="background-color: yellow;">8</td></tr> <tr><td>9</td><td>10</td><td>11</td><td>12</td></tr> </table>	1	2	3	4	5	6	7	8	9	10	11	12					<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>14</td><td>6</td><td>9</td><td style="background-color: yellow;">36</td><td>9</td></tr> <tr><td style="background-color: yellow;">34</td><td style="background-color: yellow;">14</td><td style="background-color: yellow;">25</td><td style="background-color: green;">100</td><td style="background-color: yellow;">21</td></tr> <tr><td>54</td><td>22</td><td>41</td><td style="background-color: yellow;">164</td><td>33</td></tr> </table>	14	6	9	36	9	34	14	25	100	21	54	22	41	164	33				
1	2	3	4																																	
5	6	7	8																																	
9	10	11	12																																	
14	6	9	36	9																																
34	14	25	100	21																																
54	22	41	164	33																																

Hình 3-8. Phép nhân hai ma trận

Ta xét thuật toán để nhân hai ma trận $A(p \times q)$ và $B(q \times r)$:

```

for i := 1 to p do
  for j := 1 to r do
    begin
      c[i, j] := 0;
      for k := 1 to q do
        c[i, j] := c[i, j] + a[i, k] * b[k, j];
      end;
    end;

```

Phí tổn để thực hiện phép nhân ma trận có thể đánh giá qua số lần thực hiện phép nhân số học, để nhân hai ma trận $A(p \times q)$ và $B(q \times r)$ chúng ta cần $p \times q \times r$ phép nhân*.

Phép nhân ma trận không có tính chất giao hoán nhưng có tính chất kết hợp:

$$(AB)C = A(BC) \tag{3.17}$$

Vậy nếu A là ma trận cấp 3×4 , B là ma trận cấp 4×6 và C là ma trận cấp 6×8 thì:

- Để tính $(AB)C$, phép tính (AB) cho ma trận kích thước 3×6 sau $3 \times 4 \times 6 = 72$ phép nhân số học, sau đó nhân tiếp với C được ma trận kết quả kích thước 3×8 sau $3 \times 6 \times 8 = 144$ phép nhân số học. Vậy tổng số phép nhân số học phải thực hiện sẽ là 216.
- Để tính $A(BC)$, phép tính (BC) cho ma trận kích thước 4×8 sau $4 \times 6 \times 8 = 192$ phép nhân số học, lấy A nhân với ma trận này được ma trận kết quả kích thước 3×8 sau $3 \times 4 \times 8 = 96$ phép nhân số học. Vậy tổng số phép nhân số học phải thực hiện sẽ là 288.

* Để nhân hai ma trận, có một số thuật toán tốt hơn, chẳng hạn thuật toán Strassen $O(n^{\lg 7})$ hay thuật toán Coppersmith-Winograd $O(n^{2.376})$. Nhưng để đơn giản cho bài toán này, chúng ta dùng thuật toán nhân ma trận đơn giản nhất.

Ví dụ này cho chúng ta thấy rằng trình tự thực hiện có ảnh hưởng lớn tới chi phí. Vấn đề đặt ra là tính số phí tổn ít nhất khi thực hiện phép nhân một dãy các ma trận:

$$\prod_{i=1}^n M_i = M_1 M_2 \dots M_n$$

Trong đó:

M_1 là ma trận kích thước $a_0 \times a_1$

M_2 là ma trận kích thước $a_1 \times a_2$

...

M_n là ma trận kích thước $a_{n-1} \times a_n$

Input

- Dòng 1 chứa số nguyên dương $n \leq 100$
- Dòng 2 chứa $n + 1$ số nguyên dương a_0, a_1, \dots, a_n cách nhau ít nhất một dấu cách ($\forall i: a_i \leq 1000$)

Output

Cho biết phương án nhân ma trận tối ưu và số phép nhân phải thực hiện

Sample Input	Sample Output
6 9 5 3 2 4 7 8	The best solution: (M[1].(M[2].M[3])).((M[4].M[5]).M[6])) Number of numerical multiplications: 432

□ Thuật toán

Cách giải của bài toán này gần giống như bài toán tam giác phân: Trước hết, nếu dãy chỉ có một ma trận thì chi phí bằng 0, tiếp theo, chi phí để nhân một cặp ma trận có thể tính được ngay. Bằng việc ghi nhận chi phí của phép nhân hai ma trận liên tiếp ta có thể sử dụng những thông tin đó để tối ưu hoá chi phí nhân những bộ ba ma trận liên tiếp ... Cứ tiếp tục như vậy cho tới khi ta tính được phí tổn nhân n ma trận liên tiếp.

Gọi $f[i, j]$ là số phép nhân số học tối thiểu cần thực hiện để nhân đoạn ma trận liên tiếp:

$$\prod_{z=i}^j M_z = M_i M_{i+1} \dots M_j \quad (3.18)$$

Với một cách nhân tối ưu dãy ma trận $\prod_{z=i}^j M_z$, ta quan tâm tới phép nhân ma trận cuối cùng, chẳng hạn $\prod_{z=i}^j M_z$ cuối cùng được tính bằng tích của ma trận X và ma trận Y

$$\prod_{z=i}^j M_z = X \times Y \quad (3.19)$$

Trong đó X là ma trận tạo thành qua phép nhân dãy ma trận từ M_i đến M_k và Y là ma trận tạo thành qua phép nhân dãy ma trận từ M_{k+1} tới M_j với một chỉ số k nào đó.

Vì phép kết hợp chúng ta đang xét là tối ưu để tính $\prod_{z=i}^j M_z$, nên chi phí của phép kết hợp này sẽ bằng tổng của ba đại lượng:

- Chi phí nhân tối ưu dãy ma trận từ M_i đến M_k để được ma trận $X (= f[i, k])$
- Chi phí nhân tối ưu dãy ma trận từ M_{k+1} tới M_j để được ma trận $Y (= f[k + 1, j])$
- Chi phí tính tích $X \times Y (= a_{i-1} \times a_k \times a_j)$

Ta có công thức truy hồi và công thức tính vết:

$$\begin{aligned} f[i, j] &= \min_{k:i \leq k < j} \{f[i, k] + f[k + 1, j] + a_{i-1} a_k a_j\} \\ t[i, j] &= \arg \min_{k:i \leq k < j} \{f[i, k] + f[k + 1, j] + a_{i-1} a_k a_j\} \end{aligned} \quad (3.20)$$

Cơ sở quy hoạch động:

$$f[i, i] = 0, \forall i: 1 \leq i \leq n \quad (3.21)$$

Cách truy vết để in ra phép nhân tối ưu $\prod_{z=i}^j M_z$ sẽ quy về việc in ra phép nhân tối ưu $\prod_{z=i}^k M_z$ và $\prod_{z=k+1}^j M_z$, với $k = t[i, j]$. Điều này có thể thực hiện bằng một thủ tục đệ quy.

□ Cài đặt

MATRIXCHAINMULTIPLICATION_DP.PAS ✓ Nhân tối ưu dãy ma trận

```
{ $MODE OBJFPC }
program MatrixChainMultiplication;
const
  maxN = 100;
  maxSize = 1000;
  maxValue = maxN * maxSize * maxSize * maxSize;
var
  a: array[0..maxN] of Integer;
  f: array[1..maxN, 1..maxN] of Int64;
  t: array[1..maxN, 1..maxN] of Integer;
  n: Integer;

procedure Enter; //Nhập dữ liệu
var
  i: Integer;
begin
  ReadLn(n);
  for i := 0 to n do Read(a[i]);
end;

procedure Optimize; //Giải công thức truy hồi
var
  m, i, j, k: Integer;
  Trial: Int64;
begin
  for i := 1 to n do
```

```

    f[i, i] := 0;
for m := 1 to n - 1 do
  for i := 1 to n - m do
    begin //Tinh f[i, i+m]
      j := i + m;
      f[i, j] := maxValue;
      for k := i to j - 1 do //Thử các vị trí phân hoạch k
        begin
          Trial := f[i, k] + f[k + 1, j] + Int64(a[i - 1]) * a[k] * a[j];
          if Trial < f[i, j] then //Cực tiểu hoá f[i, j] và lưu vết
            begin
              f[i, j] := Trial;
              t[i, j] := k;
            end;
        end;
      end;
    end;
end;

procedure Trace(i, j: Integer); //In ra cách nhân tối ưu dãy M[i..j]
begin
  if i = j then Write('M[', i, ', ')
  else
    begin
      Write('(');
      Trace(i, t[i, j]);
      Write('.');
      Trace(t[i, j] + 1, j);
      Write(')');
    end;
end;

procedure PrintResult; //In kết quả
begin
  WriteLn('The best solution: ');
  Trace(1, n);
  WriteLn;
  WriteLn('Number of numerical multiplications: ', f[1, n]);
end;

begin
  Enter;
  Optimize;
  PrintResult;
end.

```

3.3.6. Du lịch Đông↔Tây

Bạn là người thắng cuộc trong một cuộc thi do một hãng hàng không tài trợ và phần thưởng là một chuyến du lịch do bạn tùy chọn. Có n thành phố và chúng được đánh số từ 1 tới n theo vị trí từ Tây sang Đông (không có hai thành phố nào ở cùng kinh độ), có m tuyến bay hai chiều do hãng quản lý, mỗi tuyến bay nối giữa hai thành phố trong số n thành phố đã cho. Chuyến du lịch của bạn phải xuất phát từ thành phố 1, bay theo các tuyến bay của hãng tới thành phố n và chỉ được bay từ Tây sang Đông, sau đó lại bay theo các tuyến bay của hãng về thành phố

1 và chỉ được bay từ Đông sang Tây. Hành trình không được thăm bất kỳ thành phố nào quá một lần, ngoại trừ thành phố 1 là nơi bắt đầu và kết thúc hành trình.

Yêu cầu đặt ra là tìm hành trình du lịch qua nhiều thành phố nhất.

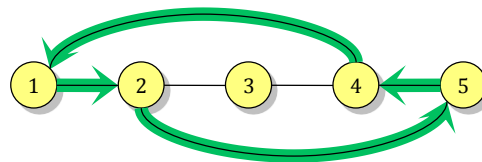
Input

- Dòng 1 chứa số thành phố (n) và số tuyến bay (m), $3 \leq n \leq 100$; $m \leq 10000$
- m dòng tiếp, mỗi dòng chứa thông tin về một tuyến bay: gồm chỉ số hai thành phố tương ứng với tuyến bay đó.

Output

Hành trình tối ưu tìm được hoặc thông báo rằng không thể thực hiện được hành trình theo yêu cầu đặt ra.

Sample Input	Sample Output
5 6	The best tour:
1 2	Number of cities: 4
2 3	1 2 5 4 1
3 4	
4 5	
1 4	
2 5	



□ Thuật toán

Ta có thể đặt một mô hình đồ thị $G = (V, E)$ với tập đỉnh V là tập các thành phố và tập cạnh E là tập các tuyến bay. Bài toán này nếu không có ràng buộc về hành trình Tây-Đông-Tây thì có thể quy dẫn về bài toán tìm chu trình Hamilton (Hamilton Circuit) trên đồ thị G . Bài toán tìm chu trình Hamilton thuộc về lớp bài toán cho tới nay chưa có thuật toán hiệu quả với độ phức tạp đa thức để giải quyết, nhưng bài toán này nhờ có ràng buộc về hướng đi nên chúng ta có thể xây dựng một thuật toán cho kết quả tối ưu với độ phức tạp $O(n^3)$. Tại IOI'93*, bài toán du lịch Đông-Tây này được coi là bài toán khó nhất trong số bốn bài toán của đề thi.

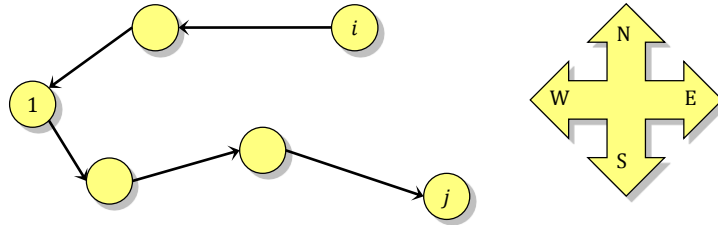
Dễ thấy rằng hành trình đi qua nhiều thành phố nhất cũng là hành trình đi bằng nhiều tuyến bay nhất mà không có thành phố nào thăm qua hai lần ngoại trừ thành phố 1 là nơi bắt đầu và kết thúc hành trình. Với hai thành phố i và j trong đó $i < j$, ta xét các đường bay xuất phát từ thành phố i , bay theo hướng Tây tới thành phố 1 rồi bay theo hướng Đông tới thành phố j sao cho không có thành phố nào bị thăm hai lần.

- Nếu tồn tại đường bay như vậy, ta xét đường bay qua nhiều tuyến bay nhất, ký hiệu $(i \rightsquigarrow 1 \rightsquigarrow j)$ và gọi $f[i, j]$ là số tuyến bay dọc theo đường bay này.

*The 5th International Olympiad in Informatics (Mendoza – Argentina)

- Nếu không tồn tại đường bay thoả mãn điều kiện đặt ra, ta coi $f[i, j] = -1$

Chú ý rằng chúng ta chỉ quan tâm tới các $f[i, j]$ với $i < j$, tức là thành phố i ở phía Tây thành phố j mà thôi.

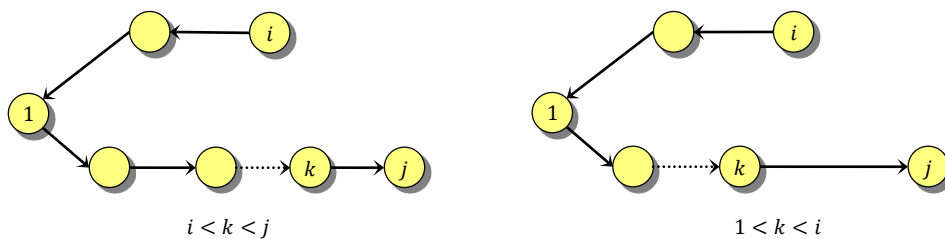


Hình 3-9. Đường bay ($i \rightsquigarrow 1 \rightsquigarrow j$)

Nếu tính được các $f[i, n]$ thì tua du lịch cần tìm sẽ gồm các tuyến bay trên một đường bay ($i \rightsquigarrow 1 \rightsquigarrow n$) nào đó ghép thêm tuyến bay (n, i) , tức là số tuyến bay (nhiều nhất) trên tua du lịch tối ưu cần tìm sẽ là $\max_i \{f[i, n]\} + 1$. Vấn đề bây giờ là phải xây dựng công thức tính các $f[i, j]$.

Với $i < j$, xét đường bay ($i \rightsquigarrow 1 \rightsquigarrow j$), đường bay này sẽ bay qua một thành phố k nào đó trước khi kết thúc tại thành phố j . Có hai khả năng xảy ra:

- Thành phố k nằm phía đông (bên phải) thành phố i ($k > i$), khi đó đường bay ($i \rightsquigarrow 1 \rightsquigarrow j$) có thể coi là đường bay ($i \rightsquigarrow 1 \rightsquigarrow k$) ghép thêm tuyến bay (k, j) . Vậy trong trường hợp này $f[i, j] = f^{(k)}[i, j] = f[i, k] + 1$
- Thành phố k nằm phía tây (bên trái) thành phố i ($k < i$), khi đó đường bay ($i \rightsquigarrow 1 \rightsquigarrow j$) có thể coi là đường bay ($k \rightsquigarrow 1 \rightsquigarrow i$) theo chiều ngược lại rồi ghép thêm tuyến bay (k, j) . Vậy trong trường hợp này $f[i, j] = f^{(k)}[i, j] = f[k, i] + 1$



Hình 3-10. Đường bay ($i \rightsquigarrow 1 \rightsquigarrow j$) và hai trường hợp về vị trí thành phố k

Vì tính cực đại của $f[i, j]$, ta suy ra công thức truy hồi:

$$f[i, j] = \max_k \{f^{(k)}[i, j]\} \quad (3.22)$$

Trong đó các chỉ số k thoả mãn $1 \leq k < j$, $k \neq i$, $f^{(k)}[i, j] \neq -1$ và (k, j) là một tuyến bay. Các giá trị $f^{(k)}[i, j]$ trong công thức truy hồi (3.22) được tính bởi:

$$f^{(k)}[i, j] = \begin{cases} f[i, k] + 1; & \text{nếu } k > i \\ f[k, i] + 1; & \text{nếu } k < i \end{cases} \quad (3.23)$$

Các phần tử của bảng f sẽ được tính theo từng hàng từ trên xuống và trên mỗi hàng thì các phần tử sẽ được tính từ trái qua phải. Các phần tử hàng 1 của bảng f ($f[1, j]$) sẽ được tính trước để làm cơ sở quy hoạch động.

Bài toán tính các giá trị $f[1, j]$ lại là một bài toán quy hoạch động: $f[1, j]$ theo định nghĩa là số tuyến bay trên đường bay Tây→Đông qua nhiều tuyến bay nhất từ 1 tới j . Đường bay này sẽ bay qua thành phố k nào đó trước khi kết thúc tại thành phố j . Theo nguyên lý bài toán con tối ưu, ta có công thức truy hồi:

$$f[1, j] = \max_k \{f[1, k]\} + 1 \quad (3.24)$$

với các thành phố k thoả mãn: $k < j$, $f[1, k] \neq -1$ và (k, j) là một tuyến bay.

Cơ sở quy hoạch động để tính các $f[1, j]$ chính là $f[1, 1] = 0$ (số tuyến bay trên đường bay Tây→Đông qua nhiều tuyến bay nhất từ 1 tới 1)

Quy trình giải có thể tóm tắt qua hai bước chính:

- Đặt $f[1, 1] := 0$ và tính các $f[1, j]$ ($1 < j \leq n$) bằng công thức (3.24)
- Dùng các $f[1, j]$ vừa tính được làm cơ sở, tính các $f[i, j]$ ($2 \leq i \leq n$) bằng công thức (3.22)

Song song với việc tính mỗi $f[i, j]$, chúng ta lưu trữ lại $trace[i, j]$ là thành phố k đứng liền trước thành phố j trên đường bay ($i \rightsquigarrow 1 \rightsquigarrow j$). Việc chỉ ra tua du lịch tối ưu sẽ quy về việc chỉ ra một đường bay ($i \rightsquigarrow 1 \rightsquigarrow n$) nào đó, việc chỉ ra đường bay ($i \rightsquigarrow 1 \rightsquigarrow j$) có thể quy về việc chỉ ra đường bay ($i \rightsquigarrow 1 \rightsquigarrow trace[i, j]$) hoặc đường bay ($trace[i, j] \rightsquigarrow 1 \rightsquigarrow i$) tùy theo giá trị $trace[i, j]$ lớn hơn hay nhỏ hơn i .

□ Cài đặt

 BITONICTOUR_DP.PAS ✓ Du lịch Đông↔Tây

```
{ $MODE OBJFPC }
program TheLongestBitonicTour;
const
  max = 1000;
type
  TPath = record
    nCities: Integer;
    c: array[1..max] of Integer;
  end;
var
  a: array[1..max, 1..max] of Boolean;
  f: array[1..max, 1..max] of Integer;
  trace: array[1..max, 1..max] of Integer;
  p, q: TPath;
  n: Integer;
  LongestTour: Integer;
```

```

procedure Enter; //Nhập dữ liệu
var
    i, m, u, v: Integer;
begin
    FillChar(a, SizeOf(a), False); //a[u,v] = True ↔ (u,v) là một tuyến bay
    ReadLn(n, m);
    for i := 1 to m do
        begin
            ReadLn(u, v); //Tuyến bay hai chiều: a[u,v] = a[v,u]
            a[u, v] := True;
            a[v, u] := True;
        end;
    end;

//target := Max(target, value), trả về True nếu target được cực đại hoá.
function Maximize(var target: Integer; value: Integer): Boolean;
begin
    Result := value > target;
    if Result then target := value;
end;

procedure Optimize; //Giải công thức truy hồi
var
    i, j, k: Integer;
begin
    //Tính các f[1,j]
    f[1, 1] := 0; //Cơ sở QHĐ để tính các f[1,j]
    for j := 2 to n do
        begin //Tính f[1,j]
            f[1, j] := -1;
            for k := 1 to j - 1 do
                if a[k, j] and (f[1, k] <> -1) and
                    Maximize(f[1, j], f[1, k] + 1) then
                    trace[1, j] := k; //Lưu vết mỗi khi f[1,j] được cực đại hoá
            end;
    //Dùng các f[1,j] làm cơ sở QHĐ, tính các f[i,j]
    for i := 2 to n - 1 do
        for j := i + 1 to n do
            begin
                f[i, j] := -1;
                for k := i + 1 to j - 1 do //k nằm phía Đông i
                    if a[k, j] and (f[i, k] <> -1) and
                        Maximize(f[i, j], f[i, k] + 1) then
                            trace[i, j] := k;
                for k := 1 to i - 1 do //k nằm phía Tây i
                    if a[k, j] and (f[k, i] <> -1) and
                        Maximize(f[i, j], f[k, i] + 1) then
                            trace[i, j] := k;
            end;
        end;
    end;

procedure FindPaths; //Truy vết tìm đường
var
    i, j, k, t: Integer;
begin
    //Tính LongestTour là số tuyến bay nhiều nhất trên đường bay k→1→n với mọi k

```

```

LongestTour := -1;
for k := 1 to n - 1 do
  if a[k, n] and Maximize(LongestTour, f[k, n]) then
    i := k;
if LongestTour = -1 then Exit;
j := n;
//Tua du lịch cần tìm sẽ là gồm các tuyến bay trên đường bay  $i \rightarrow 1 \rightarrow j = n$  ghép thêm tuyến  $(i, n)$ 
//Chúng ta tìm 2 đường bay Đông→Tây:  $p: i \rightarrow 1$  và  $q: j \rightarrow 1$  gồm các tuyến bay trên đường bay  $i \rightarrow 1 \rightarrow j$ 
p.nCities := 1; p.c[1] := i; //Lưu trữ thành phố cuối cùng của đường p
q.nCities := 1; q.c[1] := j; //Lưu trữ thành phố cuối cùng của đường q
repeat
  if i < j then //Truy vết đường bay  $i \rightarrow 1 \rightarrow j$ 
    with q do
      begin
        k := trace[i, j]; //Xét thành phố k đứng liền trước j
        Inc(nCities); c[nCities] := k; //Đưa k vào đường q
        j := k;
      end
    else //Truy vết đường bay  $j \rightarrow 1 \rightarrow i$ 
      with p do
        begin
          k := trace[j, i]; //Xét thành phố k đứng liền trước i
          Inc(nCities); c[nCities] := k; //Đưa k vào đường p
          i := k;
        end;
      until i = j; //Khi  $i = j$  thì chắc chắn  $i = j = 1$ 
end;

procedure PrintResult; //In kết quả
var
  i: Integer;
begin
  if LongestTour = -1 then
    WriteLn('NO SOLUTION!')
  else
    begin
      WriteLn('The best tour: ');
      WriteLn('Number of cities: ', LongestTour + 1);
      //Để in ra tua du lịch tối ưu, ta sẽ in ghép 2 đường:
      //In ngược đường p để có đường đi Tây→Đông  $1 \rightarrow i$ 
      //In xuôi đường q để có đường đi Đông→Tây  $n \rightarrow 1$ 
      for i := p.nCities downto 1 do Write(p.c[i], ' ');
      for i := 1 to q.nCities do Write(q.c[i], ' ');
    end;
end;

begin
  Enter;
  Optimize;
  FindPaths;
  PrintResult;
end.

```

3.3.7. Thuật toán Viterbi

Thuật toán Viterbi được đề xuất bởi Andrew Viterbi như một phương pháp hiệu chỉnh lỗi trên đường truyền tín hiệu số. Nó được sử dụng để giải mã chấp sử dụng trong điện thoại di động

kỹ thuật số (CDMA/GSM), đường truyền vệ tinh, mạng không dây, v.v... Trong ngành khoa học máy tính, thuật toán Viterbi được sử dụng rộng rãi trong Tin-Sinh học, xử lý ngôn ngữ tự nhiên, nhận dạng tiếng nói. Khi nói tới thuật toán Viterbi, không thể không kể đến một ứng dụng quan trọng của nó trong mô hình Markov ẩn (Hidden Markov Models - HMMs) để tìm dãy trạng thái tối ưu đối với một dãy tín hiệu quan sát được.

Việc trình bày cụ thể một mô hình thực tế có ứng dụng của thuật toán Viterbi là rất dài dòng, chúng ta sẽ tìm hiểu thuật toán này qua một bài toán đơn giản hơn để qua đó hình dung được cách thức thuật toán Viterbi tìm đường đi tối ưu trên lưới như thế nào.

□ Bài toán

Một dây chuyền lắp ráp ô tô có một robot và n dụng cụ đánh số từ 1 tới n . Có tất cả m loại bộ phận trong một chiếc ô tô đánh số từ 1 tới m . Mỗi chiếc ô tô phải được lắp ráp từ t bộ phận $O = (o_1, o_2, \dots, o_t)$ theo đúng thứ tự này ($\forall i: 1 \leq o_i \leq m$). Biết được những thông tin sau:

- Tại mỗi thời điểm, robot chỉ có thể cầm được 1 dụng cụ.
- Tại thời điểm bắt đầu, robot không cầm dụng cụ gì cả và phải chọn một trong số n dụng cụ đã cho, thời gian chọn không đáng kể.
- Khi đã có dụng cụ, robot sẽ sử dụng nó để lắp một bộ phận trong dãy O , biết thời gian để Robot lắp bộ phận loại v bằng dụng cụ thứ i là b_{iv} ($1 \leq i \leq n; 1 \leq v \leq m$)
- Sau khi lắp xong mỗi bộ phận, robot được phép đổi dụng cụ khác để lắp bộ phận tiếp theo, biết thời gian đổi từ dụng cụ i sang dụng cụ j là a_{ij} . (a_{ij} có thể khác a_{ji} và a_{ii} luôn bằng 0).

Hãy tìm ra quy trình lắp ráp ô tô một cách nhanh nhất.

Input

- Dòng 1: Chứa ba số nguyên dương n, m, t ($n, m, t \leq 500$)
- Dòng 2: Chứa t số nguyên dương o_1, o_2, \dots, o_t ($\forall i: 1 \leq o_i \leq m$)
- n dòng tiếp theo, dòng thứ i chứa n số nguyên, số thứ j là a_{ij} ($0 \leq a_{ij} \leq 500$)
- n dòng tiếp theo, dòng thứ j chứa m số nguyên, số thứ v là b_{iv} ($0 \leq b_{iv} \leq 500$)

Output

Quy trình lắp ráp ô tô nhanh nhất

Sample Input	Sample Output
3 4 6	Component Tool
1 2 3 2 3 4	-----
0 1 2	1 3
3 0 4	2 2
5 6 0	3 1
8 8 1 5	2 2
8 1 8 8	3 1
1 8 8 5	4 1
	Time for assembling: 23

□ Thuật toán

Gọi $f[k, i]$ là thời gian ít nhất để lắp ráp lần lượt các bộ phận o_k, o_{k+1}, \dots, o_t mà dụng cụ dùng để lắp bộ phận đầu tiên trong dãy (o_k) là dụng cụ i . Nếu dụng cụ dùng để lắp bộ phận tiếp theo (o_{k+1}) là dụng cụ j thì thời gian ít nhất để lắp lần lượt các bộ phận o_k, o_{k+1}, \dots, o_t : $f[k, i]$ sẽ được tính bằng:

- Thời gian lắp bộ phận o_k : $b[i, o_k]$
- Cộng với thời gian đổi từ dụng cụ i sang dụng cụ j : $a[i, j]$
- Cộng với thời gian ít nhất để lắp ráp lần lượt các bộ phận $o_{k+1}, o_{k+2}, \dots, o_t$, trong đó bộ phận o_{k+1} được lắp bằng dụng cụ j : $f[k + 1, j]$

Vì chúng ta muốn $f[k, i]$ là nhỏ nhất có thể, chúng ta sẽ thử mọi khả năng chọn dụng cụ j để lắp o_{k+1} và chọn phương án tốt nhất để cực tiểu hoá $f[k, i]$. Từ đó suy ra công thức truy hồi:

$$f[k, i] = \min_{j:1 \leq j \leq n} \{b[i, o_k] + a[i, j] + f[k + 1, j]\} \quad (3.25)$$

Bảng phương án f là bảng hai chiều t hàng, n cột. Trong đó các phần tử ở hàng k sẽ được tính qua các phần tử ở hàng $k + 1$, như vậy ta sẽ phải tính những phần tử ở hàng t của bảng làm cơ sở quy hoạch động. Theo định nghĩa, $f[t, i]$ là thời gian (ít nhất) để lắp duy nhất một bộ phận o_t bằng dụng cụ i nên suy ra $f[t, i] = b[i, o_t]$.

Song song với việc tính các phần tử bảng f bằng công thức (3.25), mỗi khi tính được $f[k, i]$ đạt giá trị nhỏ nhất tại $j = j^*$ nào đó, chúng ta đặt:

$$trace[k, i] := j^* = \arg \min_{j:1 \leq j \leq n} \{b[i, o_k] + a[i, j] + f[k + 1, j]\} \quad (3.26)$$

để chỉ ra rằng phương án tối ưu tương ứng với $f[k, j]$ sẽ phải chọn dụng cụ tiếp theo là dụng cụ $trace[k, i]$ để lắp bộ phận o_{k+1} .

Sau khi tính được tất cả các phần tử của bảng f , ta quan tâm tới các phần tử trên hàng 1. Theo định nghĩa $f[1, i]$ là thời gian ít nhất để lắp hoàn chỉnh một chiếc ô tô với dụng cụ đầu tiên được sử dụng là i . Cũng tương tự trên, chúng ta muốn lắp hoàn chỉnh một chiếc ô tô trong thời gian ngắn nhất nên công việc còn lại là thử mọi khả năng chọn dụng cụ đầu tiên và chọn ra dụng cụ s_1 có $f[1, s_1]$ nhỏ nhất, $f[1, s_1]$ chính là thời gian ngắn nhất để lắp hoàn chỉnh một chiếc ô tô. Từ cơ chế lưu vết, ta có dụng cụ tiếp theo cần sử dụng để lắp o_2 là $s_2 = trace[1, s_1]$, dụng cụ cần sử dụng để lắp o_3 là $s_3 = trace[2, s_2], \dots$ Những công đoạn chính trong việc thiết kế một thuật toán quy hoạch động đã hoàn tất, bao gồm: Dùng công thức truy hồi, tìm cơ sở quy hoạch động, tìm cơ chế lưu vết và truy vết.

□ Cài đặt

Nhìn vào công thức truy hồi (3.25) để tính các $f[k, i]$, có thể nhận thấy rằng việc tính phần tử trên hàng k chỉ cần dựa vào các phần tử trên hàng $k + 1$. Vậy thì tại mỗi bước tính một hàng của bảng phương án, ta chỉ cần lưu trữ hai hàng liên tiếp dưới dạng hai mảng một chiều x và y : mảng $x_{1..n}$ tương ứng với hàng vừa được tính (hàng $k + 1$) và mảng $y_{1..n}$ tương ứng với

hàng liền trước sắp được tính (hàng k), công thức truy hồi dùng mảng x tính mảng y sẽ được viết lại là:

$$y[i] = \min_{j:1 \leq j \leq n} \{b[i, o_k] + a[i, j] + x[j]\} \quad (3.27)$$

Sau khi tính xong một hàng, hai mảng x và y được đổi vai trò cho nhau và quá trình tính được lặp lại. Cuối cùng chúng ta vẫn lưu trữ được hàng 1 của bảng phương án và từ đó truy vết tìm ra phương án tối ưu.

Mẹo nhỏ này không những tiết kiệm bộ nhớ cho bảng phương án mà còn làm tăng tốc đáng kể quá trình tính toán so với phương pháp tính trực tiếp trên mảng hai chiều f . Tốc độ được cải thiện nhờ hai nguyên nhân chính:

- Việc truy cập phần tử của mảng một chiều nhanh hơn việc truy cập phần tử của mảng hai chiều vì phép tính địa chỉ được thực hiện đơn giản hơn, hơn nữa nếu bạn bật chế độ kiểm tra tràn phạm vi (range checking) khi dịch chương trình, truy cập phần tử mảng một chiều chỉ cần kiểm tra phạm vi một chỉ số trong khi truy cập phần tử mảng hai chiều phải kiểm tra phạm vi cả hai chỉ số.
- Nếu dùng hai mảng một chiều tính xoay lẫn nhau, vùng bộ nhớ của cả hai mảng bị đọc/ghi nhiều lần và bộ vi xử lý sẽ có cơ chế nạp cả hai mảng này vào vùng nhớ cache ngay trong bộ vi xử lý, việc đọc/ghi dữ liệu sẽ không qua RAM nữa nên đạt tốc độ nhanh hơn nhiều. (Sự chậm chạp của RAM so với cache cũng có thể so sánh như tốc độ của đĩa cứng và RAM).

Bạn có thể cài đặt thử phương pháp tính trực tiếp mảng hai chiều và phương pháp tính xoay vòng hai mảng một chiều để so sánh tốc độ trên các bộ dữ liệu lớn. Để thấy rõ hơn sự khác biệt về tốc độ, có thể đổi yêu cầu của bài toán chỉ là đưa ra thời gian ngắn nhất để lắp ráp mà không cần đưa ra phương án thực thi, khi đó mảng hai chiều *trace* cũng không cần dùng đến nữa.

VITERBI_DP.PAS ✓ Lắp ráp ô tô

```
{ $MODE OBJFPC }
program ViterbiAlgorithm;
const
  maxN = 500;
  maxM = 500;
  maxT = 500;
  maxTime = 500;
  Infinity = maxT * maxTime + 1;
type
  TLine = array[1..maxN] of Integer;
  PLine = ^TLine;
var
  a: array[1..maxN, 1..maxN] of Integer;
  b: array[1..maxN, 1..maxM] of Integer;
  o: array[1..maxT] of Integer;
  x, y: PLine;
```

```

    trace: array[1..maxT, 1..maxN] of Integer;
    n, m, t: Integer;
    MinTime: Integer;
    Tool: Integer;

procedure Enter; //Nhập dữ liệu
var
    i, j, k: Integer;
begin
    ReadLn(n, m, t);
    for i := 1 to t do Read(o[i]);
    ReadLn;
    for i := 1 to n do
        begin
            for j := 1 to n do Read(a[i, j]);
            ReadLn;
        end;
    for i := 1 to n do
        begin
            for k := 1 to m do Read(b[i, k]);
            ReadLn;
        end;
    end;
end;

//target := Min(target, value); Trả về True nếu target được cực tiểu hoá
function Minimize(var target: Integer; value: Integer): Boolean;
begin
    Result := value < target;
    if Result then target := value;
end;

procedure Optimize; //Quy hoạch động
var
    i, j, k: Integer;
    temp: PLine;
begin
    New(x); New(y);
    try
        for i := 1 to n do //Cơ sở quy hoạch động: x^ := hàng t của bảng phương án
            x^[i] := b[i, o[t]];
        for k := t - 1 downto 1 do
            begin //Dùng x^ tính y^ ↔ dùng f[k + 1, .] tính f[k, .]
                for i := 1 to n do
                    begin
                        y^[i] := Infinity;
                        for j := 1 to n do
                            if Minimize(y^[i], a[i, j] + x^[j]) then
                                trace[k, i] := j; //Cực tiểu hoá y^[i] kết hợp lưu vết
                                Inc(y^[i], b[i, o[k]]);
                    end;
                temp := x; x := y; y := temp; //Đảo hai con trỏ ↔ đảo vai trò x và y
            end;
        //x^ giờ đây là hàng 1 của bảng phương án, tìm phần tử nhỏ nhất của x^ và dụng cụ đầu tiên được dùng
        MinTime := Infinity;
        for i := 1 to n do
            if x^[i] < MinTime then
                begin

```

```

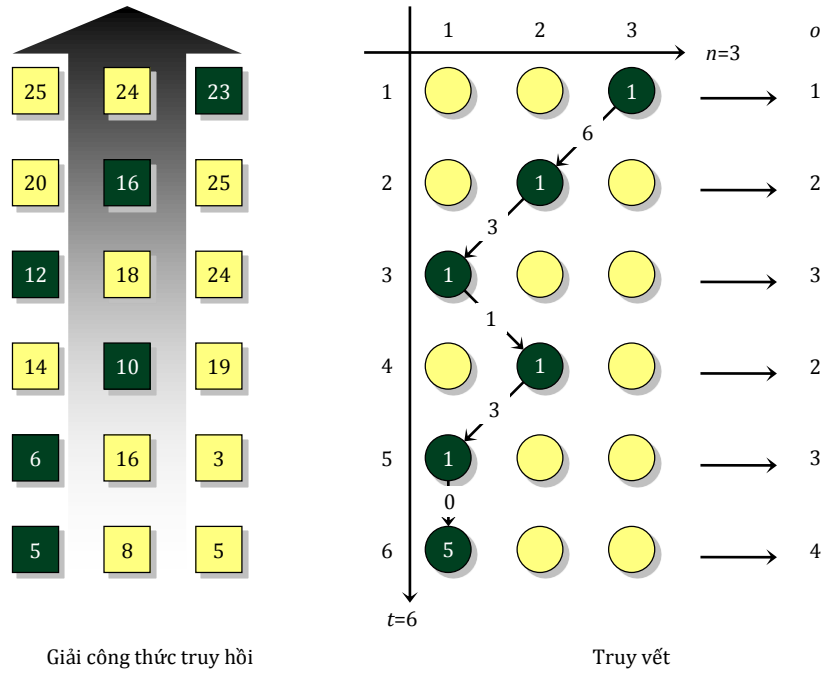
        MinTime := x^[i];
        Tool := i;
    end;
finally
    Dispose(x); Dispose(y);
end;
end;

procedure PrintResult; //In kết quả
var
    k: Integer;
begin
    WriteLn('Component  Tool');
    WriteLn('-----');
    for k := 1 to t do
        begin
            WriteLn(o[k]:5, Tool:9); //In ra dụng cụ Tool dùng để lắp o[k]
            Tool := trace[k, Tool]; //Chuyển sang dụng cụ kế tiếp
        end;
    WriteLn('Time for assembling: ', MinTime);
end;

begin
    Enter;
    Optimize;
    PrintResult;
end.

```

Người ta thường phát biểu thuật toán Viterbi trên mô hình đồ thị: Xét đồ thị có hướng gồm $t \times n$ đỉnh, tập các đỉnh này được chia làm t lớp, mỗi lớp có đúng n đỉnh. Các cung của đồ thị chỉ nối từ một đỉnh đến một đỉnh khác thuộc lớp kế tiếp. Mỗi đỉnh và mỗi cung của đồ thị đều có gán một trọng số (chi phí). Trọng số (chi phí) của một đường đi trên đồ thị bằng tổng trọng số các đỉnh và các cung đi qua. Thuật toán Viterbi chính là để tìm đường đi ngắn nhất (có chi phí ít nhất) trên lưới từ lớp 1 tới lớp t . Trong ví dụ cụ thể này, trọng số cung nối đỉnh i của lớp k với đỉnh j của lớp $k + 1$ chính là $a[i, j]$ và trọng số đỉnh i của lớp k chính là $b[i, o_k]$. Thuật toán Viterbi cần thời gian $\Theta(n^2t)$ để tính bảng phương án và cần thời gian $\Theta(t)$ để truy vết tìm ra nghiệm. Hình 3-11 mô tả cách thức tính bảng phương án và truy vết trên ví dụ cụ thể của đề bài.



Hình 3-11. Thuật toán Viterbi tính toán và truy vết

Một bài toán quy hoạch động có thể có nhiều cách tiếp cận khác nhau, chọn cách nào là tùy theo yêu cầu bài toán sao cho thuận tiện. Điều quan trọng nhất để giải một bài toán quy hoạch động chính là phải nhìn ra được bản chất đệ quy của bài toán và tìm ra công thức truy hồi để giải. Việc này không có phương pháp chung nào cả mà hoàn toàn dựa vào sự khéo léo và kinh nghiệm của bạn - những kỹ năng chỉ có được nhờ luyện tập.

Bài tập 3-1.

Cho ba số nguyên dương n, k và p ($n, k \leq 1000$), hãy cho biết có bao nhiêu số tự nhiên có không quá n chữ số mà tổng các chữ số đúng bằng k , đồng thời cho biết nếu mang tất cả các số đó sắp xếp theo thứ tự tăng dần thì số đứng thứ p là số nào.

Gợi ý: Tìm công thức truy hồi tính $f[i, j]$ là số các số có $\leq i$ chữ số mà tổng các chữ số đúng bằng j .

Bài tập 3-2.

Cho dãy số nguyên dương $A = (a_1, a_2, \dots, a_n)$ ($n \leq 1000; \forall i: a_i \leq 1000$) và một số m . Hãy chọn ra một số phần tử trong dãy A mà các phần tử được chọn có tổng đúng bằng m .

Gợi ý: Tìm công thức truy hồi tính $f[t]$ là chỉ số nhỏ nhất thoả mãn: tồn tại cách chọn trong dãy $a_1, a_2, \dots, a_{f[t]}$ ra một số phần tử có tổng đúng bằng t .

Bài tập 3-3.

Cho dãy số tự nhiên $A = (a_1, a_2, \dots, a_n)$. Ban đầu các phần tử của A được đặt liên tiếp theo đúng thứ tự cách nhau bởi dấu “?”:

$$a_1? a_2? \dots ? a_n$$

Yêu cầu: Cho trước số nguyên m , hãy tìm cách thay các dấu “?” bằng dấu cộng hay dấu trừ để được một biểu thức số học cho giá trị là m . Biết rằng $1 \leq n \leq 1000$ và $0 \leq a_i \leq 1000, \forall i$.

Ví dụ: Ban đầu dãy A là $1? 2? 3? 4$, với $m = 0$ sẽ có phương án $1 - 2 - 3 + 4$.

Bài tập 3-4. Cho một lưới ô vuông kích thước $m \times n$ ($m, n \leq 1000$), các hàng của lưới được đánh số từ 1 tới m và các cột của lưới được đánh số từ 1 tới n , trên mỗi ô của lưới ghi một số nguyên có giá trị tuyệt đối không quá 1000. Người ta muốn tìm một cách đi từ cột 1 tới cột n của lưới theo quy tắc: Từ một ô (i, j) chỉ được phép đi sang một trong các ô ở cột bên phải có đỉnh chung với ô (i, j) . Hãy chỉ ra cách đi mà tổng các số ghi trên các ô đi qua là lớn nhất.

7	2	1	2	6
1	2	5	4	5
1	5	3	5	2
5	2	3	1	1

Bài tập 3-5.

Lập trình giải bài toán cái túi với kích thước dữ liệu: $n \leq 10000, m \leq 10000$ và giới hạn bộ nhớ 10MB.

Gợi ý: Vẫn sử dụng công thức truy hồi như ví dụ trong bài, nhưng thay đổi cơ chế lưu trữ. Giải công thức truy hồi lần 1, không lưu trữ toàn bộ bảng phương án mà cứ cách 100 hàng mới lưu lại 1 hàng. Sau đó với hai hàng được lưu trữ liên tiếp thì lấy hàng trên làm cơ sở, giải công thức truy hồi lần 2 tính qua 100 hàng đến hàng dưới, nhưng lần này lưu trữ toàn bộ những hàng tính được để truy vết.

Bài tập 3-6. (Dãy con chung dài nhất Longest Common Subsequence-LCS)

Xâu ký tự S gọi là xâu con của xâu ký tự T nếu có thể xoá bớt một số ký tự trong xâu T để được xâu S . Cho hai xâu ký tự $X = x_1x_2\dots x_m$ và $Y = y_1y_2\dots y_n$ ($m, n \leq 1000$). Tìm xâu Z có độ dài lớn nhất là xâu con của cả X và Y . Ví dụ: $X = 'abcdefghi123', Y = 'abc1def2ghi3'$ thì xâu Z cần tìm là $Z = 'abcdefghi3'$.

Gợi ý: Tìm công thức truy hồi tính $f[i, j]$ là độ dài xâu con chung dài nhất của hai xâu $x_1x_2\dots x_i$ và $y_1y_2\dots y_j$.

Bài tập 3-7.

Một số nguyên dương x gọi là con của số nguyên dương y nếu ta có thể xoá bớt một số chữ số của y để được x . Cho hai số nguyên dương a và b ($a, b \leq 10^{1000}$) hãy tìm số c là con của cả a

và b sao cho giá trị của c là lớn nhất có thể. Ví dụ với $a = 123456123$, $b = 456123456$ thì $c = 456123$

Bài tập 3-8.

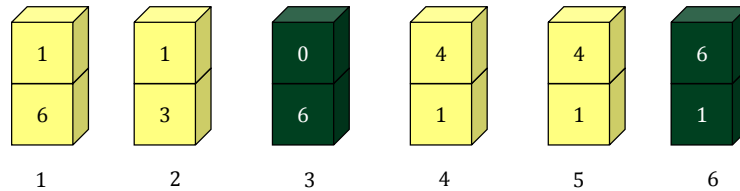
Một chuỗi ký tự X gọi là chứa chuỗi ký tự Y nếu như có thể xoá bớt một số ký tự trong chuỗi X để được chuỗi Y . Một chuỗi ký tự gọi là đối xứng (palindrome) nếu nó không thay đổi khi ta viết các ký tự trong chuỗi theo thứ tự ngược lại. Cho một chuỗi ký tự S có độ dài không quá 1000, hãy tìm chuỗi đối xứng T chứa chuỗi S và có độ dài ngắn nhất có thể.

Bài tập 3-9.

Có n loại tiền giấy đánh số từ 1 tới n , tờ giấy bạc loại i có mệnh giá là một số nguyên dương v_i ($n \leq 100, \forall i: v_i \leq 10000$). Hỏi muốn mua một món hàng giá là m ($m \leq 10000$) thì có bao nhiêu cách trả số tiền đó bằng những loại giấy bạc đã cho, nếu tồn tại cách trả, cho biết cách trả phải dùng ít tờ tiền nhất.

Bài tập 3-10.

Cho n quân cờ dominos xếp dựng đứng theo hàng ngang và được đánh số từ 1 đến n . Quân cờ dominos thứ i có số ghi ở ô trên là a_i và số ghi ở ô dưới là b_i . Xem hình vẽ:



Biết rằng $1 \leq n \leq 1000$ và $0 \leq a_i, b_i \leq 6, \forall i$. Cho phép lật ngược các quân dominos. Khi quân dominos thứ i bị lật, nó sẽ có số ghi ở ô trên là b_i và số ghi ở ô dưới là a_i . Vấn đề đặt ra là hãy tìm cách lật các quân dominos sao cho chênh lệch giữa tổng các số ghi ở hàng trên và tổng các số ghi ở hàng dưới là tối thiểu. Nếu có nhiều phương án lật tốt như nhau, thì chỉ ra phương án phải lật ít quân nhất.

Như ví dụ trên thì sẽ lật hai quân dominos thứ 3 và thứ 6. Khi đó:

Tổng các số ở hàng trên: $1 + 1 + 6 + 4 + 4 + 1 = 17$

Tổng các số ở hàng dưới: $6 + 3 + 0 + 1 + 1 + 6 = 17$

Bài tập 3-11.

Xét bảng $H = \{h_{ij}\}$ kích thước 4×4 , các hàng và các cột được đánh chỉ số A, B, C, D. Trên 16 ô của bảng, mỗi ô ghi 1 ký tự A hoặc B hoặc C hoặc D.

H	A	B	C	D
A	A	A	B	B
B	C	D	A	B
C	B	C	B	A
D	B	D	D	D

Cho xâu $S = s_1s_2\dots s_n$ chỉ gồm các chữ cái $\{A, B, C, D\}$. ($1 \leq n \leq 10^6$). Xét phép co $R(i)$: thay ký tự s_i và s_{i+1} bởi ký tự $h[s_i, s_{i+1}]$. Ví dụ: $S = 'ABCD'$, áp dụng liên tiếp 3 lần $R(1)$ sẽ được: $ABCD \rightarrow ACD \rightarrow BD \rightarrow B$

Yêu cầu: Cho trước một ký tự $x \in \{A, B, C, D\}$, hãy chỉ ra thứ tự thực hiện $n - 1$ phép co để ký tự còn lại cuối cùng trong S là x .

Bài tập 3-12.

Bạn có n việc cần làm đánh số từ 1 tới n , việc thứ i cần làm liên tục trong t_i đơn vị thời gian và nếu bạn hoàn thành việc thứ i không muộn hơn thời điểm d_i , bạn sẽ thu được số tiền là p_i . Bạn bắt đầu từ thời điểm 0 và không được phép làm một lúc hai việc mà phải thực hiện các công việc một cách tuần tự. Hãy chọn ra một số việc và lên kế hoạch hoàn thành các việc đã chọn sao cho tổng số tiền thu được là nhiều nhất. Biết rằng $n \leq 10^6$ và các giá trị t_i, d_i, p_i là số nguyên dương không quá 10^6

Bài tập 3-13.

Cho dãy số nguyên $A = (a_1, a_2, \dots, a_n)$ ($1 \leq n \leq 10000; |a_i| \leq 10^9$) và một số nguyên dương $k \leq 1000$, hãy chọn ra một dãy con gồm nhiều phần tử nhất của A có tổng chia hết cho k .

Bài tập 3-14.

Công ty trách nhiệm hữu hạn "Vui vẻ" có n cán bộ đánh số từ 1 tới n ($n \leq 10^5$). Cán bộ thứ i có đánh giá độ vui tính là h_i và có một thủ trưởng trực tiếp b_i , giả thiết rằng nếu i là giám đốc công ty thì $b_i = 0$. Bạn cần giúp công ty mời một nhóm cán bộ đến dự dạ tiệc "Những người thích đùa" sao cho tổng đánh giá độ vui tính của những người dự tiệc là lớn nhất, sao cho trong số những người được mời không đồng thời có mặt nhân viên cùng thủ trưởng trực tiếp của người đó.

Bài 4. Tham lam

Tham lam (greedy) là một phương pháp giải các bài toán tối ưu. Các thuật toán tham lam dựa vào sự đánh giá *tối ưu cục bộ địa phương (local optimum)* để đưa ra *quyết định tức thì* tại mỗi bước lựa chọn, với hy vọng cuối cùng sẽ tìm ra được phương án *tối ưu tổng thể (global optimum)*.

Thuật toán tham lam có trường hợp luôn tìm ra đúng phương án tối ưu, có trường hợp không. Nhưng trong trường hợp thuật toán tham lam không tìm ra đúng phương án tối ưu, chúng ta thường thu được một phương án khả dĩ chấp nhận được.

Với một bài toán có nhiều thuật toán để giải quyết, thông thường thuật toán tham lam có tốc độ tốt hơn hẳn so với các thuật toán tối ưu tổng thể.

Khác với các kỹ thuật thiết kế thuật toán như chia để trị, liệt kê, quy hoạch động mà chúng ta đã biết, rất khó để đưa ra một quy trình chung để tiếp cận bài toán, tìm thuật toán cũng như cài đặt thuật toán tham lam. Nếu nói về cách nghĩ, tôi chỉ có thể đưa ra hai kinh nghiệm:

- Thử tìm một thuật toán tối ưu tổng thể (ví dụ như quy hoạch động), sau đó đánh giá lại thuật toán, nhìn lại mỗi bước tối ưu và đặt câu hỏi “Liệu tại bước này có cần phải làm đến thế không?”.
- Hoặc thử nghĩ xem nếu như không có máy tính, không có khái niệm gì về các chiến lược tối ưu tổng thể thì ở góc độ con người, chúng ta sẽ đưa ra giải pháp như thế nào?

4.1. Giải thuật tham lam

Giả sử có n loại tiền giấy, loại tiền thứ i có mệnh giá là v_i (đồng). Hãy chỉ ra cách trả dùng ít tờ tiền nhất để mua một mặt hàng có giá là m đồng.

Bài toán này là một bài toán có thể giải theo nhiều cách...

Nếu có các loại tiền như hệ thống tiền tệ của Việt Nam: 1 đồng, 2 đồng, 5 đồng, 10 đồng, 20 đồng, 50 đồng, 100 đồng, 200 đồng, 500 đồng...thì để trả món hàng giá 9 đồng:

- Một người máy được cài đặt chiến lược tìm kiếm vét cạn sẽ duyệt tổ hợp của tất cả các tờ tiền và tìm tổ hợp gồm ít tờ tiền nhất có tổng mệnh giá là 9 đồng. Vấn đề sẽ xảy ra nếu món hàng không phải giá 9 đồng mà 9 tỉ đồng thì người máy sẽ nghĩ đến khi ... hết điện thì thôi.
- Một người máy khác được cài đặt chiến lược chia để trị (đệ quy hoặc quy hoạch động) sẽ nghĩ theo cách khác, để trả 9 đồng dĩ nhiên không cần dùng đến những tờ tiền mệnh giá lớn hơn 9. Bài toán trở thành đổi 9 đồng ra những tờ tiền 1 đồng, 2 đồng và 5 đồng. Nếu có một tờ k đồng trong phương án tối ưu thì vấn đề còn lại là đổi $9 - k$ đồng, quy về ba bài toán con (với $k = 1, 2, 5$), giải chúng và chọn phương án tốt nhất trong ba lời giải. Khi mà số tiền cần quy đổi rất lớn, giới hạn về thời gian và bộ nhớ sẽ làm cho giải pháp của người máy này bất khả thi.

- Bây giờ hãy thử nghĩ theo một cách rất con người, khi các bà nội trợ đi mua sắm, họ chẳng có máy tính, không có khái niệm gì về thuật toán chia để trị, quy hoạch động, vét cạn,... (mà có biết họ cũng chẳng dùng). Có điều để trả 9 đồng, chắc chắn họ sẽ trả bằng 1 tờ 5 đồng và 2 tờ 2 đồng. Cách làm của họ rất đơn giản, mỗi khi rút một tờ tiền ra trả, họ *quyết định tức thời* bằng cách lấy ngay tờ tiền mệnh giá cao nhất không vượt quá giá trị cần trả, mà không cần để ý đến “hậu quả” của sự quyết định đó.

Xét về giải pháp của con người trong bài toán đổi tiền, trên thực tế tôi chưa biết hệ thống tiền tệ nào khiến cho cách làm này sai. Tuy nhiên trên lý thuyết, có thể chỉ ra những hệ thống tiền tệ mà cách làm này cho ra giải pháp không tối ưu.

Giả sử chúng ta có 3 loại tiền: 1 đồng, 5 đồng và 8 đồng. Nếu cần đổi 10 đồng thì phương án tối ưu phải là dùng 2 tờ 5 đồng, nhưng cách làm này sẽ cho phương án 1 tờ 8 đồng và 2 tờ 1 đồng. Hậu quả của phép chọn tờ tiền 8 đồng đã làm cho giải pháp cuối cùng không tối ưu, nhưng dù sao cũng có thể tạm chấp nhận được trên thực tế.

Hậu quả của phép chọn tham lam tức thời sẽ tệ hại hơn nếu chúng ta chỉ có 2 loại tiền: 5 đồng và 8 đồng. Khi đó thuật toán này sẽ thất bại nếu cần đổi 10 đồng vì khi rút tờ 8 đồng ra rồi, không có cách nào đổi 2 đồng nữa.

4.2. Thiết kế một thuật toán tham lam

Thực ra chỉ có một kinh nghiệm duy nhất khi tiếp cận bài toán và tìm thuật toán tham lam là phải *khảo sát kỹ bài toán* để tìm ra các tính chất đặc biệt mà ở đó ta có thể đưa ra quyết định tức thời tại từng bước dựa vào sự đánh giá tối ưu cục bộ địa phương.

Những ví dụ dưới đây nêu lên một vài phương pháp tiếp cận bài toán và thiết kế giải thuật tham lam phổ biến.

4.3. Một số ví dụ về giải thuật tham lam

Với các bài toán mang bản chất đệ quy chúng ta có thể áp dụng quy trình sau để tìm thuật toán tham lam (nếu có):

- Phân rã bài toán lớn ra thành các bài toán con đồng dạng mà nghiệm của các bài toán con có thể dùng để chỉ ra nghiệm của bài toán lớn. Bước này giống như thuật toán chia để trị và chúng ta có thể thiết kế sơ bộ một mô hình chia để trị.
- Chỉ ra rằng *không cần giải toàn bộ các bài toán con* mà chỉ cần giải *một* bài toán con thôi là có thể chỉ ra nghiệm của bài toán lớn. Điều này cực kỳ quan trọng, nó chỉ ra rằng sự lựa chọn tham lam tức thời (tối ưu cục bộ) tại mỗi bước sẽ dẫn đến phương án tối ưu tổng thể.
- Phân tích dãy quyết định tham lam để sửa mô hình chia để trị thành một giải thuật lặp.

4.3.1. Xếp lịch thực hiện các nhiệm vụ

Bài toán đầu tiên chúng ta khảo sát là bài toán xếp lịch thực hiện các nhiệm vụ (*activity selection*), phát biểu như sau:

Chúng ta có rất nhiều nhiệm vụ trong ngày. Giả sử có n nhiệm vụ và nhiệm vụ thứ i phải bắt đầu ngay sau thời điểm s_i , thực hiện liên tục và kết thúc tại thời điểm f_i . Có thể coi mỗi nhiệm vụ tương ứng với một khoảng thời gian thực hiện $(s_i, f_i]$. Hãy chọn ra nhiều nhất các nhiệm vụ để làm, sao cho không có thời điểm nào chúng ta phải làm hai nhiệm vụ cùng lúc, hay nói cách khác, khoảng thời gian thực hiện hai nhiệm vụ bất kỳ là không giao nhau.

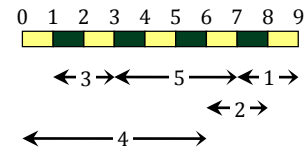
Input

- Dòng 1 chứa số nguyên dương $n \leq 10^5$
- n dòng tiếp theo, dòng thứ i chứa hai số nguyên s_i, f_i ($0 \leq s_i < f_i \leq 10^9$)

Output

Phương án chọn ra nhiều nhiệm vụ nhất để thực hiện

Sample Input	Sample Output
5	Task 3: (1, 3]
7 9	Task 5: (3, 7]
6 8	Task 1: (7, 9]
1 3	Number of selected tasks: 3
0 6	
3 7	



□ Chia để trị

Vì các nhiệm vụ được chọn ra phải thực hiện tuần tự, chúng ta sẽ quan tâm đến nhiệm vụ đầu tiên. Nhiệm vụ đầu tiên có thể là bất kỳ nhiệm vụ nào trong số n nhiệm vụ đã cho.

Nhận xét rằng đã chọn nhiệm vụ thứ i làm nhiệm vụ đầu tiên thì tất cả những nhiệm vụ khác muốn làm sẽ phải làm sau thời điểm f_i . Bài toán trở thành chọn nhiều nhiệm vụ nhất trong số những nhiệm vụ được bắt đầu sau thời điểm f_i (bài toán con).

Vậy thì chúng ta có thuật toán chia để trị: Với tập các nhiệm vụ $M = \{1, 2, \dots, n\}$, thử tất cả n khả năng chọn nhiệm vụ đầu tiên. Với mỗi phép thử lấy nhiệm vụ i làm nhiệm vụ đầu tiên, xác định M_i là tập các nhiệm vụ bắt đầu sau thời điểm f_i và giải bài toán con với tập các nhiệm vụ $M_i \subsetneq M$. Sau n phép thử chọn nhiệm vụ đầu tiên, ta đánh giá n kết quả tìm được và lấy phương án thực hiện được nhiều nhiệm vụ nhất.

□ Phép chọn tham lam

Thuật toán chia để trị phân rã bài toán lớn thành các bài toán con dựa trên phép chọn nhiệm vụ đầu tiên để làm, ta có một quan sát làm cơ sở cho phép chọn tham lam:

Định lý 4-1

Trong số các phương án tối ưu, chắc chắn có một phương án mà nhiệm vụ đầu tiên được chọn là nhiệm vụ kết thúc sớm nhất.

Chứng minh

Gọi i_1 là nhiệm vụ kết thúc sớm nhất. Với một phương án tối ưu bất kỳ, giả sử thứ tự các nhiệm vụ cần thực hiện trong phương án tối ưu đó là (j_1, j_2, \dots, j_k) . Do i_1 là nhiệm vụ kết thúc sớm nhất nên chắc chắn nó không thể kết thúc muộn hơn j_1 , vì vậy việc thay j_1 bởi i_1 trong phương án này sẽ không gây ra sự xung đột nào về thời gian thực hiện các nhiệm vụ. Sự thay thế này cũng không làm giảm bớt số lượng nhiệm vụ thực hiện được trong phương án tối ưu, nên (i_1, j_2, \dots, j_k) cũng sẽ là một phương án tối ưu.

Yêu cầu của bài toán là chỉ cần đưa ra một phương án tối ưu, vì thế ta sẽ chỉ ra phương án tối ưu có nhiệm vụ đầu tiên là nhiệm vụ kết thúc sớm nhất trong số n nhiệm vụ. Điều này có nghĩa là chúng ta không cần thử n khả năng chọn nhiệm vụ đầu tiên, đi giải các bài toán con, rồi mới đánh giá chúng để đưa ra quyết định cuối cùng. Chúng ta sẽ đưa ngay ra *quyết định tức thời*: chọn ngay nhiệm vụ kết thúc sớm nhất i_1 làm nhiệm vụ đầu tiên.

Sau khi chọn nhiệm vụ i_1 , bài toán lớn quy về bài toán con: Chọn nhiều nhiệm vụ nhất trong số các nhiệm vụ được bắt đầu sau khi i_1 kết thúc. Phép chọn tham lam lại cho ta một quyết định tức thời: nhiệm vụ tiếp theo trong phương án tối ưu sẽ là nhiệm vụ bắt đầu sau thời điểm $f[i_1]$ và có thời điểm kết thúc sớm nhất, gọi đó là nhiệm vụ i_2 . Và cứ như vậy chúng ta chọn tiếp các nhiệm vụ i_3, i_4, \dots

□ Cài đặt giải thuật tham lam

Tư tưởng của giải thuật tham lam có thể tóm tắt lại:

Chọn i_1 là nhiệm vụ kết thúc sớm nhất

Chọn i_2 là nhiệm vụ kết thúc sớm nhất bắt đầu sau khi i_1 kết thúc: $s[i_2] \geq f[i_1]$

Chọn i_3 là nhiệm vụ kết thúc sớm nhất bắt đầu sau khi i_2 kết thúc: $s[i_3] \geq f[i_2]$

...

Cứ như vậy cho tới khi không còn nhiệm vụ nào chọn được nữa.

Đến đây ta có thể thiết kế một giải thuật lặp:

- Sắp xếp các nhiệm vụ theo thứ tự không giảm của thời điểm kết thúc $f[.]$
- Khởi tạo thời điểm $FinishTime := 0$
- Duyệt các nhiệm vụ theo danh sách đã sắp xếp (nhiệm vụ kết thúc sớm sẽ được xét trước nhiệm vụ kết thúc muộn), nếu xét đến nhiệm vụ i có $s[i] \geq FinishTime$ thì chọn ngay nhiệm vụ i vào phương án tối ưu và cập nhật $FinishTime$ thành thời điểm kết thúc nhiệm vụ i : $FinishTime := f[i]$



ACTIVITYSELECTION.PAS ✓ Xếp lịch thực hiện các nhiệm vụ

```
{ $MODE OBJFPC }
program ActivitySelection;
const
  max = 100000;
var
  s, f: array[1..max] of Integer;
  id: array[1..max] of Integer;
  n: Integer;

procedure Enter; //Nhập dữ liệu
var
  i: Integer;
begin
  ReadLn(n);
  for i := 1 to n do
    begin
      ReadLn(s[i], f[i]);
      id[i] := i;
    end;
end;

//QuickSort: Sắp xếp bằng chỉ số các công việc id[L]..id[H] tăng theo thời điểm kết thúc
procedure QuickSort(L, H: Integer); //f[id[L]] ≤ f[id[L + 1]] ≤ ... ≤ f[id[H]]
var
  i, j: Integer;
  pivot: Integer;
begin
  if L >= H then Exit;
  i := L + Random(H - L + 1);
  pivot := id[i]; id[i] := id[L];
  i := L; j := H;
  repeat
    while (f[id[j]] > f[pivot]) and (i < j) do Dec(j);
    if i < j then
      begin
        id[i] := id[j]; Inc(i);
      end
    else Break;
    while (f[id[i]] > f[pivot]) and (i < j) do Inc(i);
    if i < j then
      begin
        id[j] := id[i]; Dec(j);
      end
    else Break;
  until i = j;
  id[i] := pivot;
  QuickSort(L, i - 1); QuickSort(i + 1, H);
end;

procedure GreedySelection; //Thuật toán tham lam
var
  i, nTasks: Integer;
  FinishTime: Integer;
begin
  FinishTime := 0;
```

```

nTasks := 0;
for i := 1 to n do //Xét lần lượt các nhiệm vụ id[i] theo thời điểm kết thúc tăng dần
  if s[id[i]] >= FinishTime then //Nếu gặp nhiệm vụ bắt đầu sau FinishTime
    begin
      WriteLn('Task ', id[i], ': (' , s[id[i]], ', ', f[id[i]], ')'); //Chọn tức thì
      Inc(nTasks);
      FinishTime := f[id[i]]; //Cập nhật lại thời điểm kết thúc mới FinishTime
    end;
  WriteLn('Number of selected tasks: ', nTasks);
end;

begin
  Enter;
  QuickSort(1, n);
  GreedySelection;
end.

```

Để thấy rằng thuật toán chọn tham lam ở thủ tục *GreedySelection* được thực hiện trong thời gian $\Theta(n)$. Thời gian mất chủ yếu nằm ở thuật toán sắp xếp các công việc theo thời điểm kết thúc. Như ở ví dụ này chúng ta dùng QuickSort: trung bình $\Theta(n \lg n)$.

4.3.2. Phủ

Trên trục số cho n đoạn: $[a_1, b_1], [a_2, b_2], \dots, [a_n, b_n]$. Hãy chọn một số ít nhất trong số n đoạn đã cho để phủ hết đoạn $[p, q]$.

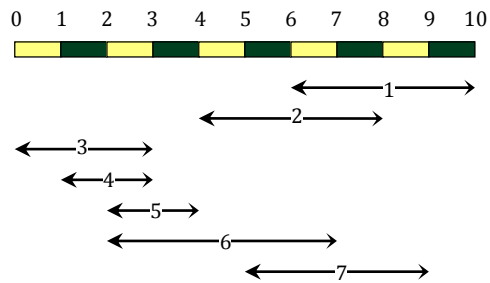
Input

- Dòng 1 chứa ba số nguyên n, p, q ($1 \leq n \leq 10^5, 0 \leq p \leq q \leq 10^9$)
- n dòng tiếp theo, dòng thứ i chứa hai số nguyên a_i, b_i ($0 \leq a_i \leq b_i \leq 10^9$)

Output

Cách chọn ra ít nhất các đoạn để phủ đoạn $[p, q]$

Sample Input	Sample Output
7 1 9	Selected Intervals:
6 10	Interval 3: [0, 3]
4 8	Interval 6: [2, 7]
0 3	Interval 1: [6, 10]
1 3	
2 4	
2 7	
5 9	



□ Chia để trị

Trước hết ta sẽ tìm thuật toán trong trường hợp dữ liệu vào đảm bảo tồn tại phương án phủ đoạn $[p, q]$. Sau đó, việc kiểm tra sự tồn tại của lời giải sẽ được tích hợp vào khi cài đặt thuật toán.

Phương án tối ưu để phủ hết đoạn $[p, q]$ chắc chắn phải chọn một đoạn $[a_i, b_i]$ nào đó để phủ điểm p . Giả sử đoạn $[a_i, b_i]$ chứa điểm p được chọn vào phương án tối ưu, khi đó nếu đoạn này phủ tới cả điểm q , ta có lời giải, còn nếu không, bài toán trở thành phủ đoạn $[b_i, q]$ bằng ít đoạn nhất trong số các đoạn còn lại.

Ta phân rã một bài toán thành nhiều bài toán con tương ứng với mỗi cách chọn đoạn $[a_i, b_i]$ phủ điểm p , giải các bài toán con này và chọn phương án tốt nhất trong tất cả các phương án.

□ Phép chọn tham lam

Để thuận tiện trong việc trình bày thuật toán, với một đoạn $[a_i, b_i]$, ta gọi a_i là cận dưới (lower bound) và b_i là cận trên (upper bound) của đoạn đó.

Thuật toán chia để trị dựa vào sự lựa chọn đoạn phủ điểm p , ta có một nhận xét làm cơ sở cho phép chọn tham lam:

Định lý 4-2

Trong phương án tối ưu sẽ chỉ có một đoạn phủ điểm p . Hơn nữa, nếu có nhiều phương án cùng tối ưu, đoạn có cận trên lớn nhất phủ điểm p chắc chắn được chọn vào một phương án tối ưu nào đó.

Chứng minh

Thật vậy, nếu có phương án chứa hai đoạn phủ điểm p thì nếu ta bỏ đi đoạn có cận trên nhỏ hơn, đoạn $[p, q]$ vẫn được phủ, nên phương án này không tối ưu.

Mặt khác, gọi $[a_{i_1}, b_{i_1}]$ là đoạn có cận trên b_{i_1} lớn nhất phủ điểm p . Với một phương án tối ưu bất kỳ, giả sử rằng phương án đó chọn đoạn $[a_{j_1}, b_{j_1}]$ để phủ điểm p , theo giả thiết, cả hai đoạn $[a_{i_1}, b_{i_1}]$ và $[a_{j_1}, b_{j_1}]$ đều phủ điểm p , nhưng đoạn $[a_{i_1}, b_{i_1}]$ sẽ phủ về bên phải điểm p nhiều hơn đoạn $[a_{j_1}, b_{j_1}]$. Điều này chỉ ra rằng nếu thay thế đoạn $[a_{j_1}, b_{j_1}]$ bởi đoạn $[a_{i_1}, b_{i_1}]$ ta vẫn phủ được cả đoạn $[p, q]$ và không làm ảnh hưởng tới số đoạn được chọn trong phương án tối ưu. Suy ra điều phải chứng minh.

Vậy thì thay vì thử tất cả các khả năng chọn đoạn phủ điểm p , ta có thể đưa ra quyết định tức thì: Chọn ngay đoạn $[a_{i_1}, b_{i_1}]$ có cận trên lớn nhất phủ được điểm p vào phương án tối ưu.

□ Cài đặt giải thuật tham lam

Giải thuật cho bài toán này có thể tóm tắt như sau:

Chọn đoạn $[a_{i_1}, b_{i_1}]$ có b_{i_1} lớn nhất thỏa mãn $p \in [a_{i_1}, b_{i_1}]$. Nếu đoạn này phủ hết đoạn $[p, q]$: $q \leq b_{i_1}$ thì xong. Nếu không, ta lại chọn tiếp đoạn $[a_{i_2}, b_{i_2}]$ có b_{i_2} lớn nhất và phủ được điểm $p_{\text{mới}} = b_{i_1}$, và cứ tiếp tục như vậy, ta có phương án tối ưu:

$$[a_{i_1}, b_{i_1}]; [a_{i_2}, b_{i_2}]; [a_{i_3}, b_{i_3}]; \dots; [a_{i_k}, b_{i_k}]$$

Nhận xét:

- Các đoạn được chọn vào phương án tối ưu có cận trên tăng dần: $b_{i_1} < b_{i_2} < \dots < b_{i_k}$. Điều này có thể dễ hình dung được qua cách chọn: Mỗi khi chọn một đoạn mới, đoạn này sẽ phải phủ qua cận trên của đoạn trước đó.
- Các đoạn được chọn vào phương án tối ưu có cận dưới tăng dần: $a_{i_1} < a_{i_2} < \dots < a_{i_k}$. Thật vậy, nếu cận dưới của các đoạn được chọn không tăng dần thì trong số các đoạn được chọn sẽ có hai đoạn chứa nhau, suy ra phương án này không phải phương án tối ưu.

Đến đây ta có thể thiết kế một giải thuật lặp:

Sắp xếp danh sách các đoạn đã cho theo thứ tự không giảm của cận dưới. Đặt chỉ số đầu danh sách $i := 1$

- Bước 1: Xét phần danh sách bắt đầu từ vị trí i gồm các đoạn có cận dưới $\leq p$. Tìm trong phần này để chọn ra đoạn có cận trên *RightMost* lớn nhất vào phương án tối ưu.
- Bước 2: Nếu *RightMost* $\geq q$, thuật toán kết thúc. Nếu không, đặt $p := \text{RightMost}$, cập nhật i thành chỉ số đứng sau đoạn vừa xét và lặp lại từ bước 1.

(Sau bước 2, chúng ta có thể thoải mái bỏ qua phần đầu danh sách gồm những đoạn đứng trước vị trí i , bởi tất cả những đoạn nằm trong phần này đều có cận trên $\leq \text{RightMost}$ mà như đã nhận xét, đoạn tiếp theo cần chọn chắc chắn phải có cận trên $> \text{RightMost}$)

INTERVALCOVER.PAS ✓ Phủ

```
{ $MODE OBJFPC }
program IntervalCover;
const
  maxN = 100000;
var
  a, b: array[1..maxN] of Integer;
  id: array[1..maxN] of Integer;
  p, q: Integer;
  n: Integer;
  NoSolution: Boolean;

procedure Enter; //Nhập dữ liệu
var
  i: Integer;
begin
  ReadLn(n, p, q);
  for i := 1 to n do
    begin
      ReadLn(a[i], b[i]);
      id[i] := i;
    end;
end;

//Sắp xếp các bảng chỉ số, thứ tự không giảm theo giá trị cận dưới: a[id[L]] ≤ a[id[L + 1]] ≤ ... ≤ a[id[H]]
procedure QuickSort(L, H: Integer);
var
  i, j: Integer;
  pivot: Integer;
```



```

begin
  if L >= H then Exit;
  i := L + Random(H - L + 1);
  pivot := id[i]; id[i] := id[L];
  i := L; j := H;
  repeat
    while (a[id[j]] > a[pivot]) and (i < j) do Dec(j);
    if i < j then
      begin
        id[i] := id[j]; Inc(i);
      end
    else Break;
    while (a[id[i]] < a[pivot]) and (i < j) do Inc(i);
    if i < j then
      begin
        id[j] := id[i]; Dec(j);
      end
    else Break;
  until i = j;
  id[i] := pivot;
  QuickSort(L, i - 1); QuickSort(i + 1, H);
end;

//Phép chọn tham lam
procedure GreedySelection;
var
  i, j: Integer;
  RightMost, k: Integer;
begin
  i := 1;
  RightMost := -1;
  //Danh sách các đoạn theo thứ tự không giảm của cận dưới: id[1], id[2], ..., id[n]
  repeat
    //Duyệt bắt đầu từ id[i], xét các đoạn có cận dưới <= p, chọn ra đoạn id[j] có cận trên RightMost lớn nhất
    j := 0;
    while (i <= n) and (a[id[i]] <= p) do
      begin
        if RightMost < b[id[i]] then
          begin
            RightMost := b[id[i]];
            j := i;
          end;
        Inc(i);
      end;
    if j = 0 then //nếu không chọn được, bài toán vô nghiệm
      begin
        NoSolution := True;
        Exit;
      end;
    id[j] := -id[j]; //Đánh dấu, đoạn id[j] được chọn vào phương án tối ưu
    p := RightMost; //Cập nhật lại p, nếu chưa phủ đến q, tìm tiếp trong danh sách (không tìm lại từ đầu)
  until p >= q;
  NoSolution := False;
end;

procedure PrintResult; //In kết quả
var

```

```

i, j: Integer;
begin
  if NoSolution then
    WriteLn('No Solution!')
  else
    begin
      WriteLn('Selected Intervals: ');
      for i := 1 to n do
        begin
          j := -id[i];
          if j > 0 then //In ra các đoạn được đánh dấu
            WriteLn('Interval ', j, ': [' , a[j], ', ', b[j], ']');
          end;
        end;
      end;
    end;

begin
  Enter;
  QuickSort(1, n);
  GreedySelection;
  PrintResult;
end.

```

Dễ thấy rằng thời gian thực hiện giải thuật chọn tham lam trong thủ tục *GreedySelection* là $\Theta(n)$, chi phí thời gian của thuật toán chủ yếu nằm ở giai đoạn sắp xếp. Trong chương trình này chúng ta dùng QuickSort: Trung bình $\Theta(n \lg n)$.

4.3.3. Mã hóa Huffman

Mã hóa Huffman [25] được sử dụng rộng rãi trong các kỹ thuật nén dữ liệu. Trong đa số các thử nghiệm, thuật toán mã Huffman có thể nén dữ liệu xuống còn từ 10% tới 80% tùy thuộc vào tình trạng dữ liệu gốc.

□ Cơ chế nén dữ liệu

Xét bài toán lưu trữ một dãy dữ liệu gồm n ký tự. Thuật toán Huffman dựa vào tần suất xuất hiện của mỗi phần tử để xây dựng cơ chế biểu diễn mỗi ký tự bằng một dãy bit.

Giả sử dữ liệu là một xâu 100 ký tự $\in \{A, B, C, D, E, F\}$. Tần suất (số lần) xuất hiện của mỗi ký tự cho bởi:

Ký tự: c	A	B	C	D	E	F
Tần suất: $f(c)$	45	13	12	16	9	5

Một cách thông thường là biểu diễn mỗi ký tự bởi một dãy bit chiều dài cố định (như bảng mã ASCII sử dụng 8 bit cho một ký tự AnsiChar hay bảng mã Unicode sử dụng 16 bit cho một ký tự WideChar). Chúng ta có 6 ký tự nên có thể biểu diễn mỗi ký tự bằng một dãy 3 bit:

Ký tự	A	B	C	D	E	F
Mã bit	000	001	010	011	100	101

Vì mỗi ký tự chiếm 3 bit nên để biểu diễn sáu ký tự đã cho sẽ cần $3 \times 6 = 18$ bit. Cách biểu diễn này gọi là biểu diễn bằng từ mã chiều dài cố định (fixed-length codewords).

Một cách khác là biểu diễn mỗi ký tự bởi một dãy bit sao cho ký tự xuất hiện nhiều lần sẽ được biểu diễn bằng dãy bit ngắn trong khi ký tự xuất hiện ít lần hơn sẽ được biểu diễn bằng dãy bit dài:

Ký tự	A	B	C	D	E	F
Mã bit	0	101	100	111	1101	1100

Với cách làm này, sáu ký tự đã cho có thể biểu diễn bằng:

$$1 \times 45 + 3 \times 13 + 3 \times 12 + 3 \times 16 + 4 \times 9 + 4 \times 5 = 224 \text{ bit}$$

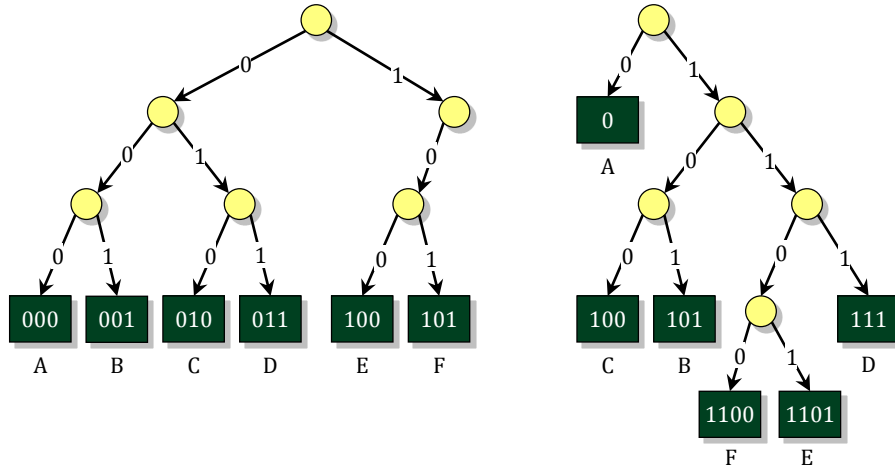
Dữ liệu được nén xuống còn xấp xỉ 75%. Cách biểu diễn này được gọi là biểu diễn bằng từ mã chiều dài thay đổi (variable-length codewords).

□ Mã phi tiền tố

Dãy bit biểu diễn một ký tự gọi là từ mã (codeword) của ký tự đó. Xét tập các từ mã của các ký tự, nếu không tồn tại một từ mã là tiền tố* của một từ mã khác thì tập từ mã này được gọi là mã phi tiền tố (*prefix-free codes* hay còn gọi là *prefix codes*). Hiển nhiên cách biểu diễn bằng từ mã chiều dài cố định là mã phi tiền tố.

Một mã phi tiền tố có thể biểu diễn bằng một cây nhị phân. Trong đó ta gọi nhánh con trái và nhánh con phải của một nút lần lượt là nhánh 0 và nhánh 1. Các nút lá tương ứng với các ký tự và đường đi từ nút gốc tới nút lá sẽ tương ứng với một dãy nhị phân biểu diễn ký tự đó. Hình 4-1 là hai cây nhị phân tương ứng với hai mã tiền tố chiều dài cố định và chiều dài thay đổi như ở ví dụ trên.

* Một chuỗi ký tự X được gọi là tiền tố của chuỗi ký tự Y nếu $\exists s: Y = X + s$



Hình 4-1. Cây biểu diễn mã tiền tố

Một chuỗi ký tự S (bản gốc) sẽ được mã hóa (nén) theo cách thức: Viết các từ mã của từng chữ cái nối tiếp nhau tạo thành một dãy bit. Dãy bit này gọi là bản nén của bản gốc S . Trong ví dụ này chuỗi "AABE" sẽ được nén thành dãy bit "001011101" nếu sử dụng mã tiền tố chiều dài thay đổi như cây bên phải Hình 4-1.

Tính chất phi tiền tố đảm bảo rằng với một từ mã, nếu xuất phát từ gốc cây, đọc từ mã từ trái qua phải và mỗi khi đọc một bit ta rẽ sang nhánh con tương ứng thì khi đọc xong từ mã, ta sẽ đứng ở một nút lá. Chính vì vậy việc giải mã (giải nén) sẽ được thực hiện theo thuật toán: Bắt đầu từ nút gốc và đọc bản nén, đọc được bit nào rẽ sang nhánh con tương ứng. Mỗi khi đến được nút lá, ta xuất ra ký tự tương ứng và quay trở về nút gốc để đọc tiếp cho tới hết.

Nếu ký hiệu C là tập các ký tự, mỗi ký tự $c \in C$ được biểu diễn bằng từ mã gồm $d_T(c)$ bit thì số bit trong bản nén là:

$$B(T) = \sum_{c \in C} d_T(c) f(c) \tag{4.1}$$

Trong đó $f(c)$ là số lần xuất hiện ký tự c trong chuỗi ký tự gốc và $d_T(c)$ cũng là độ sâu của nút lá tương ứng với ký tự c . $B(T)$ gọi là chi phí của cây T .

Kích thước bản nén phụ thuộc vào cấu trúc cây nhị phân tương ứng với mã phi tiền tố được sử dụng. Bài toán đặt ra là tìm mã phi tiền tố để nén bản gốc thành bản nén gồm ít bit nhất. Điều này tương đương với việc tìm cây T có $B(T)$ nhỏ nhất tương ứng với một bản gốc là chuỗi ký tự đầu vào S .

□ Thuật toán Huffman

Giả sử các nút trên cây nhị phân chứa bên trong các thông tin:

- Ký tự tương ứng với nút đó nếu là nút lá (c)

- Tần suất của nút (f). Tần suất của một nút là số lần duyệt qua nút đó khi giải nén. Dễ thấy rằng tần suất nút lá chính là tần suất của ký tự tương ứng trong bản gốc và tần suất của một nút nhánh bằng tổng tần suất của hai nút con.
- Hai liên kết trái, phải (l và r)

Xét danh sách ban đầu gồm tất cả các nút lá của cây. Thuật toán Huffman làm việc theo cách: Lấy từ danh sách ra hai nút x, y có tần suất thấp nhất. Tạo ra một nút z làm nút cha của cả hai nút x và y , tần suất của nút z được gán bằng tổng tần suất hai nút x, y , sau đó z được đẩy vào danh sách (ra hai vào một). Thuật toán sẽ kết thúc khi danh sách chỉ còn một nút (tương ứng với nút gốc của cây được xây dựng). Có thể hình dung tại mỗi bước, thuật toán quản lý một rừng các cây và tìm cách nhập hai cây lại cho tới khi rừng chỉ còn một cây.

Danh sách các nút trong thuật toán Huffman được tổ chức dưới dạng hàng đợi ưu tiên. Trong đó nút x được gọi là ưu tiên hơn nút y nếu tần suất nút x nhỏ hơn tần suất nút y : $x.f < y.f$

Hai thao tác trên hàng đợi ưu tiên được sử dụng là:

- Hàm *Extract*: Lấy nút có tần suất nhỏ nhất ra khỏi hàng đợi ưu tiên, trả về trong kết quả hàm
- Thủ tục *Insert(z)*: Thêm một nút mới (z) vào hàng đợi ưu tiên

Khi đó có thể viết cụ thể hơn thuật toán Huffman:

```
H := ∅; //Khởi tạo hàng đợi ưu tiên H
for ∀c∈C do
  begin
    «Tạo ra một nút mới node có tần suất f(c) và chứa ký tự c»;
    Insert(node);
  end;
while |H| > 1 do
  begin
    x := Extract; y := Extract; //Lấy từ hàng đợi ưu tiên ra hai nút có tần suất nhỏ nhất
    «Tạo ra một nút mới z»;
    z.f := x.f + y.f; //z có tần suất bằng tổng tần suất hai nút x, y
    z.l := x; z.r := y; //cho x và y làm con trái và con phải của z
    Insert(z); //Đẩy z vào hàng đợi ưu tiên
  end;
```

Phép lựa chọn tham lam trong thuật toán Huffman thể hiện ở sự quyết định tức thì tại mỗi bước: Chọn hai cây có tần suất nhỏ nhất để nhập vào thành một cây. Tính đúng đắn của thuật toán Huffman được chứng minh như sau:

Bổ đề 4-3

Cây tương ứng với mã phi tiền tố tối ưu phải là cây nhị phân đầy đủ. Tức là mọi nút nhánh của nó phải có đúng hai nút con.

Chứng minh

Thật vậy, nếu một mã tiền tố tương ứng với cây nhị phân T không đầy đủ thì sẽ tồn tại một nút nhánh p chỉ có một nút con q . Xóa bỏ nút nhánh p và đưa nhánh con gốc q của nó vào thế chỗ, ta được một cây mới T' mà các ký tự vẫn chỉ nằm ở lá, độ sâu của mọi lá trong nhánh cây gốc q giảm đi 1 còn độ sâu của các lá khác được giữ nguyên. Tức là T' sẽ biểu diễn một mã phi tiền tố nhưng với chi phí thấp hơn T . Vậy T không tối ưu.

Bổ đề 4-4

Gọi x và y là hai ký tự có tần suất nhỏ nhất. Khi đó tồn tại một cây tối ưu sao cho hai nút lá chứa x và y là con của cùng một nút. Hay nói cách khác, hai nút lá x, y là anh-em (siblings).

Chứng minh

Tần suất của nút lá bằng tần suất của ký tự chứa trong nên sẽ bất biến trên mọi cây. Để tiện trình bày ta đồng nhất nút lá với ký tự chứa trong nó.

Với T là một cây tối ưu bất kỳ, lấy một nút nhánh sâu nhất và gọi a và b là hai nút con của nó. Dễ thấy rằng a và b phải là nút lá. Không giảm tính tổng quát, giả sử rằng $f(a) \leq f(b)$ và $f(x) \leq f(y)$.

Đảo nút a và nút x cho nhau, ta được một cây mới T' mà sự khác biệt về chi phí của T và T' chỉ khác nhau ở hai nút a và x . Tức là nếu xét về mức chênh lệch chi phí:

$$\begin{aligned} B(T) - B(T') &= [f(x)d_T(x) + f(a)d_T(a)] - [f(x)d_{T'}(x) + f(a)d_{T'}(a)] \\ &= f(x)d_T(x) + f(a)d_T(a) - f(x)d_T(a) - f(a)d_T(x) \\ &= (f(a) - f(x))(d_T(a) - d_T(x)) \\ &\geq 0 \end{aligned} \tag{4.2}$$

(Bởi x là nút có tần suất thấp nhất nên $f(a) - f(x) \geq 0$, đồng thời a là nút lá sâu nhất nên $d_T(a) - d_T(x) \geq 0$)

Điều này chỉ ra rằng nếu đảo hai nút a và x , ta sẽ được cây T' ít ra là không tệ hơn cây T . Tương tự như vậy, nếu ta đảo tiếp hai nút b và y trên cây T' , ta sẽ được cây T'' ít ra là không tệ hơn cây T' . Từ cây T tối ưu, suy ra T'' tối ưu, và trên T'' thì x và y là hai nút con của cùng một nút (ĐPCM).

Định lý 4-5

Gọi C là tập các ký tự trong xâu ký tự đầu vào S và x, y là hai ký tự có tần suất thấp nhất. Gọi C' là tập các ký tự có được từ C bằng cách thay hai ký tự x, y bởi một ký tự z : $C' = C - \{x, y\} \cup \{z\}$ với tần suất $f(z) := f(x) + f(y)$. Gọi T' là cây tối ưu trên C' , khi đó nếu cho nút lá z trở thành nút nhánh với hai nút con x, y , ta sẽ được cây T là cây tối ưu trên C .

Chứng minh

Ta có

$$\begin{aligned} f(x)d_T(x) + f(y)d_T(y) &= (f(x) + f(y))(d_{T'}(z) + 1) \\ &= f(z)d_{T'}(z) + (f(x) + f(y)) \end{aligned} \quad (4.3)$$

Từ đó suy ra $B(T) = B(T') + f(x) + f(y)$ hay $B(T') = B(T) - f(x) - f(y)$.

Giả sử phản chứng rằng T không tối ưu, gọi \hat{T} là cây tối ưu trên tập ký tự C . Xét cây \hat{T} , không giảm tính tổng quát, có thể coi x, y là anh-em (Bổ đề 4-4). Đưa ký tự z vào nút cha chung của x và y với tần suất $f(z) := f(x) + f(y)$ rồi cắt bỏ hai nút lá x, y trên \hat{T} . Ta sẽ được cây mới \hat{T}' tương ứng với một mã phi tiền tố trên C' . Trong đó:

$$\begin{aligned} B(\hat{T}') &= B(\hat{T}) - f(x) - f(y) \\ &< B(T) - f(x) - f(y) \text{ (do } T \text{ không tối ưu)} \\ &= B(T') \end{aligned} \quad (4.4)$$

Vậy $B(\hat{T}') < B(T')$, mâu thuẫn với giả thiết T' là tối ưu trên tập ký tự C' . Ta có điều phải chứng minh.

Tính đúng đắn của thuật toán Huffman được suy ra trực tiếp từ Định lý 4-5: Để tìm cây tối ưu trên tập ký tự C , ta tìm hai ký tự có tần suất thấp nhất x, y và nhập chúng lại thành một ký tự z . Sau khi thiết lập quan hệ cha-con giữa z và x, y , ta quy về bài toán con: Tìm cây tối ưu trên tập $C' := C - \{x, y\} \cup \{z\}$ có lực lượng ít hơn C một phần tử.

□ Cài đặt

Chúng ta sẽ cài đặt chương trình tìm mã phi tiền tố tối ưu để mã hóa các ký tự trong một xâu ký tự đầu vào S .

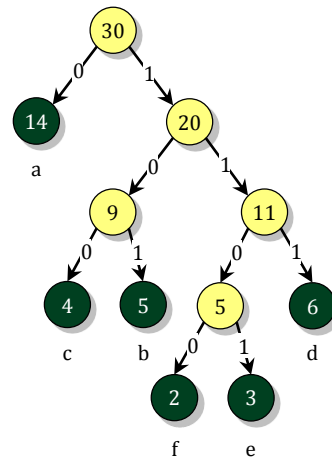
Input

Xâu ký tự S

Output

Các từ mã tương ứng với các ký tự.

Sample Input	Sample Output
abcdefabcdefabcdefabcdabdadaaaaaaaaa	0 = a 100 = c 101 = b 1100 = f 1101 = e 111 = d



Hàng đợi ưu tiên của các nút trong chương trình được tổ chức dưới dạng Binary Heap. Trên đó ta cài đặt một thao tác $Heapify(r)$ để vun nhánh Heap gốc r thành đồng trong điều kiện hai nhánh con của nó (gốc $2r$ và $2r + 1$) đã là đồng rồi. Tại mỗi bước của thuật toán Huffman, thay vì cài đặt cơ chế “ra hai vào một”, ta sẽ sử dụng một mẹo nhỏ: Lấy khỏi Heap (Pop) một nút x , đọc nút y ở gốc Heap (Get), tạo nút z là cha chung của x và y , đưa nút z vào gốc Heap đề lên nút y rồi thực hiện vun đồng từ gốc: $Heapify(1)$.

HUFFMAN.PAS ✓ Mã hóa Huffman

```
{ $MODE OBJFPC }
program HuffmanCode;
const
  MaxLeaves = 256; // Các ký tự là kiểu AnsiChar, cần đổi kích thước nếu dùng kiểu dữ liệu khác
type
  TNode = record // Cấu trúc nút trên cây Huffman
    f: Integer;
    c: AnsiChar;
    l, r: Pointer;
  end;
  PNode = ^TNode; // Kiểu con trỏ tới nút
  THeap = record // Cấu trúc dữ liệu Binary Heap
    items: array[1..MaxLeaves] of PNode;
    nItems: Integer;
  end;
var
  s: AnsiString; // Xâu ký tự đầu vào
  Heap: THeap; // Hàng đợi ưu tiên
  TreeRoot: PNode; // Gốc cây Huffman
  bits: array[1..MaxLeaves] of Integer;

// Tạo ra một nút mới chứa tần suất fq, ký tự ch, và hai nhánh con left, right
function NewNode(fq: Integer; ch: AnsiChar; left, right: PNode): PNode;
begin
  New(Result);
  with Result^ do
```



```

begin
  f:= fq;
  c := ch;
  l := left;
  r := right;
end;
end;

//Định nghĩa quan hệ ưu tiên hơn là quan hệ <: nút x ưu tiên hơn nút y nếu tần suất của x nhỏ hơn
operator < (const x, y: TNode): Boolean;
begin
  Result := x.f < y.f;
end;

//Vuốt nhánh Heap gốc r thành đống
procedure Heapify(r: Integer);
var
  c: Integer;
  temp: PNode;
begin
  with Heap do
    begin
      temp := items[r];
      repeat
        c := r * 2;
        if (c < nItems) and (items[c + 1]^ < items[c]^) then
          Inc(c);
        if (c > nItems) or not (items[c]^ < temp^) then Break;
        items[r] := items[c];
        r := c;
      until False;
      items[r] := temp;
    end;
  end;
end;

//Đọc nút ở gốc Heap
function Get: PNode;
begin
  with Heap do Result := items[1];
end;

//Lấy nút có tần suất nhỏ nhất ra khỏi Heap
function Pop: PNode;
begin
  with Heap do
    begin
      Result := items[1];
      items[1] := items[nItems];
      Dec(nItems);
      Heapify(1);
    end;
  end;
end;

//Thay nút ở gốc Heap bởi node
procedure UpdateRootHeap(node: PNode);
begin
  with Heap do

```

```

begin
  items[1] := node;
  Heapify(1);
end;
end;

procedure InitHeap; //Khởi tạo Heap ban đầu chứa các nút lá
var
  c: AnsiChar;
  i: Integer;
  freq: array[AnsiChar] of Integer;
begin
  FillDWord(freq, Length(freq), 0);
  for i := 1 to Length(s) do Inc(freq[s[i]]); //Đếm tần suất
  with Heap do
    begin
      nItems := 0;
      for c := Low(AnsiChar) to High(AnsiChar) do
        if freq[c] > 0 then //Xét các ký tự trong S
          begin
            Inc(nItems);
            items[nItems] := NewNode(freq[c], c, nil, nil); //Tạo nút chứa ký tự
          end;
        for i := nItems div 2 downto 1 do //Vun đống từ dưới lên
          Heapify(i);
        end;
      end;
    end;
end;

procedure BuildTree; //Thuật toán Huffman
var
  x, y, z: PNode;
begin
  while Heap.nItems > 1 do //Chừng nào Heap còn nhiều hơn 1 phần tử
    begin
      x := Pop; //Lấy nút tần suất nhỏ nhất khỏi Heap
      y := Get; //Đọc nút tần suất nhỏ nhất tiếp theo ở gốc Heap
      z := NewNode(x^.f + y^.f, #0, x, y); //Tạo nút z là cha của x và y
      UpdateRootHeap(z); //Đưa z vào gốc Heap và thực hiện vun đống: ↔Ra x và y, vào z
    end;
  TreeRoot := Heap.items[1]; //Giữ lại gốc cây Huffman
end;

//Thủ tục này in ra các từ mã của các lá trong cây Huffman gốc node, depth là độ sâu của node
procedure Traversal(node: PNode; depth: Integer); //Duyệt cây Huffman gốc node
var
  i: Integer;
begin
  if node^.l = nil then //Nếu node là nút lá
    begin //In ra dãy bit biểu diễn ký tự
      for i := 1 to depth do Write(bits[i]);
      WriteLn(' = ', node^.c);
      Dispose(node);
    end
  else //Nếu node là nút nhánh
    begin
      bits[depth + 1] := 0; //Các từ mã của các lá trong nhánh con trái sẽ có bit tiếp theo là 0
    end;
  end;
end;

```

```

    Traversal(node^.l, depth + 1); //Duyệt nhánh trái
    bits[depth + 1] := 1; //Các từ mã của các lá trong nhánh con phải sẽ có bit tiếp theo là 1
    Traversal(node^.r, depth + 1); //Duyệt nhánh phải
end;
end;

begin
  ReadLn(s);
  InitHeap;
  BuildTree;
  Traversal(TreeRoot, 0);
end.

```

Xét riêng thuật toán Huffman ở thủ tục BuildTree. Ta thấy rằng nếu n là số ký tự (số nút lá trong cây Huffman) thì vòng lặp while sẽ lặp $n - 1$ lần. Các lời gọi thủ tục bên trong vòng lặp đó có thời gian thực hiện $O(\lg n)$. Vậy thuật toán Huffman có thời gian thực hiện $O(n \lg n)$.

Chúng ta đã khảo sát vài bài toán mà ở đó, thuật toán tham lam được thiết kế dựa trên một mô hình chia để trị.

Thuật toán tham lam cần đưa ra quyết định tức thì tại mỗi bước lựa chọn. Quyết định này sẽ ảnh hưởng ngay tới sự lựa chọn ở bước kế tiếp. Chính vì vậy, đôi khi phân tích kỹ hai quyết định liên tiếp có thể cho ta những tính chất của nghiệm tối ưu và từ đó có thể xây dựng một thuật toán hiệu quả.

4.3.4. Lịch xử lý lỗi

Bạn là một người quản trị một hệ thống thông tin, và trong hệ thống cùng lúc đang xảy ra n lỗi đánh số từ 1 tới n . Lỗi i sẽ gây ra thiệt hại d_i sau mỗi ngày và để khắc phục lỗi đó cần t_i ngày. Tại một thời điểm, bạn chỉ có thể xử lý một lỗi. Hãy lên lịch trình xử lý lỗi sao cho tổng thiệt hại của n lỗi là nhỏ nhất có thể.

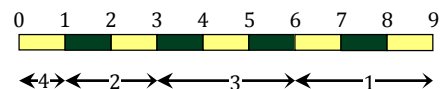
Input

- Dòng 1 chứa số nguyên dương $n \leq 10^5$
- n dòng tiếp theo, mỗi dòng chứa hai số nguyên dương $d_i, t_i \leq 100$

Output

Lịch xử lý lỗi

Sample Input	Sample Output
4	Bug 4: fixed time = 1; damage = 2
1 3	Bug 2: fixed time = 3; damage = 9
3 2	Bug 3: fixed time = 6; damage = 24
4 3	Bug 1: fixed time = 9; damage = 9
2 1	Minimum Damage = 44



Do mỗi lỗi chỉ ngưng gây thiệt hại khi nó được khắc phục, nên dễ thấy rằng lịch trình cần tìm sẽ phải có tính chất: Khi bắt tay vào xử lý một lỗi, ta sẽ phải làm liên tục cho tới khi lỗi đó được khắc phục. Tức là ta cần tìm một hoán vị của dãy số $(1, 2, \dots, n)$ tương ứng với thứ tự trong lịch trình khắc phục các lỗi.

Giả sử rằng lịch trình $A = (1, 2, \dots, i, i + 1, \dots, n)$ xử lý lần lượt các lỗi từ 1 tới n là lịch trình tối ưu. Ta lấy một lịch trình khác $B = (1, 2, \dots, i + 1, i, \dots, n)$ có được bằng cách đảo thứ tự xử lý hai lỗi liên tiếp: i và $i + 1$ trên lịch trình A

Ta nhận xét rằng ngoài hai lỗi i và $i + 1$, thời điểm các lỗi khác được khắc phục là giống nhau trên hai lịch trình, có nghĩa là sự khác biệt về tổng thiệt hại của lịch trình A và lịch trình B chỉ nằm ở thiệt hại do hai lỗi i và $i + 1$ gây ra.

Gọi T là thời điểm lỗi $i - 1$ được khắc phục (trên cả hai lịch trình A và B). Khi đó:

Nếu theo lịch trình $A = (1, 2, \dots, i, i + 1, \dots, n)$, thiệt hại của hai lỗi i và $i + 1$ gây ra là:

$$\alpha = (T + t_i)d_i + (T + t_i + t_{i+1})d_{i+1} \quad (4.5)$$

Nếu theo lịch trình $B = (1, 2, \dots, i + 1, i, \dots, n)$, thiệt hại của hai lỗi i và $i + 1$ gây ra là:

$$\beta = (T + t_{i+1} + t_i)d_i + (T + t_{i+1})d_{i+1} \quad (4.6)$$

Thiệt hại theo lịch trình A không thể lớn hơn thiệt hại theo lịch trình B (do A tối ưu), tức là thiệt hại α sẽ không lớn hơn thiệt hại β . Loại bỏ những hạng tử giống nhau ở công thức tính α và β , ta có:

$$\begin{aligned} \alpha \leq \beta &\Leftrightarrow t_i d_{i+1} \leq t_{i+1} d_i \\ \alpha \leq \beta &\Leftrightarrow \frac{t_i}{d_i} \leq \frac{t_{i+1}}{d_{i+1}} \end{aligned} \quad (4.7)$$

Vậy thì nếu lịch trình $A = (1, 2, \dots, n)$ là tối ưu, nó sẽ phải thỏa mãn:

$$\frac{t_1}{d_1} \leq \frac{t_2}{d_2} \leq \dots \leq \frac{t_n}{d_n}$$

Ngoài ra có thể thấy rằng nếu $\frac{t_i}{d_i} = \frac{t_{i+1}}{d_{i+1}}$ thì $\alpha = \beta$, hai lịch trình sửa chữa A, B có cùng mức độ thiệt hại. Trong trường hợp này, việc sửa lỗi i trước hay $i + 1$ trước đều cho các phương án cùng tối ưu. Từ đó suy ra phương án tối ưu cần tìm là phương án khắc phục các lỗi theo thứ tự tăng dần của tỉ số $\frac{t_i}{d_i}$. Lời giải lặp đơn thuần chỉ là một thuật toán sắp xếp.

BUGFIXSCHEDULING.PAS ✓ Lịch xử lý lỗi

```
{ $MODE OBJFPC }
program BugFixes;
const
  max = 100000;
type
  TBug = record
    d, t: Integer;
```

```

    end;
var
    bugs: array[1..max] of TBug;
    id: array[1..max] of Integer;
    n: Integer;

procedure Enter; //Nhập dữ liệu
var
    i: Integer;
begin
    ReadLn(n);
    for i := 1 to n do
        with bugs[i] do
            begin
                ReadLn(d, t);
                id[i] := i;
            end;
    end;

//Định nghĩa lại toán tử < trên các lỗi. Lỗi x gọi là "nhỏ hơn" lỗi y nếu x.t/x.d < y.t/y.d
operator < (const x, y: TBug): Boolean;
begin
    Result := x.t * y.d < y.t * x.d;
end;

//Sắp xếp các lỗi bugs[L...H] theo quan hệ thứ tự "nhỏ hơn" định nghĩa ở trên
procedure QuickSort(L, H: Integer);
var
    i, j: Integer;
    pivot: Integer;
begin
    if L >= H then Exit;
    i := L + Random(H - L + 1);
    pivot := id[i]; id[i] := id[L];
    i := L; j := H;
    repeat
        while (bugs[pivot] < bugs[id[j]]) and (i < j) do Dec(j);
        if i < j then
            begin
                id[i] := id[j]; Inc(i);
            end
        else Break;
        while (bugs[id[i]] < bugs[pivot]) and (i < j) do Inc(i);
        if i < j then
            begin
                id[j] := id[i]; Dec(j);
            end
        else Break;
    until i = j;
    id[i] := pivot;
    QuickSort(L, i - 1); QuickSort(i + 1, H);
end;

procedure PrintResult; //In kết quả
var
    i: Integer;
    Time, Damage: Integer;

```

```

TotalDamage: Int64;
begin
  Time := 0;
  TotalDamage := 0;
  for i := 1 to n do
    with bugs[id[i]] do
      begin
        Time := Time + t;
        Damage := Time * d;
        WriteLn('Bug ', id[i], ': fixed time = ', Time, '; damage = ', Damage);
        Inc(TotalDamage, Damage);
      end;
  WriteLn('Minimum Damage = ', TotalDamage);
end;

begin
  Enter;
  QuickSort(1, n);
  PrintResult;
end.

```

4.3.5. Thuật toán Johnson

□ Bài toán

Bài toán lập lịch gia công trên hai máy (*Two-machine flow shop model*): Có n chi tiết đánh số từ 1 tới n và hai máy A, B . Một chi tiết cần gia công trên máy A trước rồi chuyển sang gia công trên máy B . Thời gian gia công chi tiết i trên máy A và B lần lượt là a_i và b_i . Tại một thời điểm, mỗi máy chỉ có thể gia công một chi tiết. Hãy lập lịch gia công các chi tiết sao cho việc gia công toàn bộ n chi tiết được hoàn thành trong thời gian sớm nhất.

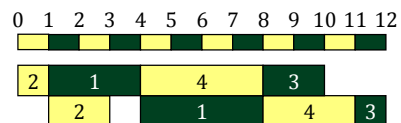
Input

- Dòng 1 chứa số nguyên dương $n \leq 10^5$
- Dòng 2 chứa n số nguyên dương a_1, a_2, \dots, a_n ($\forall i: a_i \leq 100$)
- Dòng 3 chứa n số nguyên dương b_1, b_2, \dots, b_n ($\forall i: b_i \leq 100$)

Output

Lịch gia công tối ưu

Sample Input	Sample Output
4	Job 2: A[0, 1]; B[1, 3]
3 1 2 4	Job 1: A[1, 4]; B[4, 8]
4 2 1 3	Job 4: A[4, 8]; B[8, 11]
	Job 3: A[8, 10]; B[11, 12]
	Time = 12



□ Thuật toán

Nhận xét rằng luôn tồn tại một lịch gia công tối ưu sao mà các chi tiết gia công trên máy A theo thứ tự như thế nào thì sẽ được gia công trên máy B theo thứ tự đúng như vậy. Máy A sẽ hoạt động liên tục không nghỉ còn máy B có thể có những khoảng thời gian chết khi chờ chi tiết từ máy A . Như vậy ta cần tìm một lịch gia công tối ưu dưới dạng một hoán vị của dãy số $(1, 2, \dots, n)$.

Định lý 4-6 (Định lý Johnson)

Một lịch gia công tối ưu cần thỏa mãn tính chất: Nếu chi tiết i được gia công trước chi tiết j thì $\min(a_i, b_j) \leq \min(a_j, b_i)$. Hơn nữa nếu $\min(a_i, b_j) = \min(a_j, b_i)$ thì việc đảo thứ tự gia công chi tiết i và chi tiết j trong phương án tối ưu vẫn sẽ duy trì được tính tối ưu của phương án.

Định lý 4-7

Xét quan hệ hai ngôi “nhỏ hơn hoặc bằng” (ký hiệu \leq) xác định trên tập các chi tiết. Quan hệ này định nghĩa như sau: $i \leq j$ nếu:

- Hoặc $\min(a_i, b_j) < \min(a_j, b_i)$
- Hoặc $\min(a_i, b_j) = \min(a_j, b_i)$ và $i \leq j$

Khi đó quan hệ \leq là một quan hệ thứ tự toàn phần (có đầy đủ các tính chất: phổ biến, phản xạ, phản đối xứng, và bắc cầu)

Việc chứng minh cụ thể hai định lý trên khá dài dòng, các bạn có thể tham khảo trong các tài liệu khác.

□ Cài đặt

Định lý 4-6 (Định lý Johnson) và Định lý 4-7 chỉ ra rằng thuật toán Johnson có thể cài đặt bằng một thuật toán sắp xếp so sánh. Tuy nhiên khi cài đặt thuật toán sắp xếp so sánh, chúng ta chỉ cần cài đặt phép toán $i < j$: cho biết chi tiết i bắt buộc phải gia công trước chi tiết j bằng việc kiểm tra bất đẳng thức $\min(a_i, b_j) < \min(a_j, b_i)$. Bởi khi $\min(a_i, b_j) = \min(a_j, b_i)$, gia công chi tiết i trước hay j trước đều được

JOHNSONALG.PAS ✓ Thuật toán Johnson

```
{ $MODE OBJFPC }
program JohnsonAlgorithm;
uses Math;
const
  maxN = 100000;
type
  TJob = record
    a, b: Integer;
  end;
var
  jobs: array[1..maxN] of TJob;
  id: array[1..maxN] of Integer;
  n: Integer;
```

```

procedure Enter; //Nhập dữ liệu
var
  i: Integer;
begin
  ReadLn(n);
  for i := 1 to n do Read(jobs[i].a);
  ReadLn;
  for i := 1 to n do Read(jobs[i].b);
  for i := 1 to n do id[i] := i;
end;

//Định nghĩa toán tử "phải làm trước": < Chi tiết x phải làm trước chi tiết y nếu Min(x.a, y.b) < Min(y.a, x.b)
operator < (const x, y: TJob): Boolean; //Toán tử này dùng cho sắp xếp
begin
  Result := Min(x.a, y.b) < Min(y.a, x.b);
end;

//Thuật toán QuickSort sắp xếp bằng chỉ số
procedure QuickSort(L, H: Integer);
var
  i, j: Integer;
  pivot: Integer;
begin
  if L >= H then Exit;
  i := L + Random(H - L + 1);
  pivot := id[i]; id[i] := id[L];
  i := L; j := H;
  repeat
    while (jobs[pivot] < jobs[id[j]]) and (i < j) do Dec(j);
    if i < j then
      begin
        id[i] := id[j]; Inc(i);
      end
    else Break;
    while (jobs[id[i]] < jobs[pivot]) and (i < j) do Inc(i);
    if i < j then
      begin
        id[j] := id[i]; Dec(j);
      end;
    else Break;
  until i = j;
  id[i] := pivot;
  QuickSort(L, i - 1); QuickSort(i + 1, H);
end;

procedure PrintResult; //In kết quả
var
  StartA, StartB, FinishA, FinishB: Integer;
  i, j: Integer;
begin
  StartA := 0;
  StartB := 0;
  for i := 1 to n do //Duyệt danh sách đã sắp xếp
    begin
      j := id[i];
      FinishA := StartA + jobs[j].a;

```



```

    StartB := Max(StartB, FinishA);
    FinishB := StartB + jobs[j].b;
    WriteLn('Job ', j, ': ',
           'A[' , StartA, ', ', ', FinishA, ']; B[' , StartB, ', ', ', FinishB, ']');
    StartA := FinishA;
    StartB := FinishB;
end;
WriteLn('Time = ', FinishB);
end;

begin
    Enter;
    QuickSort(1, n);
    PrintResult;
end.

```

Thời gian thực hiện thuật toán Johnson cài đặt theo cách này có thể đánh giá bằng thời gian thực hiện giải thuật sắp xếp. Ở đây ta dùng QuickSort (Trung bình $\Theta(n \lg n)$). Có thể cài đặt thuật toán Johnson theo một cách khác để tận dụng các thuật toán sắp xếp dãy khóa số, cách làm như sau:

Chia các chi tiết làm hai nhóm: Nhóm SA gồm các chi tiết i có $a_i < b_i$ và nhóm SB gồm các chi tiết j có $a_j \geq b_j$. Sắp xếp các chi tiết trong nhóm SA theo thứ tự tăng dần của các a_i , sắp xếp các chi tiết trong nhóm SB theo thứ tự giảm dần của các b_i rồi nối hai danh sách đã sắp xếp lại.

Bài tập 4-1.

Bạn là người lập lịch giảng dạy cho n chuyên đề, chuyên đề thứ i cần bắt đầu ngay sau thời điểm s_i và kết thúc tại thời điểm f_i : $(s_i, f_i]$. Mỗi chuyên đề trong thời gian diễn gia hoạt động giảng dạy sẽ cần một phòng học riêng. Hãy xếp lịch giảng dạy sao cho số phòng học phải sử dụng là ít nhất. Tìm thuật toán $O(n \lg n)$.

Bài tập 4-2.

Trên trục số cho n điểm đen và n điểm trắng hoàn toàn phân biệt. Hãy tìm n đoạn, mỗi đoạn nối một điểm đen với một điểm trắng, sao cho không có hai đoạn thẳng nào có chung đầu mút và tổng độ dài n đoạn là nhỏ nhất có thể. Tìm thuật toán $O(n \lg n)$.

Bài tập 4-3.

Xét mã phi tiền tố của một tập n ký tự. Nếu các ký tự được sắp xếp sẵn theo thứ tự không giảm của tần suất thì chúng ta có thể xây dựng cây Huffman trong thời gian $O(n)$ bằng cách sử dụng hai hàng đợi: Tạo ra các nút lá chứa ký tự và đẩy chúng vào hàng đợi 1 theo thứ tự từ nút tần suất thấp nhất tới nút tần suất cao nhất. Khởi tạo hàng đợi 2 rỗng, lặp lại các thao tác sau $n - 1$ lần:

- Lấy hai nút x, y có tần suất thấp nhất ra khỏi các hàng đợi bằng cách đánh giá các phần tử ở đầu của cả hai hàng đợi
- Tạo ra một nút z làm nút cha của hai nút x, y với tần suất bằng tổng tần suất của x và y

- Đẩy z vào cuối hàng đợi 2

Chúng minh tính đúng đắn và cài đặt thuật toán trên.

Bài tập 4-4.

Bài toán xếp ba lô (Knapsack) chúng ta đã khảo sát trong chuyên đề kỹ thuật nhánh cận và quy hoạch động là bài toán 0/1 Knapsack: Với một sản phẩm chỉ có hai trạng thái: chọn hay không chọn. Chúng ta cũng đã khảo sát bài toán Fractional Knapsack: Cho phép chọn một phần sản phẩm: Với một sản phẩm trọng lượng w và giá trị v , nếu ta lấy một phần trọng lượng $w' \leq w$ của sản phẩm đó thì sẽ được giá trị là $v * \frac{w'}{w}$. Chỉ ra rằng hàm *Estimate* trong mục 1.4.3 cho kết quả tối ưu đối với bài toán Fractional Knapsack.

Bài tập 4-5.

Giáo sư X lái xe trong một chuyến hành trình “Xuyên Việt” từ Cao Bằng tới Cà Mau dài l km. Dọc trên đường đi có n trạm xăng và trạm xăng thứ i cách Cao Bằng d_i km. Bình xăng của xe khi đổ đầy có thể đi được m km. Hãy xác định các điểm dừng để đổ xăng tại các trạm xăng sao cho số lần phải dừng đổ xăng là ít nhất trong cả hành trình.

Bài tập 4-6.

Cho n điểm thực trên trục số: $x_1, x_2, \dots, x_n \in \mathbb{R}$. Hãy chỉ ra ít nhất các đoạn dạng $[i, i + 1]$ với i là số nguyên để phủ hết n điểm đã cho.

Bài tập 4-7.

Bạn có n nhiệm vụ, mỗi nhiệm vụ cần làm trong một đơn vị thời gian. Nhiệm vụ thứ i có thời điểm phải hoàn thành (deadline) là d_i và nếu bạn hoàn thành nhiệm vụ đó sau thời hạn d_i thì sẽ phải mất một khoản phạt p_i . Bạn bắt đầu từ thời điểm 0 và tại mỗi thời điểm chỉ có thể thực hiện một nhiệm vụ. Hãy lên lịch thực hiện các nhiệm vụ sao cho tổng số tiền bị phạt là ít nhất.

Bài tập 4-8. ★

Một lần Tôn Tẫn đua ngựa với vua Tề. Tôn Tẫn và vua Tề mỗi người có n con ngựa đánh số từ 1 tới n , con ngựa thứ i của Tôn Tẫn có tốc độ là a_i , con ngựa thứ i của vua Tề có tốc độ là b_i . Luật chơi như sau:

- Có tất cả n cặp đua, mỗi cặp đua có một ngựa của Tôn Tẫn và một ngựa của vua Tề.
- Con ngựa nào cũng phải tham gia đúng một cặp đua
- Trong một cặp đua, con ngựa nào tốc độ cao hơn sẽ thắng, nếu hai con ngựa có cùng tốc độ thì kết quả của cặp đua đó sẽ hoà.
- Trong một cặp đua, con ngựa của bên nào thắng thì bên đó sẽ được 1 điểm, hoà và thua không có điểm.

Hãy giúp Tôn Tẫn chọn ngựa ra đấu n cặp đua với vua Tề sao cho hiệu số: Điểm của Tôn Tẫn - Điểm của vua Tề là lớn nhất có thể.

Tài liệu tham khảo

- 1 Adelson-Velsky, Georgy Maximovich and Landis, Yevgeniy Mikhailovich. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences* (1962), 263-266.
- 2 Aho, Alfred V., Hopcroft, John E., and Ullman, Jeffrey D. *Data Structures and Algorithms*. Addison Wesley, 1983.
- 3 Barahona, Francisco and Tardos, Éva. Note on Weintraub's Minimum-Cost Circulation Algorithm. *SIAM Journal on Computing*, 18, 3 (1989), 579-583.
- 4 Bayer, Rudolf. Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica*, 1 (1972), 290–306.
- 5 Bellman, Richard. On a Routing Problem. *Quarterly of Applied Mathematics*, 16, 1 (1958), 87-90.
- 6 Bentley, J.L. *Solution to Klee's rectangle problems*. Carnegie-Mellon university, Pittsburgh, PA, 1977.
- 7 Coppersmith, Don and Winograd, Shmuel. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing* (New York, United States 1987), ACM, 1-6.
- 8 Cormen, Thomas H., Leiserson, Charles Eric, Rivest, Ronald Linn, and Stein, Clifford. *Introduction to Algorithms*. MIT Press, 2009.
- 9 de Berg, Mark, Cheong, Otfried, van Kreveld, Marc, and Overmars, Mark. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2008.
- 10 Dijkstra, Edsger Wybe. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1 (1959), 269-271.
- 11 Ding, Yuzheng and Weiss, Mark A. Best case lower bounds for heapsort. *Computing*, 49, 1 (1993), 1-9.
- 12 Dinic, E. A. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady*, 11, 5 (1970), 1277-1280.
- 13 Edmonds, Jack and Karp, Richard Manning. Theoretical improvements in the algorithmic efficiency for network flow problems. *Journal of the ACM*, 19 (1972), 248-264.

- 14 Fenwick, Peter M. A New Data Structure for Cumulative Frequency Tables. *Software: Practice and Experience*, 24 (1994), 327-336.
- 15 Fische, Johannes and Heun, Volker. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. *Lecture Notes in Computer Science*, 4614 (2007), 459-470.
- 16 Floyd, Robert W. Algorithm 245 - Treesort 3. *Communications of the ACM*, 7, 12 (1964), 701.
- 17 Ford, Lester Randolph and Fulkerson, Delbert Ray. *Flows in Networks*. Princeton University Press, 1962.
- 18 Ford, Lester Randolph and Johnson, Selmer M. A Tournament Problem. *The American Mathematical Monthly*, 66, 5 (1959), 387-389.
- 19 Fredman, Michael Lawrence and Tarjan, Robert Endre. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34, 3 (1987), 596-615.
- 20 Goldberg, Andrew Vladislav. *Efficient graph algorithms for sequential and parallel computers*. MIT, 1987.
- 21 Goldberg, Andrew Vladislav and Robert, Endre Tarjan. Finding minimum-cost circulations by canceling negative cycles. *Journal of the ACM (JACM)*, 36, 4 (1989), 873-886.
- 22 Hoare, Charles Antony Richard. QuickSort. *Computer Journal*, 5, 1 (1962), 10-15.
- 23 Hopcroft, John Edward and Karp, Richard Manning. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2, 4 (1973), 225-231.
- 24 Hopcroft, John Edward and Tarjan, Robert Endre. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16, 6 (1973), 372-378.
- 25 Huffman, David Albert. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers*, 40, 9 (1952), 1098-1101.
- 26 Karatsuba, Anatolii Alexeevich and Ofman, Yu. Multiplication of Many-Digital Numbers by Automatic Computers. *Doklady Akad. Nauk SSSR*, 145 (1962), 293-294. Translation in *Physics-Doklady* 7, 595-596, 1963.
- 27 Karp, Manning Richard. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23, 3 (1978), 309-311.

- 28 Klein, Morton. A Primal Method for Minimal Cost Flows with Applications to the Assignment and Transportation Problems. *Management Science*, 14, 3 (1967), 205-220.
- 29 Kruskal, Joseph Bernard. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7, 1 (1956), 48-50.
- 30 Kuhn, Harold. The Hungarian Method for the assignment problem.
- 31 Lacey, Stephen and Box, Richard. A fast, easy sort. *BYTE*, 16, 4 (1991), 315-ff.
- 32 Musser, David R. Introspective Sorting and Selection Algorithms. *Software: Practice and Experience*, 27, 8 (1997), 983 - 993.
- 33 Nguyễn, Đức Nghĩa and Nguyễn, Tô Thành. *Toán Rời Rạc*. NXB Giáo Dục, 1999.
- 34 Prim, Robert Clay. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36 (1957), 1389-1401.
- 35 Seidel, Raimund G. and Aragon, Cecilia R. Randomized search trees. *Algorithmica*, 16 (1996), 464-497.
- 36 Shell, Donald L. A high-speed sorting procedure. *Communications of the ACM*, 2, 7 (1959), 30-32.
- 37 Sleator, Daniel Dominic and Tarjan, Robert Endre. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32, 3 (1985), 652-686.
- 38 Sokkalingam, P. T., Ahuja, Ravindra K., and Orlin, James B. New Polynomial-Time Cycle-Canceling Algorithms for Minimum Cost Flows. *Network*, 36 (1996), 53-63.
- 39 Stoer, Mechthild and Wagner, Frank. A simple min-cut algorithm. *Journal of the ACM (JACM)*, 44, 4 (1997), 585-591.
- 40 Strassen, Volker. Gaussian Elimination is not Optimal. *Numerische Mathematik*, 13, 4 (1969), 354-356.
- 41 Tarjan, Robert Endre. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1, 2 (1972), 146-160.
- 42 Vuillemin, Jean. A data structure for manipulating priority queues. *Communications of the ACM*, 21, 4 (1978), 309-314.
- 43 Williams, J.W.J. Algorithm 232: Heapsort. *Communications of the ACM*, 7, 6 (1964), 347-348.