NGUYỄN HỮU ĐIỂN

Một số vấn đề về THUẬT TOÁN

NHÀ XUẤT BẢN GIÁO DỤC

NGUYỄN HỮU ĐIỂN

MỘT SỐ VẤN ĐỀ VỀ THUẬT TOÁN

NHÀ XUẤT BẢN GIÁO DỤC

LỜI NÓI ĐẦU

Hiện tại sách khoa học về tin học bằng tiếng Việt rất hiếm. Đa số là các sách hướng dẫn sử dụng phần mềm, kể cả việc dạy lập trình cũng chủ yếu dựa trên cơ sở một phần mềm biên dịch nào đó. Tin học đã được phát triển dựa vào ngành Toán học, tất cả những lí luận của ngành này rất chặt chẽ dựa vào những nguyên lí toán học. Tin học trong quá trình phát triển cũng nảy sinh đặc thù riêng và những lí luận riêng đáp ứng công nghệ thực tế của nó. Đặc biệt khái niệm về thuật toán thì bất cứ một người học tin học nào cũng cần phải nắm vững. Thuật toán đã được các nhà tin học phát triển và nghiên cứu khá kĩ, thuật toán có mặt trong mọi lĩnh vực của công nghệ. Nhưng cách thức người ta phân tích và thiết kế một thuật toán như thế nào thì không phải ai cũng hiểu thấu đáo. Cuốn sách này giới thiệu các vấn đề về thuật toán với độ phức tạp tính toán của nó. Để chứng minh và phân tích thuật toán một cách đúng đắn ta phải dùng một số công cụ toán học cơ bản như phương pháp chứng minh quy nạp toán học, phương pháp dùng hàm đánh giá O-lớn hoặc o-nhỏ, phương trình hồi quy, ... và các kiến thức cơ bản của toán học rời rạc.

Cuốn sách này được biên soạn nhằm phục vụ cho các bạn yêu thích toán học ứng dụng và tin học, các bạn ở lớp chuyên toán, chuyên tin, các thầy cô giáo và những người quan tâm đến giáo dục toán học, tin học ở trường phổ thông cũng như đại học. Mỗi chương đều đi từ đơn giản đến phức tạp, mỗi khái niệm mới đều được định nghĩa trước khi vào bài tập. Xương sống trong lí luận của cuốn sách này, ngoài những định nghĩa cơ bản của các khái niệm, là phương pháp lập luận theo quy nạp toán học mà bạn đọc có thể xem trong cuốn sách riêng về chuyên đề này [13].

Chương 1 của cuốn sách hoàn toàn mang tính công cụ toán học. Những bài tập trong chương này rất hay về mặt toán học và nó còn tạo đà cho các chương sau chuyên về tin học. Chương 2 nói về tính đúng đắn của thuật toán và cách chứng minh tính đúng đắn đó cho các thuật toán quen thuộc. Những chương còn lại chỉ ra độ phức tạp tính toán của các thuật toán kết hợp với phương pháp thiết kế chúng như phương pháp chia để trị, phương pháp quy hoạch động, phương pháp tham, phương pháp quy lui, ... Việc chỉ ra độ phức tạp tính toán của các thuật toán còn cho ta hướng phân tích và thiết kế thuật toán một cách tối ưu nhất. Các thuật toán được thể hiện bằng ngôn ngữ giả mã gần như ngôn ngữ lập trình Pascal mà nhiều người đã quen thuộc. Các bước của mỗi thuật toán được đánh số theo dòng, những giải thích và lập luận đều dựa trên các số dòng này. Mỗi mục đều có những bài giải mẫu và sau đó là những bài tập. Những bài tập có thể có lời giải hoặc gợi ý hoặc bạn đọc phải suy ngẫm dựa trên các ví dụ đã giải.

Do thời gian biên soạn cuốn sách bị eo hẹp nên có thể còn vài sai sót, mong bạn đọc góp ý. Hơn nữa, còn nhiều vấn đề về thuật toán liên quan đến độ phức tạp tính toán của chúng không được trình bày ở đây. Xin hẹn trong một tài liệu khác, chúng tôi sẽ giới thiệu về các chuyên đề như thuật toán sắp xếp, tìm kiếm, thuật toán số học, các thuật toán trong lí thuyết đồ thị, ... cùng các hàm thời gian tính toán của chúng và các chủ đề tính toán với thuật toán có độ phức tạp P, NP, NP đầy đủ,...Trình bày và biên soạn cuốn sách bằng chương trình LaTeX có cài dấu tiếng Việt do tác giả thực hiện [14].

Mọi góp ý gửi về địa chỉ: Ban biên tập sách Toán, Nhà xuất bản Giáo dục, 187B Giảng Võ, Hà Nội.

Tác giả cảm ơn ban biên tập Toán - Nhà xuất bản Giáo dục Hà Nội đã hết sức giúp đỡ để cuốn sách được in ra. Tác giả đặc biệt cảm ơn các đồng nghiệp ở Khoa Toán - Cơ - Tin học, ĐHKHTN, ĐHQGHN; Phòng Giải tích số và Tính toán Khoa học, Viện Toán học, đã giúp đỡ và động viên rất nhiều trong quá trình hoàn thành cuốn sách.

Hà Nội, tháng 09 năm 2005

Nguyễn Hữu Điển

CHƯƠNG 1
CÔNG CỤ ĐỂ PHÂN TÍCH VÀ THIẾT KẾ THUẬT TOÁN

1.1. THUẬT TOÁN VÀ CÁC TÍNH CHẤT CỦA NÓ

Sử dụng thuật toán không phải chỉ bắt đầu từ khi xuất hiện máy tính. Loài người đã sử dụng những thuật toán từ rất lâu để giải một cách hệ thống các bài toán. Một cách hình thức, một thuật toán có thể được mô tả như một dãy hữu hạn những quy tắc để giải một bài toán. Còn định nghĩa chính xác và đầy đủ của thuật toán phải dùng khái niệm máy Turing. Định nghĩa sau đây đủ cho mục đích của tài liệu này.

Định nghĩa 1.1. *Thuật toán* là một thủ tục đầy đủ và từng bước cho việc giải một bài toán cụ thể. Mỗi bước phải thể hiện rõ ràng theo thuật ngữ một số hữu hạn quy tắc và đảm bảo kết thúc trong một số hữu hạn lần áp dụng của những quy tắc này. Một quy tắc đặc thù là sự thực hiện một hoặc nhiều phép toán.

Để mô tả thuật toán người ta có rất nhiều cách thể hiện nhằm mục đích dễ hiểu cho người đọc và dễ dàng thực hiện chúng trong thực tế, đặc biệt là trong máy tính. Có ba cách cơ bản mô tả thuật toán:

1. Mô tả bằng lời, liệt kê theo từng bước và các phép thực hiện của thuật toán. Cách này thường dài và cồng kềnh khi nhiều người cùng quan tâm tới thuật toán, mặt khác vì không có chuẩn mực nên rất dễ dẫn đến hiểu sai. Nhưng cách này thích hợp cho việc mô tả và nghiên cứu một thuật toán.

2. Mô tả thuật toán bằng sơ đồ khối. Hiện tại có một số loại sơ đồ khối mô tả thuật toán rất thuận lợi, trực quan và rất gọn. Nhưng với những thuật toán lớn, phức tạp thì nó không đáp ứng được những yêu cầu của người phân tích hệ thống.

3. Mô tả thuật toán bằng giả mã. Đây là cách tối ưu cho người nghiên cứu thuật toán và thiết kế thuật toán. Bằng cách này ngôn ngữ mô tả tự nhiên như ở cách 1, mặt khác rất gần với các ngôn ngữ lập trình bậc cao để thể hiện thuật toán. Với cấu trúc của giả mã ta cũng dễ phân tích, chứng minh tính đúng đắn, tìm được độ phức tạp và thiết kế được thuật toán.

Trong tài liệu này ta dùng bộ giả mã được mô tả dưới đây.

1.1.1. Giả mã mô tả thuật toán

Bộ giả mã là một tập hợp những từ khóa và một số những cú pháp gần với ngôn ngữ tự nhiên để mô tả thuật toán. Với bộ giả mã trong tài liệu này, ta chọn những từ khóa *gần giống* với ngôn ngữ lập trình Pascal.

Những thuật toán được thực hiện như một chương trình con, nó được thể hiện bằng một trong hai hình thức: Những hàm **function** và thủ tục **procedure**. Trong trường hợp hàm số, giá trị trả về sau khi thực hiện hàm được thực hiện bằng lệnh **return**. Thông tin được chuyển cho hàm và thủ tục bằng đối-thông số tương ứng. Một thủ tục thường có dạng như sau :

Thuật toán 1.1 Ví dụ một thủ tục (<tên thủ tục>)

procedure <tên thủ tục>(<danh sách đối số>)
Đầu vào: <đối số k > mô tả, $k = 1, 2, ..., n$.
Đầu ra: <đối số m > mô tả $m = n+1, n+2, ...$
<khai báo biến mới>;
...
<Các câu lệnh của thủ tục>;
...
end <tên thủ tục>

Trong những <> là những tên hoặc thông số cụ thể của thủ tục hoặc các bước lệnh trong thủ tục. Sau từ khóa **Đầu vào:** là mô tả những thông số được đưa vào thủ tục. Sau **Đầu ra:** là mô tả những thông số được đưa ra khỏi thủ tục. Ngay sau đó có thể là những dòng lệnh khai báo biến cục bộ hoặc biến toàn cục. Ta quy định bắt đầu thủ tục từ dòng đầu tiên và kết thúc với từ khóa **end** kèm theo tên của thủ tục.

Các phép toán trong giả mã giống như ngôn ngữ bậc cao Pascal và được xếp theo thứ tự ưu tiên được liệt kê như sau:

1. Những dấu ngoặc tròn: (và).
2. Phép nhân, lũy thừa và phép chia: *, ^, /.
3. Phép cộng và phép trừ: +, −.
4. Toán tử quan hệ: <,>,<=,>=,=,<>.
5. Toán tử lôgic: **and, or, not**.

Các khối lôgic trong thuật toán cũng giống như các ngôn ngữ lập trình bậc cao. Các khối **if ... end if, for ... end for, while ... end while** kết thúc bằng từ khóa **end** kết hợp với từ mở khối. Để dễ hiểu ta lấy ví dụ mô tả tính tổng n số được ghi vào mảng $X[1..n]$ (thuật toán 1.2).

Những câu lệnh trong thủ tục hoặc hàm được đánh số để thuận tiện cho việc phân tích thuật toán. Nói chung những khối lệnh của giả mã hoạt động giống như ngôn ngữ lập trình Pascal. Ta chỉ ra những cú pháp như sau:

Thuật toán 1.2 Tính tổng n số (*Sum*)

function *Sum*($X[1..n]$)
Đầu vào: $X[1..n]$ mảng n số thực $x_1, x_2, ..., x_n$.
Đầu ra: $x_1 + x_2 + ... + x_n$.

```
1:  Tong := X[1];
2:  for i := 2 to n do
3:      Tong := Tong + X[i];
4:  end for
5:  return(Tong);
end Sum
```

1. Khối rẽ nhánh:

```
if <biểu thức lôgic> then
   <câu lệnh>;
end if
```

hoặc khối rẽ nhánh đầy đủ

```
if <biểu thức lôgic> then
   <câu lệnh 1>;
else
   <câu lệnh 2>;
end if
```

2. Khối lặp hữu hạn biết trước:

```
for <chỉ số>:=<giá trị đầu> to (downto ) <giá trị cuối>
do
   <các câu lệnh>;
end for
```

3. Vòng lặp thực hiện một lần:

```
repeat
   <các câu lệnh>;
until <biểu thức lôgic>
```

4. Vòng lặp có thể không thực hiện lần nào:

```
while <biểu thức lôgic> do
```

<các câu lệnh>;
end while

Các biến theo loại dữ liệu như trong Pascal: số nguyên, số thực, chuỗi, mảng và bảng ghi dùng cho những biến động như:

Node = **record**

Info: Infotype

Next: → Node

end.

1.1.2. Một số ví dụ thuật toán

1. Thuật toán Euclid. Một bài toán cổ nhất trong lí thuyết số là xác định ước chung lớn nhất $gcd(a,b)$ của hai số nguyên dương a và b. Ước chung lớn nhất của a và b là số nguyên dương lớn nhất mà cả hai a và b đều chia hết không có dư. Bài toán tính $gcd(a,b)$ đã được biết đến từ thời cổ Hy Lạp. Thuật toán sơ đẳng nhất là phân tích ra thừa số nguyên tố của a và b và sau đó tập hợp những thừa số nguyên tố chung với lũy thừa có số mũ nhỏ nhất của chúng nhân lại cho kết quả $gcd(a,b)$. Tuy nhiên với các số a và b lớn, việc phân tích ra thừa số nguyên tố tốn rất nhiều thời gian, thậm chí ngày nay khi đã có những máy tính cực nhanh. Một thuật toán có hiệu quả được biết đến trong cuốn sách *Cơ sở* của Euclid từ 300 năm trước công nguyên. Thuật toán Euclid có lẽ đã được biết đến trước đó 200 năm hoặc sớm hơn.

Thuật toán sớm nhất này dựa trên cơ sở kết luận: *Với $a \geq b$, một số nguyên đồng thời được chia hết bởi a và b khi và chỉ khi nó được chia hết bởi $a-b$ và b.* Như vậy,

$$gcd(a,b) = gcd(a-b,b), \quad a \geq b, \quad gcd(a,a) = a. \tag{1.1}$$

Từ công thức (1.1) cho ta thuật toán 1.3. Sau mỗi vòng lặp **while** trong *gcd*, số lớn hơn trong những giá trị trước của a và b được thay bằng những số nguyên dương nhỏ hơn thực sự. Do đó chắc chắn *gcd* sẽ dừng và cho kết quả là ước chung lớn nhất của a và b.

Thuật toán Euclid tinh chế lại thuật toán *gcd* bằng công cụ dữ kiện sau đây: Nếu $a > b$ và $a-b$ vẫn còn lớn hơn b, thì $a-b$ lại được thay

Thuật toán 1.3 Tính ước chung lớn nhất (*gcd*)

```
function   gcd(a,b)
  Đầu vào:   a,b hai số nguyên dương.
  Đầu ra:    gcd(a,b) ước chung lớn nhất của a,b.
  while a <> b do
     if a > b then
        a := a - b;
     else
        b := b - a;
     end if
  end while
  return(a);
end   gcd
```

bằng $a-2b$ và sau đó mới tính tiếp. Do đó, nếu a không là bội của b, thì a được thay thế cuối cùng bằng $r = a - qb$, ở đây r là số dư khi b chia cho a. Như vậy, tất cả những phép trừ liên tiếp có thể thay bằng một lệnh gọi a **mod** b, ở đây **mod** là một hàm dựng sẵn được định nghĩa bởi

$$a \quad \textbf{mod} \quad b = a - b\lfloor\frac{a}{b}\rfloor, \text{ } a \text{ và } b \text{ là những số nguyên, } b \neq 0,$$

ở đây $\lfloor x \rfloor$ kí hiệu số nguyên lớn nhất nhỏ hơn hoặc bằng x.

Thuật toán 1.4 Tính ước chung lớn nhất theo Euclid (*Euclidgcd*)

```
function   Euclidgcd(a,b)
  Đầu vào:   a,b hai số nguyên dương.
  Đầu ra:    gcd(a,b) ước chung lớn nhất của a,b.
  while b ≠ 0 do
     Remainder := a mod b;
     a := b;
     b := Remainder;
  end while
  return(a);
end   Euclidgcd
```

Ví dụ tính $gcd(108,8)$, có 13 phép trừ $108-8, 100-8, 92-8, \ldots, 12-8$ thực hiện trong *gcd* được thay bằng một lần tính duy nhất $108 \mod 8 = 4$.

Như vậy công thức để tính ước chung lớn nhất dựa vào công thức

$$gcd(a,b) = gcd(b, a \textbf{ mod } b). \tag{1.2}$$

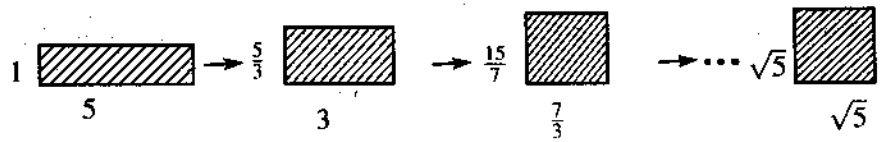
Chú ý khi a là bội của b thì $gcd(a,b) = b$ và a **mod** $b = 0$. Như vậy theo quy ước $gcd(b,0) = b$ thì (1.2) còn đúng khi a là bội của b.

Thuật toán Euclid 1.4 dựa trên cơ sở công thức (1.2) đã tương thích với khái niệm số không một cách không hình thức.

Ta có thể quan sát thuật toán *Euclidgcd* với đầu vào $a = 825, b = 420$.

a	b	r
825	420	405
420	405	15
405	15	0

2. Thuật toán lấy căn bậc hai của người Babilion. Từ thế kỉ thứ VI trước công nguyên, những người theo trường phái Pythagoras đã phát hiện ra rằng độ dài cạnh huyền tam giác vuông có hai cạnh góc vuông bằng 1 không thể biểu diễn thành tỉ số của hai số nguyên. Điều kết luận trên tương đương với $\sqrt{2}$ không là số hữu tỉ. Nghĩa là biểu diễn nó dưới dạng số thập phân không bao giờ đầy đủ. Rất lâu trước đó người ta đã quan tâm tới việc tính căn bậc hai của một số dương a với độ chính xác cho trước. Thuật toán tính căn bậc hai của một số nguyên dương đã biết từ thời Babilion, khoảng 1500 năm trước công nguyên và có lẽ không phải là thuật toán hiển nhiên đầu tiên. Thuật toán Babilion dựa trên cơ sở trung bình hai điểm trên cả hai cạnh. Có lẽ người Babilion phát hiện ra thuật toán nhờ vào cách giải bài toán cầu phương số diện tích một hình đã cho. Ví dụ, ta xét một hình có diện tích là 5. Bước xấp xỉ đầu tiên, ta coi diện tích hình chữ nhật có cạnh 1 và 5. Nếu họ thay một cạnh bằng trung bình cộng của hai cạnh trước đó thì họ sẽ nhận được hình chữ nhật có một cạnh là 3 và cạnh kia là $\frac{5}{3}$. Bước tiếp theo, một cạnh là trung bình cộng của 3 và $\frac{5}{3}$ bằng $\frac{7}{3}$, thì cạnh kia là $5 : \frac{7}{3}$. Lặp lại kĩ thuật trên và mỗi bước ta đều vẽ hình chữ nhật có cạnh tương ứng thì càng ngày, càng gần đến giá trị của $\sqrt{5}$ (hình 1.1). Bằng ngôn ngữ hiện đại mô tả thuật toán Babilion để tính căn bậc hai của một số



Hình 1.1

dương a bất kì như sau: Khởi đầu ta cho giá trị dự đoán $x = x_1$ là giá trị gần đúng của $\sqrt{a}$ ($x_1^2 \approx a$). Bất cứ giá trị nào dự đoán cũng được, nhưng số dự đoán gần đúng thì hội tụ đến nghiệm nhanh hơn. Ta có thể tính các giá trị xấp xỉ tiếp theo $x_2, x_3, \ldots$ dần đến $\sqrt{a}$ theo công thức

$$x_i = \frac{x_{i-1} + \frac{a}{x_{i-1}}}{2}, \quad i = 2, 3, \ldots$$

Ta viết thuật toán Babilion lấy căn bậc hai như một hàm số với đối số đầu vào là số a và một số thực dương ε đo độ chính xác của $\sqrt{a}$. Đơn giản nhất ta lấy giá trị xấp xỉ ban đầu là a, thuật toán 1.5.

Thuật toán 1.5 Khai căn bậc hai của người Babilion (*BabilionSQRT*)

function *BabilionSQRT*(a, ε)
Đầu vào: a, ε hai số nguyên dương.
Đầu ra: $\sqrt{a}$ với độ chính xác ε.

```
1:  x:=a;
2:  while |x − a/x| > ε do
3:      x := (x + a/x)/2;
4:  end while
5:  return(x);
end BabilionSQRT
```

Ta biết rằng tìm căn bậc hai của một số dương là trường hợp đặc biệt của bài toán tìm nghiệm của phương trình đa thức (vì tìm $\sqrt{a}$ cũng là tìm nghiệm của phương trình $x^2 - a = 0$). Vấn đề này rất lớn không đề cập đến trong cuốn sách này.

3. Tính giá trị các đa thức. Một bài toán có thể có nhiều cách tiếp cận khác nhau. Một mặt có thể tiếp cận trực tiếp bằng lí thuyết toán học, nhưng bằng cách đó nhiều khi không hiệu quả khi triển khai trên máy tính. Ta lấy ví dụ bài toán cơ bản của toán học là tìm giá trị của những đa thức. Một đa thức có dạng $P(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$,

tính giá trị của nó tại $x = v$. Một cách giải là trực tiếp tính các số hạng $a_i x^i$ độc lập với $i = 1, 2, ..., n$ và lấy tổng các số hạng lại. Tuy nhiên khi tính mỗi lũy thừa v^i, $i = 1, 2, ..., n$ một cách hiệu quả bằng cách dùng kết quả đã tính trước đó v^{i-1} nhân với v. Từ đó ta có thể viết thuật toán 1.6: *PolyEval* tính giá trị đa thức.

Thuật toán 1.6 Tính giá trị đa thức (*PolyEval*)

function *PolyEval*($a[0..n], v$)
Đầu vào: $a[0..n]$ một mảng số thực, v một số thực.
Đầu ra: Giá trị của đa thức $P(x)$ tại $x = v$.

```
Sum:=a[0];
Product:=1;
for i := 1 to n do
    Product := Product * v;
    Sum := Sum + a[i] * Product;
end for
return(Sum);
end PolyEval
```

Thuật toán *PolyEval* thực hiện $2n$ phép nhân và n phép cộng, đó là cách tốt nhất thuật toán có thể thực hiện. Tuy nhiên có một cách khác tính giá trị đa thức mà giảm một nửa số phép nhân. Thuật toán mới này dựa theo phương pháp nổi tiếng của W. G. Horner từ năm 1819. Thuật toán dựa trên cơ sở dùng dấu ngoặc tròn nhóm bậc của đa thức. Ví dụ, đa thức bậc bốn $P(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ có thể viết lại thành

$$(((a_4 * x + a_3) * x + a_2) * x + a_1) * x + a_0.$$

Với đa thức bậc bất kì ta viết lại thuật toán *PolyEval* thành thuật toán 1.7 *HornerEval* (trang 14).

Dễ thấy, thuật toán *HornerEval* thực hiện n phép nhân và n phép cộng khi tính giá trị đa thức bậc n. Có thể chứng minh rằng thuật toán trên thực hiện n phép nhân hoặc phép chia và n phép cộng hoặc phép trừ. Do đó thuật toán *HornerEval* tối ưu hơn cho việc tính giá trị đa thức.

Thuật toán 1.7 Tính giá trị đa thức (*HornerEval*)

```
function  HornerEval(a[0..n], v)
  Đầu vào:  a[0..n] một mảng số thực, v một số thực.
  Đầu ra:   Giá trị của đa thức P(x) tại x = v.
  1:  Sum:=a[n];
  2:  for i := n − 1 downto 0 do
  3:      Sum := Sum * v + a[i];
  4:  end for
  5:  return(Sum);
end  HornerEval
```

1.1.3. Thuật toán hồi quy

Phép hồi quy là một công cụ rất mạnh trong phân tích và thiết kế thuật toán, nó cũng rất tiện ích cho việc mô tả thuật toán. Trong rất nhiều trường hợp, lời giải của bài toán có thể một cách tự nhiên nằm trong nghiệm những bài toán con của cùng bài toán. Biểu thức này thường có dạng quan hệ truy hồi với điều kiện ban đầu biết trước. Thể hiện những quan hệ hồi quy thực chất là thuật toán hồi quy. Thuật toán hồi quy rất thuận tiện giải những bài toán, ngoài ra quan hệ truy hồi cũng tạo thuận tiện cho việc phân tích độ phức tạp của thuật toán.

Hàm hồi quy sẽ được xem xét ở phần sau với nội dung cụ thể, ở đây ta chỉ lưu ý những hàm hồi quy với đại lượng đầu vào, phép hồi quy gọi lại chính nó với đầu vào nhỏ hơn. Thuật toán cũng tương tự như vậy. Ví dụ ta tính hàm $factorial(n) = n! = 1.2.3...n$, ở đây quy định $0! = 1$. Vậy hàm $factorial(n)$ thỏa mãn quan hệ hồi quy

$$factorial(n) = n \times factorial(n-1), \tag{1.3}$$

điều kiện ban đầu $factorial(1) = 1$.

Từ công thức trên ta có thuật toán 1.8, tính $n!$.

Ta thấy rằng trong thuật toán này hàm phải gọi lại chính nó. Như vậy việc tính toán lặp lại, nhưng ở đây ta không triển khai một vòng lặp nào. Người lập trình đã để việc đó cho những người làm chương trình biên dịch và được thống nhất cho tất cả lập trình viên.

Một lợi thế nữa dùng thuật toán hồi quy là trong toán học ta đã biết

Thuật toán 1.8 Tính giai thừa (*Factorial*)

```
function  Factorial(n)
  Đầu vào:  n số nguyên dương.
  Đầu ra:   n!.
  if n <= 1 then
      return(1);
  else
      return(n * Factorial(n - 1));
  end if
end  Factorial
```

quy nạp toán học là một phương pháp chứng minh có kèm theo xây dựng, những nhà tin học đã sử dụng được lợi thế này và cách làm này rất gần với suy luận tự nhiên của con người.

Trong thực tế các ngôn ngữ lập trình bậc cao cung cấp khả năng xây dựng những định nghĩa hay những thuật toán như trên như Pascal, C, PL/C, Lisp, APL, ... Những người lập trình dùng những ngôn ngữ này tạo ra sức mạnh và tính linh hoạt của phương pháp hồi quy.

Thuật toán 1.9 Tính ước chung lớn nhất theo hồi quy (*Recgcd*)

```
function  Recgcd(a,b)
  Đầu vào:  a,b hai số nguyên dương.
  Đầu ra:   gcd(a,b) ước chung lớn nhất của a,b.
  if b = 0 then
      return(a);
  else
      return(Recgcd(b, a mod b));
  end if
end  Recgcd
```

Thiết lập một thuật toán theo cách hồi quy là rất tự nhiên theo tư duy. Ta trở lại bài toán tính ước chung lớn nhất của hai số nguyên dương a và b. Thuật toán *gcd* theo phương án hồi quy dựa vào phương trình hồi quy

$$gcd(a,b) = gcd(b, a \mod b), \text{ điều kiện ban đầu } gcd(a,0) = a. \quad (1.4)$$

Vì $b > (a \mod b)$, nên mỗi lần áp dụng (1.4) thì thông số vào thực sự

nhỏ hơn thông số vào trước đó. Ta có thuật toán hồi quy 1.9.

Chú ý thuật toán *factorial* và *Recgcd* chứa một lần gọi lại chính nó. Có thể có những thuật toán gọi lại chính nó nhiều lần với kích thước đầu vào khác nhau.

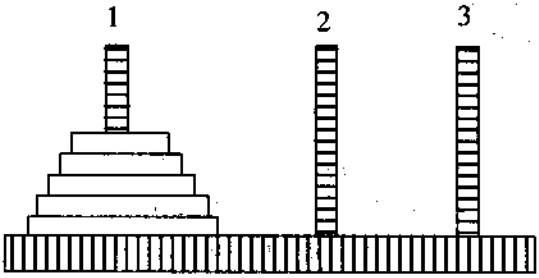
Tóm lại, một thủ tục hồi quy thường có cấu trúc như sau:

Thuật toán 1.10 Hình thức thuật toán hồi quy (*recursive*)

```
procedure   recursive (<đối số cỡ n>)
  Đầu vào:   Những đối số được nhập vào.
  Đầu ra:    Kết quả tính toán hồi quy.
      if <điều kiện A> then
         recursive (<đối số cỡ nhỏ hơn n>)
      else
         recursive (<đối số cỡ nhỏ hơn n khác>)
      end if
      <Tính toán một số thứ nữa>
end
```

1.1.4. Bài toán tháp Hà Nội

1. Bài toán. Có ba chiếc cọc, ở cọc thứ nhất người ta xuyên qua một chồng đĩa khác nhau từ lớn đến nhỏ. Hỏi có thể chuyển tất cả số đĩa sang cọc thứ ba còn nguyên thứ tự khi ta lấy cọc thứ hai làm trung gian để mỗi lần chuyển một đĩa và không được để đĩa lớn chồng lên đĩa nhỏ hơn? (hình 1.2).



Hình 1.2

Theo truyền thuyết bài toán này do một người Pháp học được của

các nhà sư ở châu Á và họ đặt tên cho bài toán này là bài toán tháp Hà Nội. Cũng có thể bài toán do các nhà sư của Việt Nam nghĩ ra làm trò chơi giải trí, và cũng có thể bài toán có từ Ấn Độ quê hương của đạo Phật.

2. Thuật toán hồi quy. Giả sử ở cọc thứ nhất có n đĩa khác nhau. Để chuyển n đĩa này sang cọc thứ ba đầu tiên ta phải làm được những việc sau đây:

- Chuyển $n-1$ đĩa từ cọc thứ nhất sang cọc thứ hai với việc dùng cọc thứ ba làm trung gian.
- Chuyển đĩa còn lại từ cọc thứ nhất sang cọc thứ ba.
- Cuối cùng là chuyển $n-1$ đĩa từ cọc thứ hai sang cọc ba dùng cọc thứ nhất làm cọc trung gian.

Từ nhận biết trên dễ dàng thiết lập được thuật toán hồi quy

Thuật toán 1.11 Tháp Hà Nội (*Towers*)

```
procedure   Towers(First, Auxiliary, Last, n)
  Đầu vào:  First, Auxiliary, Last tên của cột, n là số
            nguyên dương.
  Đầu ra:   Nghiệm của bài toán tháp Hà Nội.
    if n = 1 then
      <Chuyển đĩa thứ nhất từ First sang Last>;
    else
      Towers (First,Last,Auxiliary,n − 1);
      <Chuyển đĩa n từ First sang Last>;
      Towers (Auxiliary,First,Last,n − 1);
    end if
end   Towers
```

3. Số lần chuyển đĩa. Ta kí hiệu $T(n)$ là số lần chuyển n đĩa từ cọc thứ nhất sang cọc thứ ba thì theo cách phân tích hồi quy trên ta phải có

$$T(n) = \begin{cases} 1, & \text{nếu } n = 1 \\ 2T(n-1)+1, & \text{ngược lại} \end{cases}.$$

Do đó,

$$\begin{aligned}T(n) &= 2T(n-1)+1 = \\ &= 2(2T(n-2)+1)+1 = 4T(n-2)+2+1 = \\ &= 4(2T(n-3)+1)+2+1 = 8T(n-3)+4+2+1 = \cdots \\ &= 2^iT(n-i)+\sum_{j=0}^{i-1}2^j = 2^{n-1}T(1)+\sum_{j=0}^{n-2}2^j = 2^n-1.\end{aligned}$$

Như vậy có tất cả $T(n) = 2^n - 1$ lần chuyển đĩa.

4. Ngày tận thế. Theo truyền thuyết có 64 cái đĩa vàng và ba cọc kim cương gọi là tháp Brahma. Truyền thuyết nói rằng ngày tận thế của thế giới sẽ đến, khi tất cả các đĩa của bài toán Brahma được thực hiện xong.

Như vậy, với $n = 64$ thì có bao nhiêu lần chuyển đĩa? Ta tính theo công thức trên $T(64) = 2^{64} - 1 = 1,84 \times 10^{19} = 18.446.744.073.709.552.936$ lần tất cả. Nếu ta cho mỗi giây chuyển được một đĩa thì

$$\begin{aligned}1,84 \times 10^{19} \text{ giây } &= 3,07 \times 10^{17} \text{ phút} \\ &= 5,12 \times 10^{15} \text{ giờ} \\ &= 2,14 \times 10^{14} \text{ ngày} \\ &= 5,85 \times 10^{11} \text{ năm.}\end{aligned}$$

Ngày tận thế của thế giới là xấp xỉ 10^{10} năm.

1.2. CẤU TRÚC DỮ LIỆU CƠ SỞ

Người ta thường nói chương trình là thuật toán cộng với cấu trúc cơ sở dữ liệu. Trong phần trước những thuật toán trong ví dụ có dùng những dữ liệu cơ bản nhất như số nguyên, số thực, những kí tự và đại lượng lôgic. Trong phần này ta điểm lại những cấu trúc dữ liệu trừu tượng như danh sách, ngăn xếp, hàng đợi và cây nhị phân. Ta quan tâm tới cấu trúc trừu tượng trong ngữ cảnh áp dụng cho thuật toán.

Sự thực hiện của một thuật toán thường phụ thuộc vào việc chọn cách tổ chức dữ liệu. Loại dữ liệu trừu tượng liên quan tới việc mô tả tổ chức của dữ liệu và những phép toán được thực hiện trên dữ liệu này. Sự thực hiện đầy đủ một loại dữ liệu trừu tượng xác định một *cấu trúc dữ liệu.*

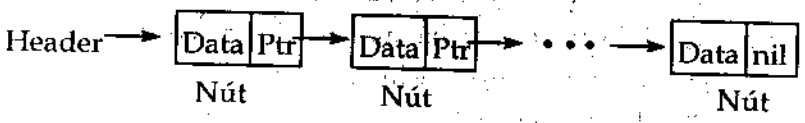
Loại dữ liệu trừu tượng tuyến tính

Rất nhiều thuật toán trong tài liệu này dùng danh sách như một trong những dữ liệu trừu tượng đầu tiên. Những phép toán ứng với danh sách được kèm theo như phép đưa vào một phần tử, xóa một phần tử và truy nhập trực tiếp một phần tử của danh sách. Để thuận tiện khi mô tả thuật toán ta thường coi dữ liệu danh sách như một mảng. Tuy nhiên trong thực tế nhiều tình thế ở đó danh sách liên kết sẽ cho ta thực hiện thuật toán tốt hơn.

1. Sự khác nhau giữa mảng và danh sách liên kết

Thiết lập mảng của một danh sách có lợi thế là truy nhập trực tiếp và rất nhanh tới bất kì phần tử nào của mảng. Tuy nhiên nó cũng có những nhược điểm là khi đưa một phần tử hoặc bỏ một phần tử vào mảng phải thực hiện một loạt các thao tác. Ví dụ như đưa một phần tử vào vị trí i của mảng có n phần tử thì ta phải thực hiện $n - i + 2$ phép gán chuyển dịch các phần tử từ vị trí i về phía cuối. Tương tự xóa một phần tử trong mảng cũng vậy. Ngoài ra mảng còn liên quan tới sự thiếu không gian bộ nhớ, vì khi khai báo mảng ta đã cố định bộ nhớ, điều này không thích hợp với dữ liệu rất lớn.

Danh sách liên kết khắc phục được những nhược điểm của mảng. Danh sách liên kết là tập những nút, mà mỗi nút chứa trường thông tin và một trường con trỏ. Trường thông tin chứa dữ liệu phụ thuộc vào các ứng dụng và trường con trỏ chứa con trỏ chỉ đến nút tiếp theo của danh sách liên kết. Mỗi danh sách liên kết còn có một con trỏ bắt đầu danh sách liên kết như hình 1.3.



Hình 1.3

Hầu như các ngôn ngữ lập trình bậc cao đều cung cấp con trỏ và biến động, nghĩa là ta thêm không gian bộ nhớ dùng con trỏ, những nút có thể sinh ra hoặc bỏ đi khi cần thiết. Những nút của danh sách liên kết có thể lưu vào các biến động, mà nó là những bản ghi như :

```
ListNote = record
        Info: InfoType;
        Next: ^ListNote;
    end
```

InfoType thể hiện một loại dữ liệu bất kì, có thể là số nguyên, số thực, chuỗi hoặc thậm chí là một bản ghi khác.

Việc thực hiện chèn một phần tử hoặc bỏ một phần tử từ danh sách liên kết rất hiệu quả vì chỉ thay đổi hai lần con trỏ là đủ. Danh sách liên kết trong tài liệu này rất ít được sử dụng nên ta không đi sâu vào các thao tác thêm và xóa phần tử từ danh sách liên kết. Bạn đọc có thể dễ dàng thực hiện các thao tác này bằng cách chuyển hướng các con trỏ vào các phần tử mới thêm hoặc bỏ đi.

2. Loại dữ liệu ngăn xếp trừu tượng

Ngăn xếp là một danh sách mà trong đó tất cả các thao tác đưa phần tử vào và lấy phần tử đi đều được thực hiện đồng thời tại điểm cuối và gọi nó là *đỉnh*. Như vậy phần tử được đưa vào sau cùng sẽ được lấy ra trước tiên, do đó người ta gọi ngăn xếp là LIFO (Last In First Out) danh sách vào sau ra trước. Thao tác đưa một phần tử vào ngăn xếp gọi là *đẩy vào* (push), thao tác lấy ra gọi là *lấy ra* (pop). Ngăn xếp được ứng dụng khắp nơi trong tin học, trong thiết kế thuật toán người ta cũng sử dụng cấu trúc dữ liệu này, như cơ sở hệ thống máy tính dùng ngăn xếp để thực hiện đối số, biến địa phương, trả về các địa chỉ, ...

Vì ngăn xếp là danh sách nên ta có thể dùng mảng hoặc danh sách liên kết. Trong trường hợp mảng ta dùng *stack*[1..*MaxSize*] và một biến tạm thời *Top* để chứa chỉ số của phần tử đang ở đỉnh của ngăn xếp. Nếu không có phần tử nào thì *Top* = 0. Khi ngăn xếp đầy thì *Top* = *MaxSize*. Chính biến *Top* cho phép ta kiểm tra ngăn xếp trống hay đã đầy rồi. Còn thao tác đẩy vào và lấy ra dễ thực hiện bằng lệnh giả mã như sau:

Đẩy phần tử *x* vào ngăn xếp:

```
if Top = MaxSize then
    <Thực hiện thao tác cần thiết khi ngăn xếp đầy>;
else
```

$Top := Top + 1;$
$Stack[Top] := x;$
end if

Lấy một phần tử trong ngăn xếp gán vào một biến x:

if $Top = 0$ **then**
<Thực hiện thao tác cần thiết khi ngăn xếp rỗng>;
else
$x := Stack[Top];$
$Top := Top - 1;$
end if

3. Loại dữ liệu hàng đợi trừu tượng

Hàng đợi là một danh sách mà trong đó tất cả các phần tử đưa vào tại một *đầu cuối* và tất cả các phần tử lấy đi tại một đầu khác, gọi là *cuối hàng*. Nghĩa là, trong hàng đợi, phần tử đưa vào trước sẽ lấy ra trước (FIFO). Cấu trúc dữ liệu này cũng được sử dụng rất nhiều trong thiết kế máy tính.

Hàng đợi là một danh sách nên có thể triển khai bằng mảng *Queue*[1..*MaxSize*] hoặc danh sách liên kết. Có nhiều cách triển khai các thao tác trên mảng đối với cấu trúc hàng đợi. Một cách tương đối hợp lí (không phải tốt nhất) là đưa vào chỉ số *Rear* - chỉ số phần tử cuối hàng và *Front* chỉ số phần tử ở đầu hàng. Nghĩa là thêm một phần tử vào thì *Rear* tăng lên 1, còn lấy ra một phần tử thì *Front* giảm đi 1. Điều kiện $Rear = Front$ chỉ ra hàng đợi rỗng, còn khi $Rear = MaxSize + 1$ chỉ ra hàng đợi. Bạn đọc có thể dễ dàng thể hiện giả mã cho các thao tác trên. Lí thuyết về loại dữ liệu này phong phú, do khuôn khổ cuốn sách có hạn tôi không đề cập ở đây.

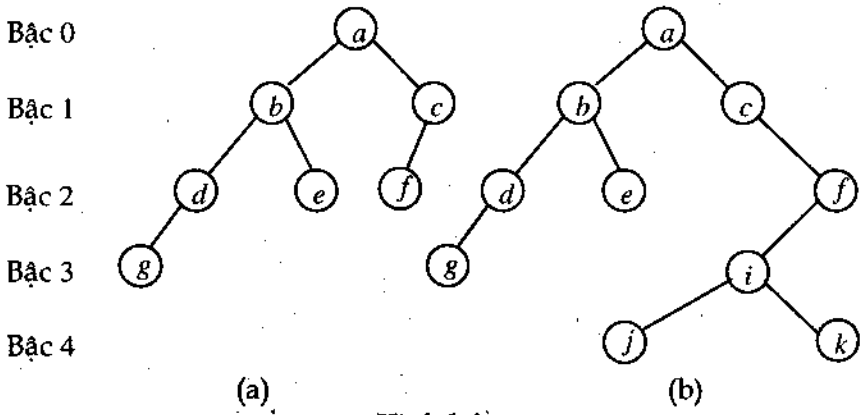
4. Loại dữ liệu cây nhị phân trừu tượng

Trong nghiên cứu công nghệ tin học người ta rất hay triển khai dữ liệu thành dạng hình cây. Ở đây, ta không nghiên cứu cấu trúc cây tổng quát mà chỉ xét lớp cây nhị phân hay được ứng dụng nhất.

Định nghĩa 1.2. *Cây nhị phân T* bao gồm một tập hợp các *nốt*, hoặc là nó bằng rỗng hoặc là nó có tính chất sau đây:

1. Một trong các nốt được gán là *nút gốc*, ta gọi nó là *R*.

2. Những nút còn lại được chia thành hai tập con rời nhau, gọi là *cây con trái* và *cây con phải*, mỗi cây này tương ứng lại là một cây nhị phân.



Hình 1.4

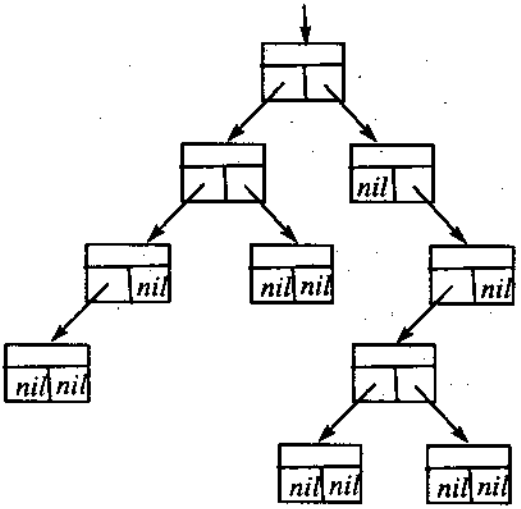
Nút gốc của cây con phải và cây con trái trong tính chất 2 của định nghĩa được gọi tương ứng là *con trái* và *con phải* tương ứng. Ví dụ cây nhị phân như hình 1.4, một nốt *V* gọi là *lá* nếu nó không có con. Người ta còn dùng các thuật ngữ khác như: *nốt cha, nốt cháu,...*. Người ta còn chia các nốt cây nhị phân ra thành các bậc, nút gốc *R* gọi là bậc 0 và tiếp sau đó là các bậc theo thứ tự lần lượt. *Độ sâu* (còn được gọi là độ cao) của cây *T* là khoảng cách bậc lớn nhất từ gốc cây *R* đến lá của nó. Hai cây ở hình 1.4a và hình 1.4b có độ sâu là 3 và 4 tương ứng.

Thể hiện cây nhị phân người ta dùng con trỏ và biến động như sau: Mỗi nốt của cây được thể hiện bằng biến động có cấu trúc

```
BinaryTreeNote = record
        Info: InfoType;
        LeftChild: ^BinaryTreeNote;
        RightChild: ^BinaryTreeNote;
    end BinaryTreeNote
```

Hơn nữa để thể hiện một cây ta thêm con trỏ, trỏ vào nốt gốc của

cây và cây trong hình 1.4b được thể hiện như hình 1.5.



Hình 1.5

1.3. PHƯƠNG PHÁP QUY NẠP TOÁN HỌC

Trong lí luận nghiên cứu thuật toán, ngoài cách tìm ra những tính chất của thuật toán suy trực tiếp từ định nghĩa, tiên đề người ta hay dùng phương pháp quy nạp toán học và phương pháp phản chứng. Trong tiết này ta nhắc lại những bước cơ bản của hai phương pháp trên. Những phương pháp này được dùng rất nhiều lần trong các chương sau.

Định nghĩa và ví dụ

Một tập hợp số tự nhiên $\mathbb{N} = \{0, 1, 2, ...\}$ cùng với hai phép toán cộng "+" và nhân "·" ta có những tính chất: Nếu $a, b, c \in \mathbb{N}$ thì $a \cdot b$ và $a + b$ cũng đều thuộc $\mathbb{N}$ và những đẳng thức sau luôn đúng: $0 + a = a$; $1 \cdot a = a$; $a + b = b + a$; $a \cdot b = b \cdot a$; $a(b + c) = ab + ac$; $a + (b + c) = (a + b) + c$ và $a(bc) = (ab)c$. Hơn nữa có một tiên đề

Tiên đề thứ tự: *Mọi tập con khác rỗng S của tập những số tự nhiên có phần tử nhỏ nhất.*

Từ tiên đề trên ta có thể giải rất nhiều bài toán: Ta nhắc lại một

số được gọi là *số vô tỉ* nếu nó không thể biểu diễn thành tỉ số hai số nguyên.

Ví dụ 1.1. *Chứng minh rằng $\sqrt{2}$ là một số vô tỉ.*

Chứng minh. Ta chứng minh bằng phản chứng, giả sử $\sqrt{2}$ là số hữu tỉ, nghĩa là $\sqrt{2} = \frac{a}{b}$ với a, b là những số nguyên và $b \neq 0$. Điều này kéo theo tập

$$S = \{n\sqrt{2} : \text{ đồng thời } n \text{ và } n\sqrt{2} \text{ là những số nguyên dương}\}$$

là khác trống vì nó có chứa a. Theo tiên đề thứ tự S có phần tử nhỏ nhất, ta cho là $j = k\sqrt{2}$. Vì $\sqrt{2} - 1 > 0$, $j(\sqrt{2} - 1) = j\sqrt{2} - k\sqrt{2} = \sqrt{2}(j - k)$ là số nguyên dương. Vì $2 < 2\sqrt{2}$ kéo theo $2 - \sqrt{2} < \sqrt{2}$ và như vậy $j\sqrt{2} = 2k$. Ta thấy rằng

$$(j - k)\sqrt{2} = k(2 - \sqrt{2}) < k\sqrt{2} = j.$$

Như vậy $(j - k)\sqrt{2}$ là một số nguyên dương trong S lại thực sự nhỏ hơn j. Điều này vô lí vì j là số nguyên dương nhỏ nhất trong S. Vậy điều giả sử là không đúng. ☺

Tiên đề thứ tự tương đương với nguyên lí quy nạp toán học. Trong suốt tài liệu này có sử dụng phương pháp quy nạp rất nhiều nên phần sau đây ta nghiên cứu khá kĩ cả về lí thuyết và thực hành. Phương pháp quy nạp gắn liền với việc chứng minh một mệnh đề đúng phụ thuộc vào biến số nguyên dương.

Nguyên lí quy nạp toán học dạng 1: *Cho n_0 là một số nguyên, $P(n)$ là mệnh đề có nghĩa với mọi số nguyên $n \geq n_0$. Nếu*

a) $P(n_0)$ đúng, và

b) Nếu $P(n)$ đúng, thì $P(n+1)$ cũng đúng với mọi số nguyên $n \geq n_0$,

khi đó $P(n)$ đúng với mọi số nguyên $n \geq n_0$.

Nguyên lí này gồm hai bước, thứ nhất ta kiểm tra khẳng định một tính chất với $n = n_0$, gọi là *Bước cơ sở*; sau đó ta chứng minh rằng nếu với mỗi $k \geq n_0$, $P(k)$ thoả mãn tính chất đã biết, thì suy ra $P(k+1)$ cũng có tính chất ấy, gọi là *Bước quy nạp*. Kết luận là $P(n)$ có tính chất đã cho với mọi $n \geq n_0$. Cách chứng minh theo quy nạp toán học là tránh cho ta phải kiểm tra vô hạn bước các khẳng định của mệnh đề. Phương pháp chứng minh như vậy gọi là *phương pháp quy nạp toán học*.

Ví dụ 1.2. *Chứng minh rằng với mọi $n \in \mathbb{N}$, $(n \neq 0)$ đẳng thức sau đúng*

$$1+2+\cdots+n = \frac{n(n+1)}{2}.$$

Chứng minh. *Bước cơ sở :* Với $n = 1$, ta có $1 = \dfrac{1(1+1)}{2}$ mệnh đề đúng.

Bước quy nạp : Ta kí hiệu $S(n) = 1+2+\cdots+n$, giả sử mệnh đề đúng với n, nghĩa là $S(n) = \dfrac{n(n+1)}{2}$. Ta phải chứng minh mệnh đề cũng đúng với $n+1$, tức là $S(n+1) = \dfrac{(n+1)(n+2)}{2}$. Thật vậy,

$$\begin{aligned} S(n+1) &= S(n) + (n+1) \\ &= \frac{n(n+1)}{2} + (n+1) \quad \text{(theo giả thiết quy nạp)} \\ &= \frac{n^2}{2} + \frac{n}{2} + n + 1 \\ &= \frac{n^2+3n+2}{2} = \frac{(n+1)(n+2)}{2}. \end{aligned}$$

☺

Trong toán rời rạc người ta kí hiệu 2^S là tập các tập con của tập hợp S. Kí hiệu $|S|$ là số phần tử của tập hợp S. Ta chứng minh khẳng định sau :

Ví dụ 1.3. *Nếu S là một tập hợp và $|S| = n$, thì $|2^S| = 2^n$.*

Chứng minh. Ta có thể kiểm tra một tập có hai phần tử $\{a,b\}$ $(n = 2)$ thì chúng có $4 = 2^2$ tập hợp con $\emptyset, \{a\}, \{b\}$ và $\{a,b\}$. Trong trường hợp $n = 3$, $\{a,b,c\}$ có các tập hợp con là những tập hợp con ở trường hợp $n = 2$ cộng thêm với các tập hợp đó thêm vào phần tử c. Ta có ý tưởng chứng minh theo quy nạp:

Bước cơ sở : Nếu $n = 0$, thì $S = \emptyset$ và $2^S = \{\emptyset\}$. Vậy $|2^S| = 1 = 2^0$.

Bước quy nạp : Nếu $n > 0$ ta chọn $c \in S$. Đặt tập hợp $T = S \setminus \{c\}$. Theo giả thiết quy nạp ta có $|2^T| = 2^{n-1}$. Những tập hợp con của S không chứa c có số lượng là số lượng của tập hợp 2^T, do đó có số lượng 2^{n-1}. Số lượng những tập hợp con chứa c là tất cả tập hợp trên thêm c vào, nên nó cũng có số lượng là 2^{n-1}. Tổng lại có $|2^S| = 2^{n-1} + 2^{n-1} = 2.2^{n-1} = 2^n$. Đó là điều cần chứng minh. ☺

Khi đánh giá các hàm tính toán người ta cũng thường dùng phương pháp quy nạp toán học. Ví dụ đánh giá tổng một lũy thừa như sau:

Ví dụ 1.4. *Chứng minh rằng với mọi số nguyên $r \geq 1$, tồn tại hằng số $c > 0$ sao cho $\sum_{k=1}^{n} k^r \leq cn^{r+1}$ với mọi n.*

Chứng minh. Ta chứng minh quy nạp theo n:

Bước cơ sở : Nếu $n = 1$ thì vế trái của bất đẳng thức cần chứng minh là $1^r = 1$. Vế phải của bất đẳng thức là $c.1^{r+1} = c$, do đó ta chọn c lớn hơn 1 bất kì thì bất đẳng thức đều đúng.

Bước quy nạp : Với $n > 1$, giả sử đã có kết luận đúng

$$\sum_{k=1}^{n-1} k^r \leq c(n-1)^{r+1}.$$

Cộng hai vế bất đẳng thức trên với n^r

$$\sum_{k=1}^{n} k^r \leq c(n-1)^{r+1} + n^r.$$

Để đạt được mục đích cần chứng minh ta chỉ cần chứng minh bất đẳng thức sau là đủ

$$c(n-1)^{r+1} + n^r \leq cn^{r+1}.$$

Ta chuyển vế và rút gọn chỉ còn chứng minh bất đẳng thức sau là đủ

$$n^r \leq c[n^{r+1} - (n-1)^{r+1}].$$

Đây lại là bài toán chứng minh bằng quy nạp toán học theo r, ta tách thành ví dụ sau đây. ☺

Ví dụ 1.5. *Chứng minh rằng với mọi số nguyên $r \geq 1$, tồn tại hằng số $c > 0$ sao cho với mọi $n \geq 1$, ta có*

$$n^r \leq c[n^{r+1} - (n-1)^{r+1}].$$

Chứng minh. Ta chứng minh bằng quy nạp theo r.

Bước cơ sở : Nếu $r = 1$, thì vế trái của bất đẳng thức là n, còn vế phải là $c[n^2 - (n-1)^2] = c[2n-1]$, do đó c lớn hơn 1 bất đẳng thức luôn đúng.

Bước quy nạp : Giả sử đã có bất đẳng thức $n^{r-1} \leq c[n^r - (n-1)^r]$. Ta nhân hai vế của bất đẳng thức với n và nhận được

$$n^r \leq c[n^{r+1} - n(n-1)^r].$$

Do đó cần chứng minh bất đẳng thức sau đây là đủ

$$\begin{aligned} c[n^{r+1} - n(n-1)^r] &\leq c[n^{r+1} - (n-1)^r] \text{ hoặc} \\ n^{r+1} - n(n-1)^r &\leq n^{r+1} - (n-1)^r \text{ hoặc} \\ -n(n-1)^r &\leq -(n-1)^r \text{ hoặc} \\ n(n-1)^r &\geq (n-1)^r \text{ hoặc} \\ n &\geq 1. \end{aligned}$$

Bất đẳng thức cuối cùng nhận được bằng cách chia hai vế cho $(n-1)^r$. Nếu giá trị này bằng 0 thì $n = 1$ và bất đẳng thức cần chứng minh là hiển nhiên với mọi $c \geq 1$. Từ $n \geq 1$ với mọi n và c ta có điều cần chứng minh. ☺

Ta cũng có thể dùng các bước quy nạp để tính tổng như ví dụ sau:

Ví dụ 1.6. *Tính tổng* $S(n) = \sum_{i=1}^{n} (5i+3)$.

Chứng minh. Ta giả thiết $S(n) = an^2 + bn + c$.

Bước cơ sở: Do $S(1) = a + b + c = 8$, do đó giả thiết đúng khi cho $a + b + c = 8$.

Bước quy nạp: Giả sử $S(n) = an^2 + bn + c$. Ta phải chứng minh rằng $S(n+1) = a(n+1)^2 + b(n+1) + c$. Thật vậy,

$$\begin{aligned} S(n+1) &= S(n) + 5(n+1) + 3 \\ &= (an^2 + bn + c) + 5(n+1) + 3 \quad \text{(theo giả thiết đã cho)} \\ &= an^2 + (b+5)n + c + 8. \end{aligned}$$

Ta muốn biểu thức sau cùng bằng kết quả như giả thiết :

$$\begin{aligned} an^2 + (b+5)n + c + 8 &= a(n+1)^2 + b(n+1) + c \\ &= an^2 + (2a+b)n + (a+b+c). \end{aligned}$$

So sánh hệ số của hai đa thức trên tại các bậc của n thì hệ số thứ hai có $b + 5 = 2a + b$, do đó $a = 2,5$. Từ bước cơ sở ta có $a + b + c = 8$ và hệ số

thứ ba $a+b+c=c+8$ suy ra $c=0$. Từ đó suy ra $b=5,5$. Cuối cùng ta phải có $S(n)=2,5n^2+5,5n=\dfrac{n(5n+11)}{2}$. ☺

Nguyên lí quy nạp toán học còn một dạng rất hay sử dụng là

Nguyên lí quy nạp toán học dạng 2: *Cho n_0 là một số nguyên, $P(n)$ là mệnh đề có nghĩa với mọi số nguyên $n \geq n_0$. Nếu*

a') $P(n_0)$ đúng;

b') Với mọi $k > n_0$, nếu $P(m)$ đúng với mọi m, $n_0 \leq m < k$, thì $P(k)$ cũng đúng,

và khi đó $P(n)$ đúng với mọi số nguyên $n \geq n_0$.

Ví dụ 1.7. *Chứng minh rằng mọi số tự nhiên lớn hơn 1 có thể biểu diễn dưới dạng tích của những số nguyên tố.*

Chứng minh. *Bước cơ sở.* Hiển nhiên mệnh đề đúng với mọi số nguyên tố, trường hợp đặc biệt $n=2$.

Bước quy nạp. Giả sử mệnh đề đúng với mọi số tự nhiên k, mà $2 \leq k < n$. Nghĩa là mọi số $2 \leq k < n$ đều biểu diễn dưới dạng tích các thừa số nguyên tố. Ta xét hai trường hợp :

1) Nếu n là số nguyên tố thì mệnh đề đúng.

2) Nếu n là hợp số thì theo định nghĩa hợp số tồn tại hai số nguyên $n_1 < n$ và $n_2 < n$ sao cho $n = n_1 n_2$. Theo giả thiết quy nạp n_1 và n_2 đều biểu diễn được thành tích các số nguyên tố. Do đó suy ra n cũng biểu diễn được thành tích các số nguyên tố. ☺

Định lý 1.1. *Nguyên lí quy nạp toán học dạng 1 đúng kéo theo nguyên lí quy nạp toán học dạng 2 cũng đúng và ngược lại.*

Chứng minh. 1. Cho $P(n)$ là mệnh đề và giả sử ta đã biết:

a') $P(n_0)$ đúng, và

b') Với mọi $n > n_0$, nếu $P(m)$ đúng với mọi m, $n_0 \leq m < n$, khi đó $P(n)$ cũng đúng.

Đặt $Q(n)$ là khẳng định " $P(m)$ là đúng với tất cả m, $n_0 \leq m < n$."

Để chỉ ra rằng $P(n)$ đúng với mọi n, cần chỉ ra rằng $Q(n)$ đúng với

mọi n. Ta chỉ ra $Q(n)$ đúng bằng quy nạp toán học dạng 1. Vậy ta cần chỉ ra:

a) $Q(n_0)$ đúng, và

b) Với mọi $n > n_0$, nếu $Q(n-1)$ đúng khi đó $Q(n)$ cũng đúng.

Khẳng định a) giống hệt a'), vậy nó đúng. Với khẳng định b), giả sử $Q(n-1)$ đúng. Khi đó $P(m)$ đúng với tất cả m, $n_0 \leq m \leq n-1$. Theo b'), $P(n)$ đúng. Như vậy $P(m)$ đúng với tất cả m, $n_0 \leq m < n$. Như vậy $Q(n)$ đúng. Theo nguyên lí quy nạp dạng 1, $Q(n)$ đúng với tất cả n. Điều ta phải chứng minh.

2. Ngược lại, hoàn toàn tương tự cách chứng minh trên. Cho $P(n)$ là mệnh đề. Nếu ta biết mệnh đề trên thoả mãn a) và b) đúng, ta cũng chỉ ra a') và b') của nguyên lí quy nạp dạng 2 đúng. Khi đó theo nguyên lí quy nạp 2 $P(n)$ đúng với mọi n. Như vậy nguyên lí quy nạp dạng 1 cũng đúng, đó là điều ta phải chứng minh. ☺

1.4. HÀM ĐO ĐỘ TÍNH TOÁN

Một công việc có thể có nhiều cách khác nhau để thực hiện công việc đó, nhất là thuật toán, có thể có nhiều thuật toán khác nhau thực hiện một công việc. Ta phải tìm một ngôn ngữ nào đó để so sánh được tốc độ thực hiện của những thuật toán đó khi làm cùng một việc, hoặc số lượng bộ nhớ chúng cùng thực hiện, hoặc đo được độ phức tạp của thuật toán khi ta dùng nó. Trong phần này ta nghiên cứu các hàm dùng làm công cụ đo các phép tính toán trong thuật toán.

1.4.1. Định nghĩa và tính chất

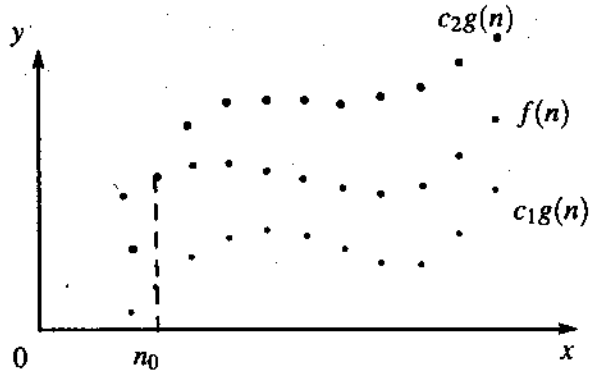
Ta chỉ nghiên cứu những hàm thực $f : \mathbb{N} \to \mathbb{R}$ xác định trên tập số nguyên dương và nhận giá trị hầu như dương, nghĩa là tồn tại một số nguyên n_0 sao cho $f(n) > 0$ với mọi $n > n_0$. Kí hiệu $\mathscr{F}$ là tập hợp tất cả những hàm như vậy.

Định nghĩa 1.3. Cho một hàm $g(n) \in \mathscr{F}$, kí hiệu $\Theta(g(n))$ là tập tất cả các hàm $f(n) \in \mathscr{F}$ có tính chất sau: Tồn tại những hằng số dương c_1, c_2 và n_0 sao cho với mọi $n \geq n_0$, ta có

$$c_1 g(n) \leq f(n) \leq c_2 g(n). \tag{1.5}$$

Nếu $f(n) \in \Theta(g(n))$, thì ta nói rằng $f(n)$ là *Thê ta lớn* của $g(n)$.

Theo định nghĩa trên thì hàm $f(n) \in \Theta(g(n))$, bắt đầu từ n_0 nằm kẹp giữa hai hàm có hai hằng số nhân với $g(n)$ như hình 1.6.



Hình 1.6

Chú ý nếu $f(n)$ là một công thức thì người ta lấy ngay công thức đó là hàm. Ví dụ như $f(n) = n^3 + 2n\log n - 3$ thì người ta có thể viết trực tiếp như là $n^3 + 2n\log n - 3 \in \Theta(n^3)$. Cũng có nghĩa như ví dụ $3\sqrt{n} + \log n + 108 \in \Theta(\sqrt{n})$ hoặc $864 \in \Theta(1)$.

Nhiều tài liệu dùng kí hiệu $f(n) = \Theta(g(n))$ thay cho $f(n) \in \Theta(g(n))$, trong tài liệu này đôi khi ta cũng dùng cả hai kí hiệu.

Mệnh đề sau đây suy ra từ định nghĩa và chỉ ra rằng Θ xác định một quan hệ tương đương trên tập hợp $\mathscr{F}$.

Mệnh đề 1.1. *Θ là một quan hệ tương đương trên tập hợp $\mathscr{F}$. Nghĩa là với mọi $f(n), g(n) \in \mathscr{F}$, những tính chất sau đúng :*

1. $f(n) \in \Theta(f(n))$ *(Θ phản xạ).*
2. $f(n) \in \Theta(g(n))$ *suy ra* $g(n) \in \Theta(f(n))$ *(Θ đối xứng).*
3. $f(n) \in \Theta(g(n))$ *và* $g(n) \in \Theta(h(n))$ *suy ra* $f(n) \in \Theta(h(n))$ *(Θ bắc cầu).*

Vì Θ xác định một quan hệ tương đương trên tập $\mathscr{F}$, nên ta có thể định nghĩa thứ tự trên tập đó.

Định nghĩa 1.4. Hai hàm $f(n)$ và $g(n)$ có *cùng thứ tự* nếu $f(n) \in \Theta(g(n))$.

Từ tính chất của Θ ta có thể phát biểu lại theo thứ tự như sau

1. $f(n)$ có cùng bậc với chính nó.

2. Nếu $f(n)$ có cùng bậc với $g(n)$ thì $g(n)$ cũng có cùng bậc với $f(n)$.

3. Nếu $f(n)$ cùng bậc với $g(n)$ và $g(n)$ cùng bậc với $h(n)$ thì $f(n)$ cùng bậc với $h(n)$.

Từ tính chất của Θ ta cũng suy ra $f(n)$ và $g(n)$ cùng bậc khi và chỉ khi $\Theta(f(n)) = \Theta(g(n))$. Như vậy $\Theta(f(n))$ có thể kí hiệu bằng $\Theta(g(n))$ với mọi hàm $g(n) \in \Theta(f(n))$. Khi mô tả thứ tự của hàm $f(n)$ người ta thường chọn tập hợp $\Theta(f(n))$ đơn giản nhất. Ví dụ

$$\Theta((8n^3+n)^{\frac{1}{2}} + \log n) = \Theta(2n^{\frac{3}{2}} + 6n\ln n)$$

$$= \Theta(n^{\frac{3}{2}}) = \Theta(n^{\frac{3}{2}} + 23n^{\frac{1}{3}}) = \Theta(\frac{n^3+n+1}{4n^2+\ln n}).$$

Vì $n^{\frac{3}{2}}$ có dạng đơn giản nhất trong lớp hàm tương đương này nên ta chọn lớp $\Theta(n^{\frac{3}{2}})$, khi đó ta nói rằng $f(n) \in \Theta(n^{\frac{3}{2}})$ có bậc $n^{\frac{3}{2}}$.

Theo định nghĩa $\Theta(g(n))$ thì những hàm thuộc tập hợp này bị kẹp giữa hai hàm tích của các hằng số với hàm $g(n)$. Bây giờ ta mở rộng lớp hàm này bằng cách chỉ xét chúng bị chặn một nửa.

Định nghĩa 1.5. Cho hàm số $g(n) \in \mathscr{F}$, ta định nghĩa $O(g(n))$ là một tập hợp tất cả các hàm $f(n) \in \mathscr{F}$ có tính chất: Tồn tại hằng số c và n_0 sao cho với mọi $n \geq n_0$

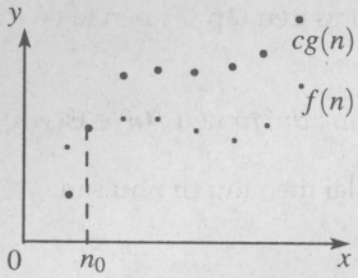
$$f(n) \leq cg(n). \tag{1.6}$$

Nếu $f(n) \in O(g(n))$, thì ta nói rằng $f(n)$ là *Ô lớn* của $g(n)$ (hình 1.7).

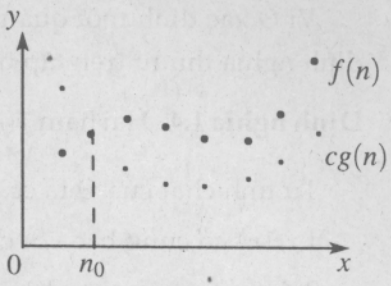
Định nghĩa 1.6. Cho hàm số $g(n) \in \mathscr{F}$, ta định nghĩa $\Omega(f(n))$ là một tập hợp tất cả các hàm $f(n) \in \mathscr{F}$ có tính chất: Tồn tại hằng số c và n_0 sao cho với mọi $n \geq n_0$

$$cg(n) \leq f(n). \tag{1.7}$$

Nếu $f(n) \in \Omega(g(n))$, thì ta nói rằng $f(n)$ là *Ô mê ga lớn* của $g(n)$ (hình 1.8).



Hình 1.7



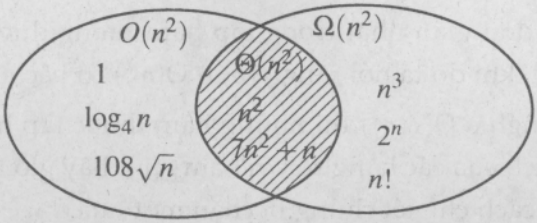
Hình 1.8

Những hàm thuộc O thường dùng như chặn dưới của hàm thực hiện tính toán trong thuật toán để giải một bài toán, còn những hàm thuộc Ω dùng như chặn trên của các hàm thực hiện tính toán trong thuật toán giải một bài toán cụ thể.

Dễ kiểm tra đẳng thức sau đúng

$$\Theta(g(n)) = \Omega(g(n)) \cap O(g(n)). \tag{1.8}$$

Hình 1.9 thể hiện sơ đồ Venn của ba tập $O(n^2), \Theta(n^2)$ và $\Omega(n^2)$.



Hình 1.9

Từ các định nghĩa trên ta có các tính chất sau:

Mệnh đề 1.2. *Những tính chất sau đây cho các hàm $f(n), g(n) \in \mathscr{F}$ là đúng:*

1. *Cho hằng số c, ta có $\Omega(f(n)) = \Omega(cf(n)), \Theta(f(n)) = \Theta(cf(n))$ và $O(f(n)) = O(cf(n))$.*

2. $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ *và* $f(n) \in \Omega(g(n))$.

3. $f(n) \in O(g(n)) \Rightarrow O(f(n)) \subseteq O(g(n))$.

4. $O(f(n)) = O(g(n)) \Leftrightarrow \Omega(f(n)) \in \Omega(g(n)) \Leftrightarrow \Theta(f(n)) \in \Theta(g(n))$.

5. $f(n) \in O(g(n)) \Rightarrow g(n) \in \Omega(f(n))$.

Người ta hay dùng kí hiệu

$$f(n) = g(n) + O(h(n)),$$

nghĩa là $f(n) - g(n) \in O(h(n))$. Tương tự như vậy cũng dùng cho Θ và Ω. Ví dụ $f(n) = 3n^3 + 4n^2 + 23n - 108$, thì ta có thể viết $f(n) = 3n^3 + O(n^2)$.

Chứng minh một hàm thuộc vào một lớp nào đó của O, Ω, Θ người ta cũng hay dùng phương pháp quy nạp như các ví dụ sau:

Ví dụ 1.8. *Chứng minh rằng* $\log n \in O(n)$.

Chứng minh. Ta phải chứng minh rằng với mọi $n \geq 1$, $\log n \leq n$. Thật vậy, ta chứng minh bằng phương pháp quy nạp theo n.

Với $n = 1$ bất đẳng thức trên đúng vì $0 < 1$.

Giả sử $n \geq 1$ và $\log n \leq n$. Khi đó

$$\begin{aligned} \log(n+1) &\leq \log 2n = \log n + 1 \\ &\leq n + 1 \quad \text{(theo giả thiết quy nạp).} \end{aligned}$$

☺

Ví dụ 1.9. *Chứng minh rằng* $2^{n+1} \in O\left(\frac{3^n}{n}\right)$.

Chứng minh. Ta sẽ chứng minh rằng với mọi $n \geq 7$ thì $2^{n+1} \leq \frac{3^n}{n}$.

Ta chứng minh bằng phương pháp quy nạp theo n. Với $n = 7$ thì $2^{n+1} = 2^8 = 256$ và $\frac{3^n}{n} = \frac{3^7}{7} > 256$.

Giả sử với $n \geq 7$ có bất đẳng thức đúng $2^{n+1} \leq \frac{3^n}{n}$. Ta phải chứng minh rằng $2^{n+2} \leq \frac{3^{n+1}}{n+1}$. Thật vậy,

$$\begin{aligned} 2^{n+2} = 2.2^{n+1} &\leq 2.\frac{3^n}{n} \quad \text{(theo giả thiết quy nạp)} \\ &\leq \frac{3n}{n+1}\frac{3^n}{n} \quad \text{(theo biến đổi dưới đây)} = \frac{3^{n+1}}{n+1}. \end{aligned}$$

(Phép biến đổi: $\frac{3n}{n+1} \geq 2 \Leftrightarrow 3n \geq 2n+2 \Leftrightarrow n \geq 2$). ☺

Hàm O lớn là một hàm rất quan trọng nên ta có mệnh đề :

Mệnh đề 1.3. *Nếu $f_1(n) \in O(g_1(n))$ và $f_2(n) \in O(g_2(n))$, thì*

$$f_1(n) + f_2(n) \in O(g_1(n) + g_2(n)).$$

Chứng minh. Giả sử với mọi $n \geq n_1$, $f_1(n) \leq c_1 g_1(n)$ và với mọi $n \geq n_2$, $f_2(n) \leq c_2 g_2(n)$.

Ta đặt $n_0 = \max\{n_1, n_2\}$ và $c_0 = \max\{c_1, c_2\}$. Khi đó với mọi $n \geq n_0$, ta có

$$f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n) \leq c_0(g_1(n) + g_2(n)).$$ ☺

Mệnh đề 1.4. *Nếu $f_1(n) \in O(g_1(n))$ và $f_2(n) \in O(g_2(n))$, thì*

$$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

Chứng minh. Giả sử với mọi $n \geq n_1$, $f_1(n) \leq c_1 g_1(n)$ và với mọi $n \geq n_2$, $f_2(n) \leq c_2 g_2(n)$.

Ta đặt $n_0 = \max\{n_1, n_2\}$ và $c_0 = c_1 + c_2$. Khi đó với mọi $n \geq n_0$, ta có

$$f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n) \leq (c_1 + c_2)(\max\{g_1(n), g_2(n)\}).$$ ☺

Mệnh đề 1.5. *Nếu $f_1(n) \in O(g_1(n))$ và $f_2(n) \in O(g_2(n))$, thì*

$$f_1(n).f_2(n) \in O(g_1(n).g_2(n)).$$

Chứng minh. Giả sử với mọi $n \geq n_1$, $f_1(n) \leq c_1 g_1(n)$ và với mọi $n \geq n_2$, $f_2(n) \leq c_2 g_2(n)$.

Ta đặt $n_0 = \max\{n_1, n_2\}$ và $c_0 = c_1.c_2$. Khi đó với mọi $n \geq n_0$, ta có

$$f_1(n).f_2(n) \leq c_1 g_1(n).c_2 g_2(n) = c_0(g_1(n).g_2(n)).$$ ☺

1.4.2. Thứ bậc trong tập hàm số

Từ định nghĩa của O ta có thể chỉ ra $n^2 \notin O(n)$ bằng phản chứng. Như vậy theo những mệnh đề ở phần trước suy ra n và n^2 nằm ở những lớp khác nhau.

Vì $n \in O(n^2)$ ta suy ra $O(n) \subset O(n^2)$. Một cách tự nhiên ta nói rằng n có thứ bậc nhỏ hơn n^2. Một cách tổng quát ta định nghĩa:

Định nghĩa 1.7. Cho hai hàm $f(n)$ và $g(n)$, ta nói rằng $f(n)$ có *thứ bậc nhỏ hơn* $g(n)$ nếu $O(f(n))$ thực sự chứa trong $O(g(n))$, nghĩa là $O(f(n)) \subset O(g(n))$.

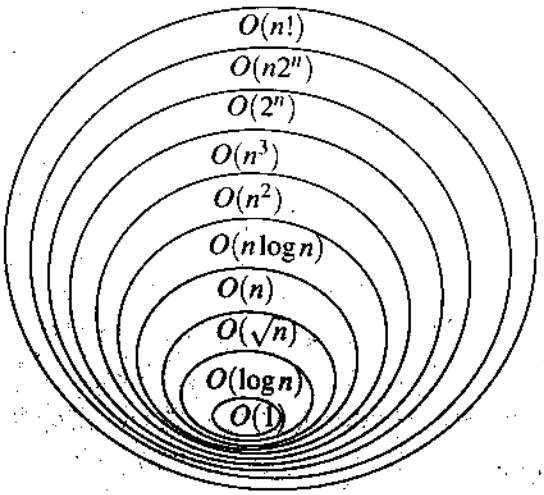
Trong phân tích thuật toán người ta hay dùng các hàm sau đây

$$1, \log n, \sqrt{n}, n, n\log n, n^2, n^3, 2^n, n2^n, n!.$$

Những hàm này tạo ra một dây xích thứ bậc tăng như sau

$$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n\log n)$$
$$\subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n2^n) \subset O(n!).$$

Ta có thể mô tả thứ bậc của các hàm trên bằng sơ đồ Venn như sau (hình 1.10) :



Hình 1.10

Như vậy hai hàm thuộc hai thứ bậc khác nhau không so sánh được. Ta có thể chỉ ra những thứ bậc trên thực sự khác nhau bởi một chú ý: Những thứ bậc của hai hàm $f(n)$ và $g(n)$ không so sánh được nếu $f(n) \notin O(g(n))$ và $g(n) \notin O(f(n))$.

Bây giờ ta xét một số kỹ thuật có thể dùng để thiết lập quan hệ thứ bậc giữa hai hàm $f(n)$ và $g(n)$. Đôi khi từ định nghĩa của các hàm Θ, O, Ω ta có thể chỉ ra rằng những đa thức bậc k đều thuộc $O(n^k)$, với điều kiện các hệ số của đa thức đều dương.

Mệnh đề 1.6. *Cho $P(n) = a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0$ là đa thức bậc k, $a_k > 0$. Khi đó $P(n) \in \Theta(n^k)$.*

Chứng minh. Do tính chất 2 trong mệnh đề 1.2, ta chỉ cần chứng minh $P(n) \in O(n^k)$ và $P(n) \in \Omega(n^k)$ là đủ. Ta chứng minh $P(n) \in O(n^k)$.

$$\begin{aligned}
P(n) &= a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0 \\
&\le a_k n^k + |a_{k-1}|n^{k-1} + \cdots + |a_1|n + |a_0| \\
&\le |a_k|n^k + |a_{k-1}|n^k + \cdots + |a_1|n^k + |a_0|n^k \\
&= (a_k + |a_{k-1}| + \cdots + |a_1| + |a_0|)n^k.
\end{aligned}$$

Vậy nếu $c = a_k + |a_{k-1}| + \cdots + |a_1| + |a_0|$, thì $P(n) \le cn^k$ với mọi $n \ge n_0 = 1$, vậy $P(n) \in O(n^k)$.

Ta chứng minh $P(n) \in \Omega(n^k)$.

$$\begin{aligned}
P(n) &= a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0 \\
&\ge a_k n^k - |a_{k-1}|n^{k-1} - \cdots - |a_1|n - |a_0| \\
&= \frac{a_k}{2}n^k + \left(\frac{a_k}{2}n^k - |a_{k-1}|n^{k-1} - \cdots - |a_1|n - |a_0|\right) \\
&\ge \frac{a_k}{2}n^k + \left(\frac{a_k}{2}n^k - (|a_{k-1}|n^{k-1} + \cdots + |a_1|n^{k-1} + |a_0|n^{k-1})\right) \\
&= \frac{a_k}{2}n^k + \left(\frac{a_k}{2}n - (|a_{k-1}| + \cdots + |a_1| + |a_0|)\right)n^{k-1}.
\end{aligned}$$

Nếu $c = \dfrac{a_k}{2}$ và $n = n_0 = (|a_{k-1}| + \cdots + |a_1| + |a_0|)\dfrac{2}{a_k}$, thì bất đẳng thức $P(n) \ge cn^k$ đúng với mọi $n \ge n_0$. Như vậy $P(n) \in \Omega(n^k)$. ☺

Khi chứng minh đa thức thuộc $\Theta(n^k)$ ta có thể dùng trực tiếp các định nghĩa. Nhưng nhiều trường hợp những hàm với số mũ ở biến không phải là nguyên thì rất khó, ví dụ như $2n^{\frac{3}{2}} + 6n\ln n \in \Theta(n^{\frac{3}{2}})$. Nói chung trong các trường hợp này ta phải tạo ra tỉ số giữa hai hàm $f, g \in \mathscr{F}$ như $\dfrac{f(n)}{g(n)}$ và tìm giới hạn theo n tiến tới vô cùng.

Mệnh đề 1.7. *Cho $f(n), g(n) \in \mathscr{F}$. Nếu giới hạn của tỉ số $\dfrac{f(n)}{g(n)}$ tồn tại khi n tiến tới ∞, thì hai hàm $f(n)$ và $g(n)$ có thể so sánh được. Hơn nữa, Giả sử $L = \lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)}$ tồn tại, khi đó những kết quả sau đây đúng.*

1. $0 < L < \infty \Rightarrow f(n) \subset \Theta(g(n))$, *khi đó f và g có cùng bậc.*

2. $L = 0 \Rightarrow O(f(n)) \subset O(g(n))$, *khi đó f có bậc nhỏ hơn g.*

3. $L = \infty \Rightarrow O(g(n)) \subset O(f(n))$, *khi đó f có bậc lớn hơn g.*

Chứng minh. Theo định nghĩa giới hạn, $L = \lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)}$ nghĩa là cho một số thực $\varepsilon > 0$ tồn tại số nguyên N_ε sao cho mọi $n \geq N_\varepsilon$ ta có $|\frac{f(n)}{g(n)} - L| < \varepsilon$, nghĩa là tương đương với

$$(L-\varepsilon)g(n) < f(n) < (L+\varepsilon)g(n), \tag{1.9}$$

với mọi $n \geq N_\varepsilon$.

Trường hợp 1 : $0 < L < \infty$. Với việc chọn ε là một số thực dương bất kì, ta có thể chọn $\varepsilon = \dfrac{L}{2}$. Đặt $c_1 = \dfrac{L}{2}, c_2 = \dfrac{3L}{2}$ và $n_0 = N_{\frac{L}{2}}$, thay tất cả vào (1.9) ta có

$$c_1 g(n) < f(n) < c_2 g(n), \quad \text{với mọi } n \geq n_0.$$

Do đó $f(n) \in \Theta(g(n))$.

Trường hợp 2 : $L = 0$. Thay $L = 0$ vào (1.9), với ε là số thực dương bất kì ta có

$$f(n) < \varepsilon g(n), \quad \text{với mọi } n \geq N_\varepsilon. \tag{1.10}$$

Suy ra $f(n) \in O(g(n))$. Để chứng minh $O(f(n)) \subset O(g(n))$, ta cần chỉ ra $f(n) \notin \Omega(g(n))$. Giả sử ngược lại tồn tại hằng số c và n_0 sao cho

$$cg(n) \leq f(n), \quad \text{với mọi } n \geq n_0.$$

Thay $\varepsilon = c$ vào (1.10) cho ta $f(n) < cg(n)$ với mọi $n \geq N_c$. Như vậy, với số nguyên bất kì n lớn hơn max của n_0 và N_c ta có hai điều trái ngược $f(n) < cg(n)$ và $cg(n) \leq f(n)$ điều này vô lí. Do đó $f(n) \notin \Omega(g(n))$, điều này kéo theo $O(f(n)) \subset O(g(n))$.

Trường hợp 3 : $L = \infty$. Tương tự như trên ta phải chứng minh $O(g(n)) \subset O(f(n))$. ☺

Điều kiện $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 0$ rất có ích trong tính toán, ta đưa vào khái niệm mới sau đây :

Định nghĩa 1.8. Cho hàm $g(n)$, ta định nghĩa tập hợp $o(g(n))$ là tập tất

cả những hàm $f(n)$ có tính chất

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Nếu $f(n) \in o(g(n))$, thì ta nói rằng $f(n)$ là *Ô nhỏ* của $g(n)$.

Định nghĩa 1.9. Cho hàm $g(n)$, ta định nghĩa $\sim (g(n))$ là tập hợp tất cả các hàm $f(n)$ có tính chất

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1.$$

Nếu $f(n) \in \sim (g(n))$, thì ta nói rằng $f(n)$ là *tiệm cận mạnh* tới $g(n)$ và kí hiệu là $f(n) \sim g(n)$.

Mệnh đề sau đây suy ra từ mệnh đề 1.7 và định nghĩa của o và $\sim$.

Mệnh đề 1.8. *Cho $f(n), g(n) \in \mathscr{F}$. Khi đó*

1. $f(n) \sim g(n) \Rightarrow f(n) \in \Theta(g(n))$.
2. $f(n) \in o(g(n)) \Rightarrow O(f(n)) \subset O(g(n))$.

Từ mệnh đề trên suy ra mệnh đề trước đây.

Mệnh đề 1.9. *Cho $P(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$ là đa thức bậc k, $a_k > 0$. Khi đó $P(n) \sim a_k n^k$.*

Chứng minh. Ta xét giới hạn tỉ số

$$\lim_{n\to\infty} \frac{a_k n^k + \cdots + a_0}{a_k n^k} = \lim_{n\to\infty} \left(1 + \frac{a_{k-1}}{a_k n} + \cdots + \frac{a_1}{a_k n^{k-1}} + \frac{a_0}{a_k n^k}\right) = 1.$$

Mệnh đề được chứng minh từ định nghĩa $\sim$. ☺

Sử dụng mệnh đề 1.7 và mệnh đề 1.8 đòi hỏi phải tính $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)}$, điều này rất khó làm trực tiếp. Thường thường người ta dùng quy tắc Lôpital.

Quy tắc Lôpital : Cho hai hàm $f(x)$ và $g(x)$ có đạo hàm với số thực x đủ lớn. Nếu $\lim\limits_{n\to\infty} f(x) = \infty$ và $\lim\limits_{n\to\infty} g(x) = \infty$, thì

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}.$$

Ví dụ để chứng minh rằng $2n^{\frac{3}{2}} + 6n\ln n \in \Theta(n^{\frac{3}{2}})$ ta tiến hành như sau:

$$\begin{aligned}\lim_{x\to\infty} \frac{2x^{\frac{3}{2}} + 6x\ln x}{x^{\frac{3}{2}}} &= 2 + 6\lim_{x\to\infty} \frac{\ln x}{x^{\frac{1}{2}}} \\ &= 2 + 6\lim_{x\to\infty} \frac{\frac{1}{x}}{\frac{1}{2}x^{-\frac{1}{2}}} \text{ (dùng quy tắc Lôpital)} \\ &= 2 + 12\lim_{x\to\infty} \frac{1}{x^{\frac{1}{2}}} = 2 + 0 = 2.\end{aligned}$$

Dựa vào quy tắc Lôpital ta có thể chỉ ra được một đa thức bất kì có bậc nhỏ hơn hàm số mũ.

Mệnh đề 1.10. *Với bất kì hằng số thực không âm k và $a > 1$, ta có $O(n^k) \subset O(a^n)$.*

Chứng minh. Ta xét tỉ số $\frac{x^k}{a^x}$. Liên tiếp áp dụng quy tắc Lôpital ta được

$$\lim_{x\to\infty} \frac{x^k}{a^x} = \lim_{x\to\infty} \frac{kx^{k-1}}{(\ln a)a^x} = \lim_{x\to\infty} \frac{k(k-1)x^{k-2}}{(\ln a)^2 a^x} = \ldots = \lim_{x\to\infty} \frac{k!}{(\ln a)^k a^x} = 0.$$

Do đó $n^k \in o(a^n)$. Mệnh đề được chứng minh suy ra từ mệnh đề 1.8. ☺

1.4.3. Đánh giá một số tổng các số hạng

Tổng $L(b,n) = \log_b 1 + \log_b 2 + \cdots + \log_b n = \log_b(n!)$ và tổng điều hòa $H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ hay được dùng trong phân tích thuật toán. Ta sẽ xác định bậc những tổng này.

Ta chú ý là với bất kì hằng số thực a và $b > 1$, $\log_a n \in \Theta(\log_b n)$, như vậy với mọi cơ số dấu, phụ thuộc trên đều đúng, nên ta có thể quy định bỏ cơ số trong hàm lôgarit như là cơ số bất kì. Khi đó

$$L(n) = \log(n!) = \log 1 + \log 2 + \cdots + \log n.$$

Mệnh đề 1.11. *Chứng minh rằng* $\log(n!) \in \Theta(n\log n)$.

Chứng minh. Vì

$$\log 1 + \log 2 + \cdots + \log n \le \log n + \log n + \cdots + \log n = n\log n.$$

Suy ra $L(n) \in O(n\log n)$.

Ta còn chỉ ra rằng $L(n) \in \Omega(n\log n)$. Đặt $m = \lfloor \frac{n}{2} \rfloor$. Ta có

$$\begin{aligned} L(n) &= (\log 1 + \log 2 + \cdots + \log m) + [\log(m+1) + \cdots + \log n] \\ &\geq \log(m+1) + \log(m+2) + \cdots + \log n \\ &\geq \log(m+1) + \log(m+1) + \cdots + \log(m+1) \\ &= (n-m)\log(m+1) \\ &\geq \frac{n}{2}\log\frac{n}{2} = \frac{n}{2}(\log n - \log 2). \end{aligned}$$

Dễ thấy $\frac{n}{2}(\log n - \log 2) \geq \frac{n}{2}[\log n - \frac{1}{2}\log n]$ với mọi $n \geq n_0$, ở đây n_0 đủ lớn (n_0 phụ thuộc vào cơ số của lôgarit). Như vậy $L(n) \geq \frac{1}{4}(n\log n)$ suy ra $L(n) \in \Omega(n\log n)$. ☺

1.5. PHƯƠNG TRÌNH HỒI QUY

Trong chương 3 của cuốn sách [13] có dành một mục trình bày về phương trình truy hồi bậc hai với cách tìm nghiệm thông qua phương trình đặc trưng. Phần này ta trình bày cách tìm nghiệm của phương trình truy hồi bậc nhất. Những phương pháp trong phần này đều dựa vào tìm hệ số hằng số, một đại lượng được coi là bất biến đối với các biến đổi của các thông số khác. Đại lượng bất biến tìm được nhờ những giá trị khác nhau của các thông số khác và dẫn đến nghiệm của phương trình truy hồi.

Ta nhắc lại một số khái niệm cơ bản: *Bậc* của phương trình truy hồi là hiệu giữa chỉ số lớn nhất và chỉ số nhỏ nhất của phương trình. Ví dụ những số Fibonacci được định nghĩa $f_0 = 0, f_1 = 1$ và

$$f_{n+2} = f_n + f_{n+1}, \quad n \geq 1.$$

Đây là phương trình bậc hai. Phương trình $u_{n+2} - u_{n+1} = 2$ là phương trình *tuyến tính* bậc nhất.

Nhưng các phương trình $x_n^2 + nx_{n-1} = 1$ và $x_n + 2^{x_{n-1}} = 3$ là *không tuyến tính.*

Phương trình truy hồi gọi là *thuần nhất* nếu tất cả các số hạng chỉ chứa các biến chỉ số. Ví dụ

$$x_{m+3} + 8x_{m+2} - 9x_m = 0$$

là phương trình thuần nhất. Phương trình

$$x_{m+3} + 8x_{m+2} - 9x_m = m^2 - 3$$

là phương trình không thuần nhất.

Giải phương trình truy hồi là tìm công thức tổng quát thứ n của phép truy hồi thỏa mãn phương trình đã cho.

Trong tiết này ta xét phương pháp giải phương trình truy hồi bậc nhất có dạng

$$x_n = ax_{n-1} + P(n), \quad n \neq 1,$$

ở đây P là một đa thức.

1. Giải phương trình thuần nhất $x_n = ax_{n-1}$ bằng cách nâng lên lũy thừa thành dạng $x^n = ax^{n-1}$. Phương trình này gọi là *phương trình đặc trưng*. Cuối cùng ta có $x = a$. Nghiệm của phương trình thuần nhất trên có dạng $x_n = Aa^n$, ở đây A là một hằng số sẽ xác định.

2. Thử lại nghiệm của phương trình có dạng $x_n = Aa^n + Q(n)$, ở đây Q là đa thức cùng bậc với P.

Ví dụ 1.10. *Cho $x_0 = 7$ và $x_n = 2x_{n-1}, n \geq 1$. Hãy tìm công thức tổng quát của x_n.*

Lời giải. *Cách thứ nhất:* Tìm nghiệm của phương trình đặc trưng $x^n = 2x^{n-1}$. Vậy $x = 2$. Như vậy nghiệm có dạng $x_n = A2^n$, ở đây A là một hằng số. Tìm A: Do $7 = x_0 = A2^0$ suy ra $A = 7$. Do đó nghiệm của phương trình là $x_n = 7.2^n$.

Cách thứ hai: Ta có

$$x_0 = 7, x_1 = 2x_0, x_2 = 2x_1, \cdots, x_n = 2x_{n-1}.$$

Nhân theo vế các đẳng thức trên, $x_0x_1...x_n = 7.2^n x_0x_1...x_{n-1}$. Ta nhận được cùng kết quả $x_n = 7.2^n$. ☺

Ví dụ 1.11. *Cho $x_0 = 7$ và $x_n = 2x_{n-1} + 1, n \geq 1$. Hãy tìm công thức tổng quát của x_n.*

Lời giải. *Cách thứ nhất:* Tìm nghiệm của phương trình đặc trưng $x^n = 2x^{n-1}$. Vậy $x = 2$. Như vậy nghiệm có dạng $x_n = A2^n$, ở đây A là

một hằng số. Do $P(n) = 1$ là đa thức bậc 0, nghĩa là đa thức chỉ có số hạng tự do. Do đó nghiệm tổng quát có dạng $x_n = A2^n + B$. Tìm A, B: Do $7 = x_0 = A2^0 + B = A + B$. Ta có $x_1 = 2x_0 + 1 = 15$ và như vậy $15 = x_1 = 2A + B$. Như vậy ta đi giải hệ phương trình:

$$\begin{cases} A + B &= 7, \\ 2A + B &= 15, \end{cases}$$

Ta tìm được $A = 8, B = -1$. Như vậy nghiệm của phương trình là $x_n = 8.2^n - 1 = 2^{n+3} - 1$.

Cách thứ hai: Ta có

$$\begin{array}{lcl} x_0 = 7, & & 2^n x_0 = 2^n 7, \\ x_1 = 2x_0 + 1, & & 2^{n-1}x_1 = 2^n x_0 + 2^{n-1}, \\ x_2 = 2x_1 + 1, & \Rightarrow & 2^{n-2}x_2 = 2^{n-1}x_1 + 2^{n-2}, \\ \dots\dots\dots & & \dots\dots\dots \\ x_{n-1} = 2x_{n-2} + 1, & & 2x_{n-1} = 2^2 x_{n-2} + 2, \\ x_n = 2x_{n-1} + 1, & & x_n = 2x_{n-1} + 1, \end{array}$$

ở đây ta nhân hàng thứ k với 2^{n-k}.

Cộng theo vế của các đẳng thức sau cùng cho kết quả

$$\begin{aligned} x_n &= 7.2^n + (1 + 2 + 2^2 + \cdots + 2^{n-1}) \\ &= 7.2^n + 2^n - 1 = 2^{n+3} - 1. \end{aligned}$$

Cách thứ ba: Đặt $u_n = x_n + 1 = 2x_{n-1} + 2 = 2(x_{n-1} + 1) = 2u_{n-1}$. Ta giải phương trình $u_n = 2u_{n-1}$ như ví dụ trước và nhận được $u_n = 2^n u_0 = 2^n(x_0 + 1) = 2^n.8 = 2^{n+3}$. Cuối cùng ta nhận được $x_n = u_n - 1 = 2^{n+3} - 1$. ☺

Ví dụ 1.12. *Cho $x_0 = 2$ và $x_n = 9x_{n-1} - 56n + 63$. Hãy tìm công thức tổng quát của x_n.*

Lời giải. Bằng cách nâng lên lũy thừa trong phương trình thuần nhất và tìm được phương trình đặc trưng $x^n = 9.x^{n-1}$, nghĩa là $x = 9$. Nghiệm của phương trình thuần nhất có dạng $x_n = A.9^n$. Do $P(n) = -56n + 63$ là đa thức bậc nhất và như vậy $Q(n)$ cũng là đa thức bậc nhất có dạng: $Q(n) = Bn + C$. Do đó nghiệm tổng quát có dạng $x_n = A.9^n + Bn + C$.

Bây giờ đi tìm các hệ số: Từ phương trình ban đầu ta có $x_0 = 2, x_1 = 9.2 - 56 + 63 = 25, x_2 = 9.25 - 56.2 + 63 = 176$. Thay các giá trị này vào nghiệm tổng quát cho ta hệ

$$\begin{cases} 2 &= A + C, \\ 25 &= 9A + B + C, \\ 176 &= 81A + 2B + C. \end{cases}$$

Ta tìm được $A = 2, B = 7, C = 0$. Do đó nghiệm tổng quát là $x_n = 2.9^n + 7n$.

☺

Ví dụ 1.13. *Cho $x_0 = 1$ và $x_n = 3x_{n-1} - 2n^2 + 6n - 3$. Hãy tìm công thức tổng quát của x_n.*

Lời giải. Bằng cách nâng lên lũy thừa phương trình thuần nhất ta tìm được phương trình đặc trưng $x^n = 3.x^{n-1}$, nghĩa là $x = 3$. Nghiệm của phương trình thuần nhất có dạng $x_n = A.3^n$. Do $P(n) = -2n^2 + 6n - 3$ là đa thức bậc hai và như vậy $Q(n)$ cũng là đa thức bậc hai dạng: $Q(n) = Bn^2 + Cn + D$. Do đó nghiệm tổng quát có dạng $x_n = A.3^n + Bn^2 + Cn + D$. Bây giờ đi tìm các hệ số: Từ phương trình ban đầu ta có $x_0 = 1, x_1 = 3.1 - 2 + 6 - 3 = 4, x_2 = 3.4 - 2.2^2 + 6.2 - 3 = 13$, $x_3 = 3.13 - 2.3^2 + 6.3 - 3 = 36$. Thay các giá trị này vào nghiệm tổng quát cho ta hệ

$$\begin{cases} 1 &= A + D, \\ 4 &= 3A + B + C + D, \\ 13 &= 9A + 4B + 2C + D, \\ 36 &= 27A + 9B + 3C + D. \end{cases}$$

Ta tìm được $A = B = 1, C = D = 0$. Do đó nghiệm tổng quát là $x_n = 3^n + n^2$.

☺

Ví dụ 1.14. *Cho $x_0 = 2$ và $x_n = 2x_{n-1} + 3^{n-1}$. Hãy tìm công thức tổng quát của x_n.*

Lời giải. Ta thử tìm nghiệm tổng quát dưới dạng $x_n = A.2^n + B.3^n$. Do $x_0 = 2, x_1 = 2.2 + 3^0 = 5$. Ta giải hệ

$$\begin{cases} 2 &= A + B, \\ 5 &= 2A + 3B. \end{cases}$$

Ta tìm được $A = 1, B = 1$. Nghiệm tổng quát là $x_n = 2^n + 3^n$. ☺

Trường hợp $a = 1$ trong phương trình truy hồi bậc nhất được thực hiện với việc xem xét đa thức $Q(n)$ có bậc cao hơn một bậc đối với đa thức $P(n)$.

Ví dụ 1.15. *Cho $x_0 = 7$ và $x_n = x_{n-1} + n, n \geq 1$. Hãy tìm công thức tổng quát của x_n.*

Lời giải. Bằng cách nâng lên lũy thừa phương trình thuần nhất ta tìm được phương trình đặc trưng $x^n = x^{n-1}$, nghĩa là $x = 1$. Nghiệm của phương trình thuần nhất có dạng $x_n = A.1^n = A$. Do $P(n) = n$ là đa thức bậc nhất và như vậy $Q(n)$ có bậc lớn hơn $P(n)$ một bậc và có dạng : $Q(n) = Bn^2 + Cn + D$. Do đó nghiệm tổng quát có dạng $x_n = A + Bn^2 + Cn + D$. Vì A và D đều là hằng số, ta có thể kết hợp chúng lại thành E, vậy $x_n = Bn^2 + Cn + E$. Bây giờ đi tìm các hệ số: Từ phương trình ban đầu ta có $x_0 = 7, x_1 = 7 + 1 = 8, x_2 = 8 + 2 = 10$. Thay các giá trị này vào nghiệm tổng quát cho ta hệ

$$\begin{cases} 7 &= E, \\ 8 &= B + C + E, \\ 10 &= 4B + 2C + E. \end{cases}$$

Ta tìm được $B = C = \frac{1}{2}, E = 7$. Do đó nghiệm tổng quát là $x_n = \frac{n^2}{2} + \frac{n}{2} + 7$.

Cách thứ hai: Ta có

$$\begin{aligned} x_0 &= 7, \\ x_1 &= x_0 + 1, \\ x_2 &= x_1 + 2, \\ &\ldots\ldots\ldots \\ x_n &= x_{n-1} + n. \end{aligned}$$

Cộng theo vế các đẳng thức trên

$$x_0 + x_1 + \ldots + x_n = 7 + x_0 + x_1 + \ldots + x_{n-1} + (1 + 2 + \ldots + n).$$

Ta nhận được cùng kết quả $x_n = 7 + \frac{n(n+1)}{2}$. ☺

Một số phương trình truy hồi không tuyến tính bậc nhất có thể đưa về phương trình truy hồi tuyến tính bậc nhất bằng cách đặt ẩn số phụ.

Ví dụ 1.16. *Cho* $u_0 = 3$ *và* $u_{n+1}^2 = u_n, n \geq 1$. *Hãy tìm công thức tổng quát của* u_n.

Lời giải. Đặt $v_n = \log u_n = \log u_{n-1}^{\frac{1}{2}} = \frac{1}{2}\log u_{n-1} = \frac{v_{n-1}}{2}$. Vì $v_n = \frac{v_{n-1}}{2}$, ta có nghiệm $v_n = \frac{v_0}{2^n}$, vậy $\log u_n = \frac{\log u_0}{2^n}$. Do đó $u_n = 3^{\frac{1}{2^n}}$. ☺

Một cách tổng quát hơn, xét phương trình hồi quy dạng

$$x_{n+1} = b_{n+1}x_n \quad (n \geq 0, x_0 \text{ đã cho }). \tag{1.11}$$

Biết rằng $b_1, b_2, \ldots$ đã cho, ta phải tìm $x_1, x_2, \ldots, x_n, \ldots$. Bài toán này dễ dàng giải được bởi $x_1 = b_1x_0$, $x_2 = b_2x_1 = b_2b_1x_0$ và $x_3 = b_3x_2 = b_3b_2b_1x_0, \ldots$ và cuối cùng ta dự đoán là

$$x_n = \{\prod_{i=1}^{n} b_i\}x_0 \quad (n = 0, 1, 2, \ldots). \tag{1.12}$$

Một dạng đặc biệt nữa ta quan tâm là

$$x_{n+1} = b_{n+1}x_n + c_{n+1} \quad (n \geq 0, x_0 \text{ đã cho }). \tag{1.13}$$

Hai dãy số $\{b_1, b_2, \ldots\}$ và $\{c_1, c_2, \ldots\}$ đã cho, ta muốn tìm công thức cho x_n?

Nếu ta cứ tiến hành như trên thì

$$x_1 = b_1x_0 + c_1, x_2 = b_2b_1x_0 + b_2c_1 + c_2, \ldots$$

và tiếp tục như vậy thì rất nhàm chán.

Ta có thể giải bài toán trên bằng cách đặt ẩn số phụ y_n

$$x_n = b_1b_2\ldots b_ny_n \quad (n \geq 1, x_0 = y_0). \tag{1.14}$$

Thay x_n vào (1.13) ta nhận được

$$b_1b_2\ldots b_{n+1}y_{n+1} = b_{n+1}b_1b_2\ldots b_ny_n + c_{n+1}.$$

Nhưng hệ số trước y_n và y_{n+1} như nhau nên ta có

$$y_{n+1} = y_n + d_{n+1} \quad (n \geq 0, y_0 \text{ đã cho }). \tag{1.15}$$

ở đây $d_{n+1} = \frac{c_{n+1}}{b_1b_2\ldots b_n}$ và hoàn toàn xác định.

Bây giờ ta giải (1.15) hoàn toàn đơn giản và suy ra công thức

$$y_n = y_0 + \sum_{j=1}^{n} d_j \quad (n \geq 0).$$

Từ công thức (1.14) ta có

$$x_n = (b_1 b_2 ... b_n)[x_0 + \sum_{j=1}^{n} d_j] \quad (n \geq 1). \tag{1.16}$$

Tóm lại để giải loại phương trình này ta cần tiến hành các bước sau:

1. Ta tiến hành đặt ẩn số phụ như công thức (1.14).
2. Giải phương trình cho kết quả tổng.
3. Trở về biến ban đầu.

Ví dụ 1.17. *Tìm công thức tổng quát cho phương trình bậc nhất*

$$x_{n+1} = 3x_n + n \quad (n \geq 0, x_0 = 0). \tag{1.17}$$

Lời giải. Tiến hành đặt ẩn số phụ cho (1.17), $x_n = 3^n y_n$. Thay vào (1.17) rồi đơn giản hóa cho kết quả

$$y_{n+1} = y_n + \frac{n}{3^{n+1}} \quad (n \geq 0, y_0 = 0).$$

Lấy tổng hai vế của đẳng thức trên cho kết quả

$$y_n = \sum_{j=1}^{n-1} \frac{j}{3^{j+1}} \quad (n \geq 0).$$

Theo cách đặt ẩn số phụ ta có kết quả

$$x_n = 3^n \sum_{j=1}^{n-1} \frac{j}{3^{j+1}} \quad (n \geq 0).$$

☺

Bây giờ ta trở lại với phương trình thuần nhất bậc hai dạng hay thường gặp

$$x_{n+1} = ax_n + bx_{n-1} \quad (n \geq 1, x_0, x_1 \text{ đã cho }). \tag{1.18}$$

Nghiệm của những phương trình này được tìm dưới dạng $x_n = \alpha^n$. Thay đại lượng này vào (1.18) và giản ước đi α^{n-1} cho ta phương trình bậc hai

$$\alpha^2 = a\alpha + b.$$

Thường thường phương trình bậc hai có hai nghiệm khác nhau khi giải phương trình trên, ta kí hiệu đó là α_1 và α_2. Khi đó nghiệm tổng quát của phương trình hồi quy có dạng

$$x_n = c_1 \alpha_1^n + c_2 \alpha_2^n \quad (n = 0, 1, 2, ...). \tag{1.19}$$

Hằng số c_1, c_2 tìm được thông qua giá trị ban đầu x_0, x_1.

Ví dụ 1.18. *Tìm công thức tổng quát cho những số Fibonacci*

$$F_{n+1} = F_n + F_{n-1} \quad (n \geq 1; F_0 = F_1 = 1). \tag{1.20}$$

Lời giải. Theo lí luận ở trên ta tìm nghiệm phương trình hồi quy ở dạng $F_n = \alpha^n$. Thay giá trị vừa đặt vào (1.20) và giản ước đại lượng chung còn $\alpha^2 = \alpha + 1$. Từ phương trình này giải được hai nghiệm $\alpha_1 = \dfrac{1+\sqrt{5}}{2}$ và $\alpha_2 = \dfrac{1-\sqrt{5}}{2}$. Từ đây ta có công thức nghiệm như (1.19) và chỉ còn lại tìm c_1 và c_2. Từ giá trị của $F_0 = F_1 = 1$ ta có hệ phương trình $F_0 = 1 = c_1 + c_2$; $F_1 = c_1\alpha_1 + c_2\alpha_2$. Giải ra ta tìm được $c_1 = \dfrac{\alpha_1}{\sqrt{5}}$ và $c_2 = -\dfrac{\alpha_2}{\sqrt{5}}$. Nghiệm của phương trình hồi quy đã cho là

$$F_n = \frac{1}{\sqrt{5}}\left\{\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right\} \quad (n = 0, 1, 2, \ldots)$$

☺

Dãy Fibonacci $1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$, nên cho dù công thức nghiệm rất cồng kềnh nhưng kết quả đều là số nguyên với giá trị của F_n.

Mặt khác do $\dfrac{1+\sqrt{5}}{2} > 1$ và $\left|\dfrac{1-\sqrt{5}}{2}\right| < 1$, suy ra khi n đủ lớn thì

$$F_n \sim \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}.$$

Bây giờ ta xét bất phương trình hồi quy có dạng

$$x_{n+1} \leq x_n + x_{n-1} + n^2 \quad (n \geq 1, x_0 = x_1 = 0). \tag{1.21}$$

Câu hỏi đặt ra là sự tăng của dãy x_n và bị chặn như thế nào?

Tương tự như phương trình hồi quy ta cũng tìm nghiệm cho bất phương trình trên có dạng $x_n \leq K\alpha^n$ với K là hằng số. Ta giả sử rằng $x_n \leq K\alpha^n$ đúng với $n = 0, 1, 2, \ldots, N$. Khi đó từ (1.21) với $n = N$ ta tìm được

$$x_{N+1} \leq K\alpha^N + K\alpha^{N-1} + N^2.$$

Lấy c là nghiệm thực dương của phương trình $c^2 = c + 1$ và giả sử rằng

$\alpha > c$. Khi đó $\alpha^2 > \alpha + 1$ và ta đặt $\alpha^2 - \alpha - 1 = t$, ở đây $t > 0$. Do đó

$$\begin{aligned} x_{N+1} &\le K\alpha^{N-1}(1+\alpha) + N^2 \\ &= K\alpha^{N-1}(\alpha^2 - t) + N^2 \\ &= K\alpha^{N+1} - (tK\alpha^{N-1} - N^2). \end{aligned}$$

Để cho chắc chắn $x_{N+1} < K\alpha^{N+1}$ thì từ công thức trên cần phải có $tK\alpha^{N-1} > N^2$. Do đó ta có thể chọn K sao cho

$$K > \max_{N \ge 2} \left\{ \frac{N^2}{t\alpha^{N-1}} \right\}.$$

Vế phải của bất đẳng thức trên hữu hạn nên bước quy nạp toán học luôn luôn đúng.

Kết luận là bất đẳng thức (1.21) luôn luôn kéo theo với mỗi $\varepsilon > 0$, $x_n = O((c+\varepsilon)^n)$ ở đây $c = \dfrac{1+\sqrt{5}}{2}$.

Bằng lí luận hoàn toàn tương tự ta có:

Định lý 1.2. *Cho dãy $\{x_n\}$ thỏa mãn bất phương trình hồi quy dạng*

$$x_{n+1} \le b_0 x_n + b_1 x_{n-1} + \cdots + b_p x_{n-p} + G(n) \quad (n \ge p)$$

ở đây $b_i \ge 0$ với mọi i, $\sum b_i > 1$. Hơn nữa c là nghiệm thực dương của phương trình $c^{p+1} = b_0 c^p + \cdots + b_p$. Cuối cùng giả sử $G(n) = o(c^n)$.

Khi đó với mỗi $\varepsilon > 0$ cố định ta có $x_n = O((c+\varepsilon)^n)$.

Chứng minh. Cố định $\varepsilon > 0$ và đặt $\alpha = c + \varepsilon$, ở đây c là nghiệm của phương trình đã chỉ ra trong định lí. Từ $\alpha > c$, nếu ta đặt

$$t = \alpha^{p+1} - b_0 c^p - \cdots - b_p,$$

thì $t > 0$. Cuối cùng ta đặt

$$K = \max \left\{ |x_0|, \frac{|x_1|}{\alpha}, \ldots, \frac{|x_p|}{\alpha^p}, \max_{n \ge p} \left\{ \frac{G(n)}{t\alpha^{n-p}} \right\} \right\}.$$

Khi đó K hữu hạn và dễ thấy $|x_j| \le K\alpha^j$ với $j \le p$. Ta chứng minh

$|x_n| \leq K\alpha^n$ đúng với mọi n là đủ. Thật vậy,

$$\begin{aligned}
|x_{n+1}| &\leq b_0 |x_n| + \cdots + b_p |x_{n-p}| + G(n) \\
&\leq b_0 K\alpha^n + \cdots + b_p K\alpha^{n-p} + G(n) \\
&= K\alpha^{n-p}(b_0\alpha^p + \cdots + b_p) + G(n) \\
&= K\alpha^{n-p}(\alpha^{p+1} - t) + G(n) \\
&= K\alpha^{n+1} - (tK\alpha^{n-p} - G(n)) \leq K\alpha^{n+1}.
\end{aligned}$$

Đó là điều cần chứng minh. ☺

1.6. ĐỊNH LÍ CƠ BẢN CHO PHÂN TÍCH THUẬT TOÁN

Phân tích thuật toán là đi tìm hàm thời gian thực hiện của thuật toán và thường rút ra phương trình truy hồi cho các hàm hồi quy này. Một định lí cơ bản cho việc tìm hàm hồi quy loại này là:

Định lý 1.3. *Nếu n là lũy thừa của c, thì nghiệm của phương trình hồi quy*

$$T(n) = \begin{cases} d & \text{nếu } n \leq 1 \\ aT(\frac{n}{c}) + bn & \text{nếu } n > 1. \end{cases}$$

có công thức sau

$$T(n) = \begin{cases} O(n) & \text{nếu } a < c, \\ O(n\log n) & \text{nếu } a = c, \\ O(n^{\log_c a - 1}) & \text{nếu } a > c. \end{cases}$$

Chứng minh. Nếu n là lũy thừa của c, thì

$$\begin{aligned}
T(n) = a \cdot T\left(\frac{n}{c}\right) + bn &= a\left(a \cdot T\left(\frac{n}{c^2}\right) + \frac{dn}{c}\right) + bn \\
&= a^2 \cdot T\left(\frac{n}{c^2}\right) + \frac{abn}{c} + bn \\
&= a^2\left(a \cdot T\left(\frac{n}{c^3}\right) + \frac{bn}{c^2}\right) + \frac{abn}{c} + bn \\
&= a^3 \cdot T\left(\frac{n}{c^3}\right) + \frac{a^2bn}{c^2} + \frac{abn}{c} + bn \\
&\vdots \\
&= a^i T\left(\frac{n}{c^i}\right) + bn\sum_{j=0}^{i-1}\left(\frac{a}{c}\right)^j.
\end{aligned}$$

Ta đặt $i = \log_c n$ cho ta công thức

$$T(n) = a^{\log_c n} T\left(\frac{n}{c^{\log_c n}}\right) + bn \sum_{j=0}^{\log_c n - 1} \left(\frac{a}{c}\right)^j.$$

Ta có những công thức sau từ tính chất lôgarit:

$$a^{\log_c n} = \left(c^{\log_c a}\right)^{\log_c n} = \left(c^{\log_c n}\right)^{\log_c a} = n^{\log_c a}.$$

Do đó,

$$T(n) = d \cdot n^{\log_c a} + bn \sum_{j=0}^{\log_c n - 1} \left(\frac{a}{c}\right)^j.$$

Ta xét ba trường hợp cho tổng vế phải của bất đẳng thức trên:

Trường hợp 1. $a < c$: Trường hợp này tổng ở vế phải được đánh giá chuỗi của những số hạng là cấp số nhân,

$$\sum_{j=0}^{\log_c n - 1} \left(\frac{a}{c}\right)^j < \sum_{j=0}^{\infty} \left(\frac{a}{c}\right)^j = \frac{c}{c-a}.$$

Do đó

$$T(n) < dn^{\log_c a} + \frac{bcn}{c-a} = O(n)$$

vì thừa số thứ nhất có $a < c$ thì không ảnh hưởng đến sự đánh giá trên.

Trường hợp 2. $a = c$: Khi đó

$$T(n) = d \cdot n^{\log_c a} + bn \sum_{j=0}^{\log_c n - 1} 1^j = O(n \log n).$$

Trường hợp 3. $a > c$: Khi đó tổng của một cấp số nhân ở vế phải

$$T(n) = d \cdot n^{\log_c a} + bn \sum_{j=0}^{\log_c n - 1} \left(\frac{a}{c}\right)^j = d \cdot n^{\log_c a} + bn \frac{\left(\frac{a}{c}\right)^{\log_c n} - 1}{\frac{a}{c} - 1}$$

$$= d \cdot n^{\log_c a} + bn \frac{n^{\log_c a - 1} - 1}{\frac{a}{c} - 1} = O(n^{\log_c a - 1}).$$

☺

Từ định lí trên ta suy ra những trường hợp sau đây

- Nếu $T(n) = 2T(\frac{n}{3}) + dn$, thì $T(n) = O(n)$.
- Nếu $T(n) = 2T(\frac{n}{2}) + dn$, thì $T(n) = O(n \log n)$. Đây là công thức của thuật toán *mergesort*.
- Nếu $T(n) = 4T(\frac{n}{2}) + dn$, thì $T(n) = O(n^2)$.

1.7. BÀI TẬP

▷ **1.19.** Chứng minh rằng $\sum_{i=1}^{n} \frac{1}{2^i} = 1 - \frac{1}{2^n}$ với mọi $n \geq 1$.

▷ **1.20.** Chứng minh rằng $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ với mọi $n \geq 0$.

▷ **1.21.** Cho số thực $x \in \mathbb{R}^+$. Kí hiệu $\lfloor x \rfloor$ là số nguyên lớn nhất không lớn hơn x. Còn $\lceil x \rceil$ là số nguyên nhỏ nhất mà không nhỏ hơn x. Chứng minh rằng

a) Với mọi $n \geq 0$, đẳng thức sau đúng

$$\lfloor x \rfloor = \begin{cases} \frac{n}{2}, & \text{nếu } n \text{ là số chẵn} \\ \frac{n-1}{2}, & \text{nếu } n \text{ là số lẻ.} \end{cases}$$

b) Với mọi $n \geq 0$, đẳng thức sau đúng

$$\lceil x \rceil = \begin{cases} \frac{n}{2}, & \text{nếu } n \text{ là số chẵn} \\ \frac{n+1}{2}, & \text{nếu } n \text{ là số lẻ.} \end{cases}$$

▷ **1.22.** Chứng minh rằng $n^3 + 2n$ chia hết cho 3 với mọi $n \geq 0$.

▷ **1.23.** Số Fibonacci F_n với $n \geq 0$ được định nghĩa truy hồi như sau: $F_0 = 0, F_1 = 1$ và với $n \geq 2$, $F_n = F_{n-1} + F_{n-2}$. Chứng minh rằng

$$\sum_{i=0}^{n} F_i = F_{n+2} - 1.$$

▷ **1.24.** Những đẳng thức sau đây có đúng không?

a) $\sqrt{n} = O(\log n)$.

b) $\sqrt{n} \log n = O(n)$.

c) $\frac{1}{n} = O(\log n)$.

d) $3n^2 + \sqrt{n} = O(n^2)$.

▷ **1.25.** Chứng minh rằng $(n+1)^2 = O(n^2)$.

▷ **1.26.** Chứng minh rằng $\lceil \log n \rceil = O(n)$.

▷ **1.27.** Chứng minh rằng $3n\lfloor \log n \rfloor = O(n^2)$.

▷ **1.28.** Giải phương trình hồi quy với $T(1) = 1$ và với mọi $n \geq 2$, $T(n) = 3T(n-1) + 2$.

▷ **1.29.** Giải phương trình hồi quy với $T(1) = 1$ và với mọi n là lũy thừa của 2 ($n \geq 2$), $T(n) = 2T\left(\frac{n}{2}\right) + 6n - 1$.

▷ **1.30.** Giải phương trình hồi quy với $T(1) = 1$ và với mọi n là lũy thừa của 2 ($n \geq 2$) , $T(n) = 2T\left(\frac{n}{2}\right) + n^2 - n$.

▷ **1.31.** Giải phương trình hồi quy với $T(1) = 1, T(2) = 6$ và với mọi $n \geq 3$, $T(n) = T(n-2) + 3n + 4$.

▷ **1.32.** Giải phương trình hồi quy với $T(1) = 1$ và với mọi $n \geq 2$,

$$T(n) = \sum_{i=1}^{n-1} T(i) + 7.$$

CHƯƠNG 2
TÍNH ĐÚNG ĐẮN CỦA THUẬT TOÁN

2.1. GIỚI THIỆU KIỂM CHỨNG THUẬT TOÁN

Giả sử chúng ta đã thiết kế được một thuật toán và đưa nó vào chương trình để thể hiện nó. Ta có thể tin được rằng thuật toán có đưa ra lời giải đúng hay không? Không tính sai sót về mặt cú pháp khi thuật toán đã thể hiện được những kết quả khác nhau khi ta cho những mẫu đầu vào để thử. Tuy vậy với các mẫu đã cho, chương trình cho kết quả đúng đi chăng nữa thì thuật toán cũng còn tiềm tàng những lời giải sai. Như vậy ta phải chứng minh được kết quả của thuật toán cho đầu ra luôn luôn đúng. Những phương pháp luận đã được xây dựng để chứng minh tính đúng đắn của thuật toán khi có đầu vào đúng và kết quả đầu ra phải là đúng. Thường có hai phương pháp lôgic để kiểm tra thuật toán hoặc chương trình đúng hay không đó là kiểm thử (testing) và chứng minh tính đúng đắn (Correctness proof).

Kiểm thử thuật toán là thử thuật toán với *những mẫu đầu vào*. Chứng minh tính đúng đắn là *chứng minh theo toán học*. Việc kiểm thử thuật toán tiềm ẩn những lỗi không ngờ tới. Chỉ dùng kiểm thử nhiều khi rất nguy hiểm cho việc triển khai nhiều thuật toán với nhau.

Khi đó những lỗi trong các thuật toán khác nhau sẽ kết hợp làm cho ta khá bối rối không biết chính xác chương trình hoạt động thế nào? Chứng minh tính đúng đắn của thuật toán cũng nhiều khi còn có lỗi, vì vậy ta phải kết hợp cả hai.

Một thuật toán thế nào là đúng đắn? Ta có thể định nghĩa như sau:

Một thuật toán được gọi là *đúng đắn* nếu với mọi đầu vào chấp nhận được, nó cho đầu ra đúng. Việc chứng minh tính đúng đắn của thuật toán bao gồm hai phần:

1. Ta phải chỉ ra rằng thuật toán kết thúc thì nhận được kết quả đúng. Phần này người ta thường gọi là xác minh *tính đúng đắn bộ phận* của thuật toán.

2. Chứng tỏ thuật toán luôn luôn kết thúc.

Để định rõ thế nào là một thuật toán cho thông tin đúng, người ta thường xét hai vấn đề sau: Thứ nhất là những *khẳng định đầu vào*, nó bao gồm các tính chất thông tin đầu vào cần phải có. Thứ hai là *khẳng định đầu ra* bao gồm những tính chất thông tin đầu ra cần phải có theo mục đích của thuật toán. Như vậy khi chứng minh tính đúng đắn của thuật toán ta phải chuẩn bị khẳng định đầu vào và khẳng định đầu ra thích hợp.

Định nghĩa 2.1. Một thuật toán S được gọi là *đúng đắn* đối với khẳng định đầu vào p và khẳng định đầu ra q, nếu p là đúng với các giá trị vào của S và nếu S kết thúc thì q là đúng với các giá trị ra của S.

Ví dụ sau đây minh họa khái niệm khẳng định đầu vào và khẳng định đầu ra.

Ví dụ 2.1. *Hãy chỉ ra rằng trong thuật toán 2.1 (ở trang 55) là đúng đắn đối với khẳng định đầu vào* $p : x = 1$ *và khẳng định đầu ra* $q : z = 3$.

Lời giải. Giả sử p đúng, nghĩa là $x = 1$ bắt đầu chương trình. Sau đó y được gán giá trị 2, tiếp đó z được gán tổng của x và y bằng 3, vậy nếu thủ tục kết thúc thì q đúng. Theo định nghĩa thủ tục trên đúng đắn với khẳng định đầu vào p và khẳng định đầu ra q. ☺

Thuật toán 2.1 Tính đúng đắn (vd)

procedure $vd(x)$
Đầu vào: Giá trị x
Đầu ra: Giá trị $x+2$
1: $y := 2;$
2: $z := x + y;$
end

Phần sau đây ta sẽ thể hiện chứng minh các thuật toán đúng đắn bằng phương pháp quy nạp và khẳng định đầu (khẳng định đầu vào) và khẳng định cuối (khẳng định đầu ra) được thể hiện ngay trong khi chứng minh chứ không phát biểu riêng như ví dụ trên. Bình thường người ta suy luận theo các khối trong thuật toán như khối có điều kiện, khối vòng lặp, khối liệt kê thứ tự, ... Trong tài liệu này ta cũng thực hiện các khía cạnh khác theo loại thuật toán hồi quy hay không hồi quy. Vì vậy ta phân biệt một chút về các thuật toán hồi quy và không hồi quy nó như thế nào.

2.2. THUẬT TOÁN HỒI QUY

Giả sử ta muốn tính $n!$ với n là một số nguyên. Ta có hai cách tính thứ nhất là từ định nghĩa giai thừa và cách tính này không hồi quy:

1. $n!$ *là tích của tất cả các số kể từ* 1 *đến* n.

Cách thứ hai là cách tính có áp dụng hồi quy:

2. *Nếu* $n = 1$ *thì* $n! = 1$, *ngược lại thì* $n! = n.(n-1)!$.

Ta chú ý tới cách thứ hai này thì nghe chừng có gì đó vô lí vì ta định nghĩa một cái gì đó mà chính trong định nghĩa lại dùng nó. Chỉ có chú ý là giá trị của $n!$ lại phải dùng đến giá trị nhỏ hơn $(n-1)!$. Như vậy ta chỉ có dùng quy nạp toán học mới chỉ ra rằng định nghĩa theo cách này là cho tất cả n nguyên dương.

Một cách hình thức việc tính không theo hồi quy có thể viết như thuật toán 2.2.

Thuật toán 2.2 Giai thừa không hồi quy (fact)

```
function  fact(n)
  Đầu vào:  Giá trị n > 0
  Đầu ra:   Tính n!
  fact := 1;
  for i := 1 to n do
      fact := i · fact;
  end for
end  fact
```

Còn tính theo hồi quy của $n!$ là thuật toán 2.3:

Thuật toán 2.3 Giai thừa hồi quy (fact)

```
function  fact(n)
  Đầu vào:  Giá trị n > 0
  Đầu ra:   Tính n!
  if n = 1 then
      fact := 1
  else
      fact := n · fact(n − 1);
  end if
end  fact
```

Trong chương này ta chứng minh tính đúng đắn của một số thuật toán hồi quy. Còn chương sau trở đi là chứng minh độ phức tạp của những thuật toán này.

2.3. TÍNH ĐÚNG CỦA THUẬT TOÁN HỒI QUY

Chứng minh tính đúng đắn của thuật toán hồi quy ta dùng trực tiếp phương pháp quy nạp. Cụ thể để chứng minh tính đúng đắn của thuật toán hồi quy ta phải tiến hành những bước sau đây:

1. Chứng minh bằng quy nạp theo cỡ của bài toán đã cho, ví như cỡ của mảng, số bit trong số nguyên, cỡ hàng và cột của ma trận, ...
2. Khởi đầu cơ sở của phép hồi quy chính là *bước cơ sở* của phương pháp quy nạp.

3. Cần phải chứng minh rằng phép gọi hồi quy tới bài toán con không là hồi quy vô hạn (thường thường là hiển nhiên khi tạo lập thuật toán).

4. *Bước quy nạp:* Giả sử phép gọi hồi quy là đúng, và dùng giả thiết quy nạp để chứng minh rằng phép gọi hiện tại là đúng.

Những ví dụ sau đây thể hiện các bước chứng minh trên:

Ví dụ 2.2. Thuật toán Fibonacci hồi quy: *Cho dãy Fibonacci:* $F_0 = 0$; $F_1 = 1$, *và với mọi* $n \geq 2$, $F_n = F_{n-2} + F_{n-1}$. *Chứng minh thuật toán sau đây đúng :*

Thuật toán 2.4 Số Fibonacci (fib)

```
function  fib(n)
  Đầu vào:  Giá trị n > 0
  Đầu ra:   Kết quả F_n
  1:  if n <= 1 then
  2:      return(n)
  3:  else
  4:      return(fib(n-1) + fib(n-2))
  5:  end if
end  fib
```

Chứng minh. Kết hợp thông tin đầu vào và thông tin đầu ra trong việc chứng minh tính đúng đắn thuật toán trên ta phải chứng minh mệnh đề sau:

Chứng minh rằng với mọi $n \geq 0$, $fib(n)$ *đều trả về* F_n.

Thật vậy, ta chứng minh bằng phương pháp quy nạp:

Bước cơ sở: Với $n = 0$, $fib(0)$ trả về 0 như mệnh đề. Với $n = 1$, $fib(1)$ trả về 1 như mệnh đề.

Bước quy nạp: Giả sử $n \geq 2$ và với mọi $0 \leq m < n$, $fib(m)$ đều trả về F_m. Ta phải chứng minh rằng $fib(n)$ trả về F_n. Thật vậy, trong trường hợp

này $fib(n)$ trả về kết quả

$$fib(n-1) + fib(n-2) = F_{n-1} + F_{n-2} \text{ (theo giả thiết quy nạp)}$$
$$= F_n. \qquad ☺$$

Ví dụ 2.3. Thuật toán Maximum hồi quy: *Tìm số lớn nhất được lưu trong mảng một chiều $A[1..n]$. Chứng minh thuật toán sau đây đúng :*

Thuật toán 2.5 Tính giá trị lớn nhất của chuỗi (maximum)

```
function  maximum(A[1..n])
  Đầu vào:  Giá trị A[1..n].
  Đầu ra:   Giá trị max của A[1..n].
  if n <= 1 then
     return(A[1])
  else
     return(max(maximum(n − 1), A[n]))
  end if
end  maximum
```

Chứng minh. Ta chứng minh mệnh đề sau:

Với mọi $n \geq 1$, $maximum(n)$ trả về giá trị $\max\{A[1], A[2], ..., A[n]\}$.

Ta chứng minh bằng quy nạp toán học theo $n \geq 1$ mệnh đề trên đúng.

Bước cơ sở: Với $n = 1$ thì $maximum(1)$ trả về $A[1]$ mệnh đề đúng.

Bước quy nạp: Giả sử $n \geq 1$ và $maximum(n)$ trả về kết quả $\max\{A[1], A[2], ..., A[n]\}$. Ta phải chứng minh rằng $maximum(n+1)$ trả về kết quả $\max\{A[1], A[2], ..., A[n], A[n+1]\}$.

Thật vậy, theo thuật toán thì $maximum(n+1)$ trả về kết quả:

$$\max(maximum(n), A[n+1]) = \max(\max\{A[1], A[2], ..., A[n]\}, A[n+1])$$
$$\text{(theo giả thiết quy nạp)}$$
$$= \max\{A[1], A[2], ..., A[n], A[n+1]\} \qquad ☺$$

Ví dụ 2.4. Thuật toán phép nhân hồi quy: *Với $x \in \mathbb{R}$, $\lfloor x \rfloor$ là số nguyên lớn nhất không vượt quá x. Chứng minh thuật toán phép nhân sau đây đúng :*

Thuật toán 2.6 Nhân hai số nguyên (multiply)

```
function multiply(y, z)
  Đầu vào: Cho giá trị y, z.
  Đầu ra: Giá trị tích yz.
  if z = 0 then
     return(0)
  else
     if z là số lẻ then
        return( multiply(2y, ⌊z/2⌋) + y )
     else
        return( multiply(2y, ⌊z/2⌋))
     end if
  end if
end multiply
```

Chứng minh. Ta phải chứng minh mệnh đề sau:

Với mọi $y, z \geq 0$, *multiply*(y,z) *trả về* yz.

Ta chứng minh bằng phương pháp quy nạp toán học theo $z \geq 0$.

Bước cơ sở: Với $z = 0$ thì *multiply*(y,z) trả về 0 như mệnh đề khẳng định.

Bước quy nạp: Giả sử với $z \geq 0$ và với mọi $0 \leq q \leq z$, *multiply*(y,z) trả về yq. Ta phải chứng minh rằng *multiply*$(y, z+1)$ trả về $y(z+1)$. Thật vậy, *multiply*$(y, z+1)$ trả về cái gì? Trả lời câu hỏi này có hai trường hợp xảy ra phụ thuộc vào $z+1$ là số chẵn hoặc lẻ.

Nếu $z+1$ là một số lẻ, thì *multiply*$(y, z+1)$ trả về:

$$\begin{aligned} multiply(2y, \lfloor \frac{z}{2} \rfloor) + y &= 2y\lfloor \frac{z}{2} \rfloor + y \quad \text{(theo giả thiết quy nạp)} \\ &= 2y(\frac{z}{2}) + y \quad \text{(vì } z \text{ là số chẵn)} \\ &= y(z+1). \end{aligned}$$

Nếu $z+1$ là một số chẵn, thì *multiply*$(y, z+1)$ trả về:

$$\begin{aligned} multiply(2y, \lfloor \frac{z+1}{2} \rfloor) &= 2y\lfloor \frac{z+1}{2} \rfloor \quad \text{(theo giả thiết quy nạp)} \\ &= 2y(\frac{z+1}{2}) \quad \text{(vì } z \text{ là số lẻ)} = y(z+1). \end{aligned}$$

☺

2.4. TÍNH ĐÚNG CỦA THUẬT TOÁN KHÔNG HỒI QUY

Chứng minh tính đúng đắn của thuật toán không hồi quy đều liên quan đến vòng lặp. Điển hình là vòng lặp: **while** <điều kiện> thực hiện *S*. Ta lưu ý rằng *S* được lặp đi lặp lại cho tới khi nào <điều kiện> nhận giá trị sai.

Định nghĩa 2.2. Ta gọi một điều khẳng định nào đó là *bất biến vòng lặp* nếu nó vẫn còn đúng sau mỗi lần thi hành.

Nói cách khác, nếu thông tin vào *p* kết hợp với <điều kiện> vào thuật toán *S* cho thông tin ra *p* là đúng thì *p* là bất biến vòng lặp.

Như vậy giả sử *p* là bất biến vòng lặp. Từ đó suy ra *p* là đúng trước khi đoạn chương trình được thực hiện, *p* và phủ định <điều kiện> là đúng sau khi kết thúc.

Ví dụ 2.5. *Hãy dùng bất biến vòng lặp chứng tỏ thuật toán 2.7 kết thúc với factorial = n!, trong đó n là số nguyên dương.*

Thuật toán 2.7 Tính giai thừa theo vòng lặp (*fact*)

```
function  fact(n)
  Đầu vào:  Cho giá trị n.
  Đầu ra:   Giá trị n!.
  1:  i := 1;
  2:  factorial := 1;
  3:  while i < n do
  4:      i := i + 1;
  5:      factorial := factorial * i;
  6:  end while
  7:  return(factorial)
end  fact
```

Chứng minh. Giả sử *p* là mệnh đề: *factorial* := *i*! *và* $i \leq n$. Ta sẽ chứng minh rằng *p* là bất biến vòng lặp bằng phương pháp quy nạp toán học. Đầu tiên ta nhớ rằng *p* là đúng trước khi vào lặp, vì $i = 1$, $factorial = 1 = 1!$ và $1 \leq n$. Giả sử *p* đúng và $i < n$ sau khi thực hiện vòng lặp và giả sử vòng while được thi hành một lần nữa. Trước tiên *i*

tăng lên 1, như vậy vẫn còn nhỏ hơn hay bằng n. Do giả thiết quy nạp toán học $factorial = (i-1)!$ trước khi vào vòng lặp và nó sẽ được tăng bằng $(i-1)! * i = i!$. Vì thế p vẫn còn đúng. Do đó p là bất biến vòng lặp.

Vì vòng lặp kết thúc sau khi lặp $n-1$ lần, khi đó $i = n$ và $fractorial = n!$. ☺

Ta có thể nói chứng minh thuật toán không hồi quy phải tiến hành các bước sau đây:

1. Phân tích thuật toán tại một vòng lặp, từ khởi đầu đến trong vòng lặp và cho đến vòng lặp sau.
2. Với mỗi vòng lặp để lại *bất biến vòng lặp* mà nó vẫn còn đúng mỗi lần xuyên qua vòng lặp, và ta bắt lấy quy trình tạo ra bởi vòng lặp.
3. Chứng minh những bất biến vòng lặp là đúng.
4. Dùng bất biến vòng lặp để chứng minh thuật toán phải dừng.
5. Dùng bất biến vòng lặp để chứng minh thuật toán tính toán đúng kết quả.

Chú ý : Ta sẽ tập trung xem xét vào một vòng lặp của thuật toán. Giá trị của một biến x ngay sau vòng lặp thứ i được kí hiệu là x_i (với $i = 0$ nghĩa là giá trị trước khi vào vòng lặp thứ nhất). Ví dụ như x_6 là giá trị của biến x sau vòng lặp thứ 6.

Ví dụ 2.6. Thuật toán Fibonacci lặp: *Chứng minh thuật toán 2.8 sau đây đúng (trang 62).*

Chứng minh. Ta chứng minh mệnh đề: $fib(n)$ *trả về* F_n.

Việc chứng minh được tiến hành theo các bước sau đây :

1. Những sự kiện về thuật toán theo kí hiệu đã nói ở trên như sau :

$i : i_0 = 2; i_{j+1} = i_j + 1;$

$a : a_0 = 0; a_{j+1} = b_j;$

Thuật toán 2.8 Tính số fibonacci (*fib*)

```
function fib(n)
  Đầu vào: Cho giá trị n.
  Đầu ra: Giá trị F_n.
  if n = 0 then
     return(0)
  else
     a := 0; b := 1; i := 2;
     while i <= n do
        c := a + b; a := b; b := c; i := i + 1;
     end while
  end if
  return(b)
end fib
```

$b : b_0 = 1; a_{j+1} = c_{j+1};$

$c : c_{j+1} = a_j + b_j.$

2. Vòng lặp bất biến: Mệnh đề sau là vòng lặp bất biến :

Với mọi số tự nhiên $j \geq 0$, thì $i_j = j+2$, $a_j = F_j$ và $b_j = F_{j+1}$.

Ta chứng minh bằng quy nạp theo j. *Bước cơ sở :* $j = 0$, hiển nhiên mệnh đề đúng vì $i_0 = 2$, $a_0 = 0 = F_0$ và $b_0 = 1 = F_1$. *Bước quy nạp :* Giả sử $j \geq 0$, $i_j = j+2$, $a_j = F_j$ và $b_j = F_{j+1}$, ta phải chứng minh rằng $i_{j+1} = j+3$, $a_{j+1} = F_{j+1}$ và $b_{j+1} = F_{j+2}$.

Thật vậy, sau một lần lặp của thuật toán ta có

$$\begin{aligned} i_{j+1} &= i_j + 1 \\ &= (j+2)+1 \quad \text{(theo giả thiết quy nạp)} \\ &= j+3. \end{aligned}$$

$$\begin{aligned} a_{j+1} &= b_j \\ &= F_{j+1} \quad \text{(theo giả thiết quy nạp)} \end{aligned}$$

$$\begin{aligned} b_{j+1} = c_{j+1} &= a + j + b_j \\ &= F_j + F_{j+1} \quad \text{(theo giả thiết quy nạp)} \\ &= F_{j+2}. \end{aligned}$$

3. Chứng minh tính toán đúng:

Thuật toán kết thúc với b chứa giá trị F_n.

Sự đòi hỏi của mệnh đề trên đúng nếu $n = 0$. Nếu $n > 0$ thì ta lại vào vòng lặp, như chứng minh trên cho kết quả đúng.

Tính kết thúc của thuật toán: Vì $i_{j+1} = i_j + 1$, thì hiển nhiên i sẽ phải đến $n+1$ và khi đó vòng lặp sẽ kết thúc. Giả sử thuật toán dừng sau t vòng lặp. Vì $i_t = n+1$ và $i_t = t+2$ ta sẽ tìm được $t = n-1$.

Kết quả trả về: Theo bất biến vòng lặp, $b_t = F_{t+1} = F_n$. Đó là điều cần chứng minh. ☺

Ví dụ 2.7. Thuật toán Maximum lặp: *Tìm số lớn nhất được lưu trong mảng một chiều $A[1..n]$. Chứng minh thuật toán sau đây đúng :*

Thuật toán 2.9 Tìm giá trị cực đại trong mảng (*maximum*)

```
function   maximum(A, n)
  Đầu vào:  Cho giá trị A[1..n].
  Đầu ra:   Giá trị max của A[1..n].
  m := A[1]; i := 2;
  while i <= n do
     if A[i] > m then
        m := A[i]; i := i + 1;
     end if
  end while
  return(m)
end  maximum
```

Chứng minh. Ta chứng minh mệnh đề sau:

Với mọi $n \geq 1$, maximum(n) trả về giá trị $\max\{A[1], A[2], ..., A[n]\}$.

1. Những yếu tố quan trọng thay đổi trong thuật toán:

$m : m_0 = A[1]; m_{j+1} = \max\{m_j, A[i_j]\}$;

$i : i_0 = 2; i_{j+1} = i_j + 1$.

2. Vòng lặp bất biến:

Với mọi số tự nhiên $j \geq 0$, ta có

$$m_j = \max\{A[1], A[2], ..., A[j+1]\} \text{ và } i_j = j+2.$$

Ta chứng minh bằng phương pháp quy nạp theo j. Bước cơ sở dễ thấy $j = 0$ thì $m_0 = A[1]$ và $i_0 = 2$. Giả sử $j \geq 0$, ta có $i_j = j+2$ và $m_j = \max\{A[1], A[2], ..., A[j+1]\}$, ta phải chứng minh rằng $i_{j+1} = j+3$ và $m_{j+1} = \max\{A[1], A[2], ..., A[j+2]\}$.

Thật vậy,

$$\begin{aligned} i_{j+1} &= i_j + 1 \\ &= (j+2)+1 \quad \text{(theo giả thiết quy nạp)} \\ &= j+3. \end{aligned}$$

$$\begin{aligned} m_{j+1} &= \max\{m_j, A[i_j]\} \\ &= \max\{m_j, A[j+2]\} \quad \text{(theo giả thiết quy nạp)} \\ &= \max\{\max\{A[1], A[2], ..., A[j+1]\}, A[j+2]\} \quad \text{(giả thiết quy nạp)} \\ &= \max\{A[1], A[2], ..., A[j+1], A[j+2]\}. \end{aligned}$$

3. Chứng minh tính đúng đắn :

Thuật toán sẽ kết thúc với m chứa giá trị lớn nhất của $A[1..n]$.

Thật vậy, sự kết thúc của thuật toán vì $i_{j+1} = i_j + 1$ hiển nhiên i sẽ đến lúc tới $n+1$ và vòng lặp sẽ dừng. Giả sử thuật toán sẽ dừng sau t vòng lặp. Vì $i_t = t+2$, nên có $t = n-1$.

Kết quả tính toán : Theo bất biến của vòng lặp

$$\begin{aligned} m_t &= \max\{A[1], A[2], ..., A[t+1]\} \qquad (2.1) \\ &= \max\{A[1], A[2], ..., A[n]\} \end{aligned}$$

☺

Ví dụ 2.8. Thuật toán phép nhân lặp : *Chứng minh thuật toán 2.10, phép nhân sau đây đúng (trang 65).*

Chứng minh. Ta phải chứng minh mệnh đề sau:

Với mọi $y, z \geq 0$, *multiply*(y, z) *trả về* yz.

1. Ta chứng minh một bổ đề phụ trợ:

Với mọi $n \in \mathbb{N}$, *ta có*

$$2\left\lfloor \frac{z}{2} \right\rfloor + (n \mod 2) = n.$$

Thuật toán 2.10 Tích hai số nguyên (*multiply*(y,z))

```
function multiply(y,z)
  Đầu vào: Cho giá trị y,z ∈ ℕ.
  Đầu ra: Giá trị tích yz.
  x := 0;
  while z > 0 do
    if z là số lẻ then
      x := x + y;
      y := 2y; z := ⌊z/2⌋;
    end if
  end while
  return(x)
end multiply
```

Thật vậy, ta xét hai trường hợp :

Trường hợp 1, n là số chẵn: Khi đó $\left\lfloor \frac{z}{2} \right\rfloor = \frac{n}{2}$, và $(n \mod 2) = 0$, kết quả đúng.

Trường hợp 2, n là số lẻ: Khi đó $\left\lfloor \frac{z}{2} \right\rfloor = \frac{n-1}{2}$, và $(n \mod 2) = 1$, kết quả đúng.

2. Những yếu tố cơ bản của thuật toán :

$y:\ y_{i+1} = 2y_i;\ z_{j+1} = \left\lfloor \frac{z_j}{2} \right\rfloor;$

$x:\ x_0 = 0;\ x_{j+1} = x_j + y_j(z_j \mod 2).$

3. Vòng lặp bất biến : (mệnh đề đối với biến luôn luôn còn đúng khi mỗi lần đi qua vòng lặp).

Với tất cả những số tự nhiên $j \geq 0$, đẳng thức sau đúng :

$$y_j z_j + x_j = y_0 z_0.$$

Ta chứng minh bằng quy nạp toán học theo j. Bước cơ sở là hiển nhiên

$$y_j z_j + x_j = y_0 z_0 + x_0 = y_0 z_0.$$

Giả sử $j \geq 0$, đẳng thức đúng $y_j z_j + x_j = y_0 z_0$. Ta phải chứng minh rằng

$$y_{j+1} z_{j+1} + x_{j+1} = y_0 z_0.$$

Thật vậy, theo những yếu tố chính của thuật toán liệt kê ở trên ta có

$$\begin{aligned} y_{j+1}z_{j+1} + x_{j+1} &= 2y_j\lfloor\frac{z_j}{2}\rfloor + x_j(z_j \mod 2) \\ &= y_j(2\lfloor\frac{z_j}{2}\rfloor + x_j(z_j \mod 2)) + x_j \\ &= y_jz_j + x_j \quad \text{(do bổ đề hỗ trợ)} \\ &= y_0z_0 \quad \text{(theo giả thiết quy nạp).} \end{aligned}$$

4. Chứng minh tính đúng đắn của thuật toán:

Thuật toán dừng với x chứa giá trị tích của hai số y và z.

a) Thuật toán dừng: Mỗi lần lặp, giá trị của z được chia đôi (được làm tròn xuống nếu nó là lẻ). Do đó đến một vòng lặp nào đó t tại đó $z_t = 0$. Tại thời điểm này thì thuật toán dừng.

b) Kết quả tính toán: Giả sử vòng lặp dừng sau t vòng, với $t \geq 0$. Theo kết quả của bất biến vòng lặp ta luôn có

$$y_tz_t + x_t = y_0z_0.$$

Từ $z_t = 0$ ta thấy ngay $x_t = y_0z_0$. Do đó thuật toán kết thúc với x là tích của hai giá trị ban đầu của y và z. ☺

2.5. BÀI TẬP

▷ **2.9.** Chứng minh thuật toán cộng hai số tự nhiên sau đây đúng :

```
function   add(y,z)
  Đầu vào:   y,z hai số nguyên dương.
  Đầu ra:    y+z.
    x := 0; c := 0; d := 1;
    while (y > 0) ∨ (z > 0) ∨ (c > 0) do
        a := y mod 2; b := z mod 2;
        if a ⊕ b ⊕ c then
            x := x + d;
        end if
        c = (a ∧ b) ∨ (b ∧ c) ∨ (a ∧ c);
        d := 2d; y := ⌊y/2⌋; z := ⌊z/2⌋;
    end while
    return(x);
end   add
```

▷ **2.10.** Chứng minh thuật toán nhân hai số tự nhiên sau đây đúng :

```
function   multiply(y,z)
  Đầu vào:   y,z hai số nguyên dương.
  Đầu ra:    yz.
  x := 0;
  while z > 0 do
      if z mod 2 = 1 then
          x := x + y;
      end if
      y := 2y; z := ⌊z/2⌋;
  end while
  return(x);
end   multiply
```

▷ **2.11.** Chứng minh thuật toán hàm số mũ sau đây đúng :

```
function   power(y,z)
  Đầu vào:   y ∈ ℝ, z ∈ ℕ.
  Đầu ra:    y^z.
  x := 1;
  while z > 0 do
      x := x * y;
      z := z - 1;
  end while
  return(x);
end   power
```

▷ **2.12.** Chứng minh thuật toán cộng các phần tử trong mảng $A[1..n]$ sau đây là đúng :

```
function   sum(A)
  Đầu vào:   mảng A[1..n].
  Đầu ra:    Σ_{i=1}^{n} A[i].
  s := 0;
  for i := 1 to n do
      s := s + A[i];
  end for
  return(s);
end   sum
```

▷ **2.13.** Chứng minh thuật toán hồi quy nhân hai số tự nhiên sau đây là đúng :

```
function   multiply((y,z))
  Đầu vào:   y,z ∈ ℕ.
  Đầu ra:    tích yz.
  if z = 0 then
     return(0)
  else
     if z là số lẻ then
        return(multiply(2y, ⌊z/2⌋) + y)
     else
        return(multiply(2y, ⌊z/2⌋))
     end if
  end if
end   multiply
```

CHƯƠNG 3
PHÂN TÍCH ĐỘ PHỨC TẠP THUẬT TOÁN

Chương trước ta quan tâm tới đáp số đúng của thuật toán và cách chứng minh thuật toán đúng. Nhưng thuật toán hiệu quả lại là một vấn đề khác, có thể có những thuật toán đúng nhưng không hiệu quả. Chương này và những chương sau ta xét vấn đề này cho các thuật toán thông dụng nhất hiện hay.

Một thước đo hiệu quả đó là thời gian mà máy tính sử dụng để giải bài toán theo thuật toán đang xét, khi các giá trị đầu vào có một kích thước xác định. Một thước đo thứ hai là bộ nhớ đòi hỏi để thực hiện thuật toán đó khi giá trị đầu vào có kích thước cho trước. *Độ phức tạp tính toán* của thuật toán bao gồm những thước đo như vậy để so sánh hiệu quả thực hiện của thuật toán. Gắn liền với thời gian tính toán của thuật toán là *độ phức tạp thời gian* và với bộ nhớ là *độ phức tạp không gian*. Biết được độ phức tạp thời gian cho một thuật toán là rất quan trọng, vì khi đó ta biết được thời gian một phút, một năm, một tỉ năm để thực hiện thuật toán đó. Độ phức tạp không gian đòi hỏi của thuật toán mà ta biết được thì cho ta một bước chuẩn bị và thấy được khả năng đáp ứng trong việc tính toán của thuật toán và độ phức tạp này không thể bỏ qua.

Độ phức tạp không gian gắn liền với cấu trúc dữ liệu đặc biệt dùng để tính toán trong thuật toán. Trong tài liệu này chúng ta không nghiên cứu về cơ sở dữ liệu nên ta bỏ qua độ phức tạp không gian.

3.1. ĐÁNH GIÁ THUẬT TOÁN QUA PHÉP TOÁN THỰC HIỆN

Độ phức tạp thời gian của một thuật toán có thể được xem xét qua các phép toán được dùng với các giá trị đầu vào xác định. Các phép toán được dùng để đo độ phức tạp thời gian có thể là phép so sánh các số nguyên, phép cộng, trừ, nhân và chia các số nguyên hoặc bất kì một phép toán sơ cấp nào khác.

Lấy ví dụ đầu vào và các phép toán ta xét đoạn chương trình tính tổng sau:

```
1:  n := 100;
2:  sum := 0;
3:  for i := 2 to n do
4:      for j := 2 to n do
5:          sum := sum + 1;
6:      end for
7:  end for
```

Ta quan tâm tới tốc độ tính toán của đoạn chương trình trên, nghĩa là tiêu chuẩn nào xác định tốc độ của đoạn chương trình đó? Ta có thể đưa đoạn chương trình vào một chương trình cụ thể để thử xem nó thực hiện và kết thúc trong thời gian bao lâu. Ta lại có thể thử với những giá trị n cụ thể khác nhau. Giả sử ta thu được kết quả

$n = 10$	0,000001 giây
$n = 100$	0,0001 giây
$n = 1000$	0,01 giây
$n = 10000$	1,71 giây
$n = 100000$	106,543 giây
$n = 1000000$	10663,6 giây

Dễ thấy rằng trong bảng trên khi n tăng 10 lần thì thời gian thực hiện chương trình tăng lên 100.

Ta phân tích kĩ hơn đoạn chương trình trên. Dòng số 1 và 2 có phép tính cố định và mất một thời gian hằng số, kí hiệu đó là a. Cho các phép toán gán $i := 0$, kiểm tra $i < n$ và $i := i + 1$ cũng chiếm thời gian hằng số (mỗi phép toán này phụ thuộc vào cấu trúc của bộ xử lí cụ thể), ta kí hiệu lần lượt là b, c, d. Tương tự ở dòng 4, thời gian cho các phép toán

$j = 0$; $j < n$; $j := j + 1$ kí hiệu là e, f, g. Cuối cùng phép toán ở dòng 5 đòi hỏi thời gian hằng số là h.

Với những đại lượng ta đưa vào ở trên thì không khó tính được thời gian chung cho thực hiện đoạn chương trình với giá trị bất kì n nào đó:

$$\begin{aligned} T(n) &= a + b + nc + nd + n(e + nf + ng + nh) \\ &= a + b + nc + nd + ne + n^2 f + n^2 g + n^2 h \\ &= n^2(f + g + h) + n(c + d + e) + a + b. \end{aligned}$$

Bởi vì a, b, c, d, e, f, g, h là những hằng số, ta đặt $i = f + g + h$, $j = c + d + e$, $k = a + b$. Khi đó thuật toán thực hiện mất thời gian là

$$T(n) = i.n^2 + j.n + k.$$

Những hằng số i, j, k rất quan trọng cho thực hiện tốc độ của thuật toán, nhưng chưa xác định. Trong thực tế khi nghiên cứu tính hiệu quả của thuật toán đã cho, ta phải quan tâm tới những hằng số này. Những hằng số này trước tiên phụ thuộc vào phần cứng để thể hiện chương trình.

Hơn nữa khi xét tốc độ thuật toán ta có thể bỏ qua cả những số hạng $j.n$ và k và chỉ để lại một số hạng có số mũ cao nhất của n. Tại vì hàm với số mũ cao nhất tăng rất nhanh khi cỡ của dữ liệu vào tăng lên.

Ta xét hai hàm số phụ thuộc n thể hiện thời gian thực hiện hai thuật toán A_1 và A_2: $f = 2.n^2$ và $g = 200.n$. Dễ thấy rằng tuy hệ số của g là rất lớn so với của f, nhưng khi n tăng lên cỡ lớn (ví dụ như $n > 100$) thì thuật toán A_2 tính toán nhanh hơn thuật toán A_1 rất nhiều. Vậy tăng n này lên bao nhiêu, thì tỉ số thời gian thực hiện của hai thuật toán cũng tăng bấy nhiêu có lợi cho A_2. Nhìn vào hàm số của hai thuật toán ta thấy rằng độ phức tạp tính toán của A_2 là tuyến tính, còn của A_1 là bình phương. Vấn đề này liên quan đến các độ đo độ phức tạp tính toán mà ngay chương đầu tiên ta đã xem xét và nghiên cứu.

3.2. XÁC ĐỊNH ĐỘ PHỨC TẠP TÍNH TOÁN

Ta xét một số tính chất mà với chúng ta có thể định nghĩa được độ phức tạp tính toán theo cách thể hiện của giả mã như ở chương 1. Ta sử

dụng kí hiệu T(<khối mã>) cho độ phức tạp tính toán của một toán tử riêng cũng như một đoạn mã chương trình. Khi mã chương trình được tách biệt rõ ràng thì ta kí hiệu $T(n)$ độ phức tạp tính toán là hàm biến số n với n là số phép toán cơ sở. Độ phức tạp tính toán được đo bằng hàm $O(...)$ là chính.

1. Phép tính cơ sở. *Độ phức tạp của toán tử cơ sở là một hằng số, nghĩa là* $O(1)$. Toán tử cơ sở là gì không phải dễ xác định, với những bắt buộc khác nhau cho ta định nghĩa lại phép toán cơ sở. Theo nguyên tắc, phép toán cơ sở là phép toán thực hiện một hằng số thời gian, phụ thuộc vào dung lượng thao tác trên thông tin. Thường thường phép toán cơ bản là các phép tính cộng, trừ, nhân, chia,... Nhưng chú ý rằng khi ta thực hiện số lớn hàng tỉ thì không thể chấp nhận phép nhân các số lớn là một phép tính cơ sở. Cũng không thể nhận phép tính cơ sở là các hàm lượng giác, hàm số mũ, hàm lôgarit,... bởi vì chúng tính toán theo dãy.

2. Dãy các phép tính. *Độ phức tạp thời gian của dãy liên tiếp các phép tính xác định bởi độ phức tạp cao nhất trong chúng.* Một cách hình thức, nếu toán tử s_1 với độ phức tạp F_1, sau đó là toán tử s_2 với độ phức tạp F_2, ta có thể viết

$$T(s_1) \in O(F_1), T(s_2) \in O(F_2) \Rightarrow T(s_1; s_2) \in \max\{O(F_1), O(F_2)\}.$$

Điều trên tương đương với độ đo tiệm cận $f_1 + f_2 \in \max\{O(f_1), O(f_2)\}$.

3. Hợp của các phép tính. *Sự tác động của toán tử trên miền thực hiện của một toán tử khác thì độ phức tạp tính như tích của các độ phức tạp của chúng,* nghĩa là

$$T(s_1) \in O(F_1), T(s_2) \in O(F_2) \Rightarrow T(s_1\{s_2\}) \in \max\{O(F_1.F_2)\}.$$

Điều trên tương đương với độ đo tiệm cận $f_1.f_2 \in \max\{O(f_1.f_2)\}$.

4. Cấu trúc if: if (p) then s_1; else s_2. *Nếu độ phức tạp tính toán của* p, s_1, s_2 *tương ứng là* $O(P), O(F_1), O(F_2)$, *thì độ phức tạp của cấu trúc if là* $\max\{O(P), O(F_1), O(F_2)\}$, nghĩa là độ phức tạp tăng nhanh nhất của hàm P, F_1, F_2.

$$T(p) \in O(P), T(s_1) \in O(F_1), T(s_2) \in O(F_2)$$
$$\Rightarrow T(\text{if } (p) \text{ then } s_1 \text{ else } s_2) \in \max\{O(P), O(F_1), O(F_2)\}$$

Ta thử giải thích quy tắc này. Giả sử điều kiện p là đúng. Khi đó cấu trúc if tương đương với dãy lệnh thứ tự p, s_1, mà dãy này độ

phức tạp được xác định như phần trước là $\max(O(p), O(s_1))$. Cũng như vậy ta nhận được độ phức tạp của cấu trúc nếu p không đúng là $\max(O(p), O(s_2))$. Vì ta chưa biết được khối nào sẽ được thực hiện trong cấu trúc này, nên ta giả thiết rằng độ phức tạp của cấu trúc là độ phức tạp lớn nhất của một trong hai khối này. Ta nhận được $T(\text{if } (p) \text{ then } s_1 \text{ else } s_2) \in \max\{\max(O(P), O(F_1)), \max(O(P), O(F_2))\}$

$$= \max\{O(P), O(F_1), O(F_2)\}.$$

Tương tự ta cũng tính được độ phức tạp của cấu trúc case.

5. Vòng lặp for. Ta xét chu trình tính sau:

```
1:  fact := 1;
2:  for i := 1 to n do
3:      fact := fact * i;
4:  end for
```

Ta có thể giả thiết phần bên trong của chu trình có hằng số thời gian tính c không phụ thuộc vào n. Dễ thấy độ phức tạp của vòng lặp là $O(n)$. Bởi vì theo nguyên tắc kết hợp ở trên thì độ phức tạp toàn bộ chu trình là $O(c.n)$, cũng là $O(n)$. Nếu tính chi tiết ra thì ta phải thêm vào độ phức tạp của khởi tạo biến trước khi có chu trình có độ phức tạp $O(1)$, lại theo quy tắc dãy thứ tự phép toán thì có độ phức tạp $O(1+n)$. Nhưng cuối cùng theo các tính chất của hàm này thì độ phức tạp của chu trình trên là $O(n)$.

6. Vòng lặp lồng nhau. Ta xét hai vòng lặp

```
1:  sum := 0;
2:  for i := 0 to n do
3:      for j := 0 to n do
4:          sum := sum + 1;
5:      end for
6:  end for
```

Độ phức tạp tính toán của chu trình lồng nhau có chỉ số chạy độc lập có thể tính được dễ dàng, vì độ phức tạp của một vòng lặp trong đã là $O(n)$ và độ phức tạp của cả đoạn hai vòng lặp là $T(n) = n.O(n) = O(n^2)$. Ta chú ý hơn chút nữa thì tổng cũng được tính với độ phức tạp $O(n^2)$,

cho rằng mỗi phép tính cộng là hằng số $O(1)$. Do quy tắc dãy liên tiếp các phép tính nên độ phức tạp của đoạn chương trình trên chỉ là $O(n^2)$.

Bây giờ ta xét vòng lặp có chỉ số phụ thuộc nhau

```
sum := 0;
for i := 0 to n − 1 do
    for j := i to n do
        sum := sum + 1;
    end for
end for
```

Dễ thấy phép tính $sum := sum + 1$ thực hiện $\frac{n(n-1)}{2}$ lần và như vậy độ phức tạp của đoạn chương trình trên cũng là $O(n^2)$.

7. Một số ví dụ chu trình lồng nhau. Để làm quen với việc tính độ phức tạp tính toán của một số vòng lặp, các bạn kiểm tra lại độ phức tạp của chu trình sau đây dựa vào các hàm O và quy tắc đã liệt kê trên.

a) Độ phức tạp của đoạn chương trình sau là $O(n^2)$:

```
sum := 0;
for i := 0 to (n − 1) * (n − 1) do
    sum := sum + 1;
end for
```

b) Độ phức tạp của đoạn chương trình sau là $O(n^2)$ vì trong toán tử if thực hiện n^2, tuy chỉ có n lần kết quả kiểm tra $i = j$ là thật sự. Bởi vì độ phức tạp của tổng là tuyến tính nên cuối cùng chỉ có $O(n^2)$.

```
sum := 0;
for i := 0 to n − 1 do
    for j := i to n − 1 do
        if i = j then
            for k := 0 to n − 1 do
                sum := sum + 1;
            end for
        end if
    end for
end for
```

c) Độ phức tạp của đoạn chương trình sau là $O(n^3)$.

```
sum := 0;
for i := 0 to n − 1 do
    for j := 0 to i * i do
        sum := sum + 1;
    end for
end for
```

Vòng lặp bên trong sẽ thực hiện n lần. Bên trong của vòng lặp lại thực hiện

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Do đó độ phức tạp của đoạn chương trình trên là $O(n^3)$.

3.3. ĐỘ PHỨC TẠP CỦA THUẬT TOÁN

Một thuật toán gồm nhiều dòng lệnh hoặc hợp một số đoạn lệnh, phân tích độ phức tạp của một tuật toán đầy đủ cũng như cách làm ở phần trên là phụ thuộc vào cỡ của dữ liệu vào và phép tính cơ sở của thuật toán. Như vậy, phân tích một thuật toán là chỉ ra phép toán cơ sở và đếm xem thuật toán thực hiện bao nhiêu phép toán cơ sở này. Từ đó cho ta thời gian thực hiện của thuật toán mà không phụ thuộc vào phần cứng (vì mỗi phần cứng đều có thời gian để thực hiện phép tính cơ sở). Ta có thể lấy một số ví dụ để chỉ ra phép tính cơ bản của một số bài toán:

Bài toán	**Cỡ đầu vào**	**Phép toán**
Tìm trong một danh sách	Danh sách n phần tử	so sánh
Sắp xếp trong một danh sách	Danh sách n phần tử	so sánh
Nhân hai ma trận	Hai ma trận cỡ $n \times n$	phép nhân
Phân tích ra số nguyên tố	Số n chữ số	phép chia
Bài toán tháp Hà Nội	n đĩa	lấy một đĩa

Bảng 3.1 Độ phức tạp thuật toán

Ta đo độ phức tạp của thuật toán như là một hàm của *cỡ đầu vào* n của thuật toán. Ta nhìn vào bảng trên sẽ thấy độ phức tạp của thuật

toán mà các bài toán đặt ra là các hàm phụ thuộc n. Trong một số thuật toán, số các phép toán cơ bản thực hiện bằng đúng bất kì cỡ đưa vào n. Tuy nhiên, đa số thuật toán số các phép toán cơ bản thực hiện có thể hoàn toàn khác nhau với hai đầu vào cùng một cỡ. Do đó người ta đưa ra khái niệm *độ phức tạp trường hợp tốt nhất, độ phức tạp trường hợp xấu nhất, độ phức tạp trường hợp trung bình* như các hàm của đầu vào cỡ n.

Các định nghĩa hình thức cho độ phức tạp này đòi hỏi phải chỉ ra tập đầu vào cỡ n cho một thuật toán. *Một đầu vào I* cho thuật toán là cơ sở dữ liệu như các số, các chuỗi kí tự, các bản ghi, mà trên nó các phép toán của thuật toán thực hiện tính toán. Kí hiệu $\mathscr{F}_n$ là tập hợp tất cả đầu vào cỡ n cho thuật toán. Cột thứ hai trong bảng 3.1 chỉ ra tương ứng tập hợp $\mathscr{F}_n$ với bài toán khác nhau và tương ứng thuật toán khác nhau. Với $I \in \mathscr{F}_n$, kí hiệu $\tau(I)$ là số lượng phép toán cơ bản mà chúng thực hiện khi thuật toán hoạt động với đầu vào I.

Định nghĩa 3.1. *Độ phức tạp trường hợp tốt nhất* của một thuật toán là hàm số $B(n)$ sao cho $B(n)$ bằng giá trị nhỏ nhất của $\tau(I)$ với sự biến đổi I trên tất cả đầu vào cỡ n. Nghĩa là

$$B(n) = \min\{\tau(I) | I \in \mathscr{F}_n\}.$$

Độ phức tạp trường hợp tốt nhất $B(n)$ dễ dàng tính được, nhưng nó không có ý nghĩa thực tế nhiều để đo độ tính toán của thuật toán. Thật vậy, $B(n)$ chỉ dùng như là cận dưới của độ phức tạp trung bình $A(n)$ của thuật toán. Một độ đo độ phức tạp thuật toán quan trọng hơn là độ phức tạp trường hợp xấu nhất.

Định nghĩa 3.2. *Độ phức tạp trường hợp xấu nhất* của một thuật toán là hàm số $W(n)$ sao cho $W(n)$ bằng giá trị lớn nhất của $\tau(I)$ với sự biến đổi I trên tất cả đầu vào cỡ n. Nghĩa là

$$W(n) = \max\{\tau(I) | I \in \mathscr{F}_n\}.$$

Trong thực tế cho một thuật toán thường được thực hiện với đầu vào cỡ n khác nhau. Một độ đo quan trọng khác cho việc tính toán của thuật toán là độ phức tạp trung bình $A(n)$. Một cách hình thức tổng

quát, giả sử tập hợp $\mathscr{F}_n$ hữu hạn và mỗi đầu vào $I \in \mathscr{F}_n$ có xác suất $p(I)$ xảy ra cho đầu vào với thuật toán.

Định nghĩa 3.3. *Độ phức tạp trung bình* của thuật toán với tập hữu hạn $\mathscr{F}_n$ được định nghĩa

$$A(n) = \sum_{I \in \mathscr{F}_n} \tau(I)p(I).$$

Nếu p_i kí hiệu xác suất mà thuật toán thực hiện đúng i phép toán cơ sở trên đầu vào cỡ n, thì công thức viết lại là

$$A(n) = \sum_{i=1}^{W(n)} ip_i,$$

ở đây $W(n)$ đã được định nghĩa ở trên.

Để hiểu bản chất của định nghĩa độ phức tạp trung bình của thuật toán ta xét một số ví dụ.

Thuật toán tìm kiếm và phân tích độ phức tạp tính toán của chúng

Tìm kiếm phần tử trong một danh sách là một nhiệm vụ thường thấy với máy tính. Ta giả sử danh sách chưa được sắp xếp, thì thuật toán *LinearSearch* làm việc dựa trên cơ sở rà soát tuần tự trên danh sách. Ta phân tích thuật toán này bằng cách tính các hàm số $B(n), W(n)$ và $A(n)$.

LinearSearch thực hiện dựa trên cơ sở so sánh phần tử phải tìm với các phần tử của danh sách.

1. Tìm kiếm tuyến tính

Ta thực hiện *LinearSearch* như một hàm mà nó trả về vị trí trong danh sách $L[1..n]$ khi mà phần tử cần tìm X xuất hiện, còn ngược lại trả về 0 nếu X không có trong danh sách. Phép toán cơ sở của thuật toán *LinearSearch* là so sánh phần tử phải tìm với các phần tử trong danh sách. Dễ thấy, *LinearSearch* thực hiện chỉ một lần so sánh khi phần tử đầu tiên của danh sách là phần tử ta phải tìm, do đó độ phức tạp trường hợp tốt nhất $B(n) = 1$. *LinearSearch* thực hiện phép so sánh nhiều nhất khi X không có trong danh sách hoặc nó ở phần tử cuối cùng của danh sách. Do đó độ phức tạp trường hợp xấu nhất $W(n) = 1$.

Thuật toán 3.1 Tìm kiếm tuyến tính (*LinearSearch*($L[1..n], X$))

```
function  LinearSearch(L[1..n], X)
  Đầu vào:  Cho L[1..n] và X phải tìm.
  Đầu ra:   Chỉ số đầu tiên của danh sách khi X xuất
            hiện, ngược lại là 0.
  for i := 1 to n do
     if X = L[i] then
        return(i)
     end if
  end for
  return(0)
end  LinearSearch
```

Để đơn giản việc phân tích thuật toán cho độ phức tạp tính toán trung bình, ta giả sử X có trong danh sách và sự xuất hiện của nó tại mọi vị trí của $L[1..n]$ đều như nhau. Như vậy, thuật toán thực hiện i phép so sánh khi X được tìm ra ở vị trí i trong danh sách. Vậy xác suất *LinearSearch* thực hiện i phép so sánh tính bằng $p_i = \frac{1}{n}$. Thay giá trị trên vào công thức trong định nghĩa,

$$A(n) = \sum_{i=1}^{W(n)} ip_i = \sum_{i=1}^{n} i\frac{1}{n} = \frac{n(n+1)}{2} \cdot \frac{1}{n} = \frac{n+1}{2}.$$

Công thức trên bằng trực quan ta cũng thấy nó đúng với giả thiết khi X có trong danh sách. Để hiểu công thức trên là trung bình, ta giả sử *LinearSearch* thực hiện m lần (m rất lớn) với danh sách cố định $L[1..n]$ và với việc tìm kiếm X ngẫu nhiên chọn một trong số các phần tử của danh sách. Kí hiệu m_i số lần thực hiện thuật toán mà X được tìm ra tại vị trí i. Khi đó tổng số phép so sánh thực hiện trong m thực hiện thuật toán là $1.m_1 + 2.m_2 + \cdots + n.m_i$. Đem tổng vừa tính được chia cho m được số trung bình $A_m(n)$ phép so sánh trên toàn bộ m lần chạy thuật toán, nghĩa là

$$A_m(n) = 1\left(\frac{m_1}{m}\right) + 2\left(\frac{m_2}{m}\right) + \cdots + n\left(\frac{m_n}{m}\right).$$

Ta chú ý rằng với m lớn, tỉ số $\frac{m_i}{m}$ trong công thức trên xấp xỉ với xác suất p_i của X xuất hiện tại vị trí i trong danh sách. Vì ta giả thiết X vai trò bằng nhau khi tìm ở n vị trí, vậy mỗi m_i xấp xỉ bằng $\frac{m}{n}$. Thay vào công

thức cuối cùng cho kết quả

$$A_m(n) = \frac{1+2+\cdots+n}{n} = \frac{n+1}{2} = A(n).$$

Tóm lại, *LinearSearch* có độ phức tạp trường hợp tốt nhất, độ phức tạp trường hợp xấu nhất, độ phức tạp trung bình tương ứng là $1, n$ và $\frac{n+1}{2}$. Vậy độ phức tạp trường hợp tốt nhất là hằng số không phụ thuộc vào n, trong khi đó độ phức tạp trường hợp xấu nhất và độ phức tạp trung bình phụ thuộc hàm tuyến tính vào n. Để ngắn gọn người ta thường gọi thuật toán *LinearSearch* có độ phức tạp trường hợp tốt nhất hằng số 1, độ phức tạp trường hợp xấu nhất và độ phức tạp trung bình tuyến tính.

2. Tìm phần tử nhỏ nhất và lớn nhất trong danh sách

Bây giờ ta xét bài toán tìm phần tử lớn nhất (hoặc nhỏ nhất) trong danh sách $L[1..n]$. Tìm phần tử lớn nhất có thể thực hiện bằng cách sửa đổi *LinearSearch* bằng cách lưu giá trị lớn nhất lại và cập nhật mới khi có phần tử lớn hơn.

Thuật toán 3.2 Tìm phần tử lớn nhất ($max(L[1..n])$)

```
function   LinearSearch(L[1..n],X)
  Đầu vào:  Cho L[1..n].
  Đầu ra:   Phần tử lớn nhất trong L[1..n].
  1:  MaxValue := L[1]
  2:  for i := 2 to n do
  3:      if L[i] > MaxValue then
  4:          MaxValue := L[i]
  5:      end if
  6:  end for
  7:  return(MaxValue)
end   Max
```

Khi phân tích thuật toán này ta cũng chọn phép toán so sánh làm cơ sở ($L[i] > MaxValue$). Chỉ có một phép toán khác thực hiện là phép gán vào *MaxValue*. Tuy nhiên với một danh sách đưa vào số phép tính so sánh giữa các phần tử của danh sách vượt hơn hẳn (hoặc khống chế) số phép gán vào *MaxValue*, vì vậy ta chọn phép so sánh là phép toán cơ bản.

Ta chú ý rằng *Max* thực hiện $n-1$ phép so sánh khi đầu vào một danh sách bất kì n phần tử. Do đó, độ phức tạp trường hợp tốt nhất, độ phức tạp trường hợp xấu nhất và độ phức tạp trung bình đều bằng nhau và bằng $n-1$. Người ta còn chứng minh được rằng thuật toán tìm phần tử cực đại dựa trên phép toán so sánh cần phải thực hiện ít nhất $n-1$ phép so sánh, như vậy thuật toán trên là tối ưu nhất.

3.4. PHÂN TÍCH ĐỘ PHỨC TẠP THUẬT TOÁN KHÔNG HỒI QUY

Để dễ dàng cho việc phân tích thuật toán ta kết hợp những kí hiệu độ đo độ phức tạp và phần đầu của chương này. Thường người ta phân tích trong trường hợp xấu nhất để biết được thời gian thực hiện thuật toán trong thời gian xấu nhất là bao nhiêu? Người ta định ra độ phức tạp của các phép toán trong trường hợp xấu nhất của thuật toán là:

1. Phép gán được tính độ phức tạp là :$O(1)$.
2. Phép nhập vào thủ tục độ phức tạp là $O(1)$.
3. Phép ra khỏi thủ tục độ phức tạp là $O(1)$.
4. Mệnh đề **if <điều kiện>** độ phức tạp là thời gian so sánh cộng với O(max của hai nhánh).
5. Vòng lặp **(while)** độ phức tạp là tổng tất cả vòng lặp với thời gian của mỗi vòng lặp đó.

Ví dụ 3.1. *Phân tích độ phức tạp trong trường hợp xấu nhất của thuật toán 3.3 nhân hai số (trang 81).*

Lời giải. Giả sử y và z có n bit. Nhìn vào thuật toán 3.3 ta có thể liệt kê :

- Nhập vào và thoát khỏi hàm có độ phức tạp $O(1)$;
- Dòng 3, 4, 5, 6, 7 có độ phức tạp $O(1)$;
- Vòng lặp từ dòng 2 đến dòng 8 có độ phức tạp $O(n)$ (nó thực hiện nhiều nhất n lần);

Thuật toán 3.3 Phép nhân hai số (*multiply*(y,z))

```
function  multiply(y,z)
  Đầu vào:  Cho y,z ∈ ℕ.
  Đầu ra:   Tích yz.
1:  x := 0;
2:  while z > 0 do
3:      if z là số lẻ then
4:          x := x + y;
5:      else
6:          y := 2y; z := ⌊z/2⌋;
7:      end if
8:  end while
9:  return(x)
end  multiply
```

- Dòng 1 có độ phức tạp $O(1)$;

Do đó thuật toán trên có độ phức tạp $O(n)$ (do các nguyên tắc nhân và cộng của hàm O). ☺

Ví dụ 3.2. *Mô tả độ phức tạp thời gian của thuật toán sắp xếp sủi bọt:*

Thuật toán 3.4 Sắp xếp sủi bọt *bubblesort*

```
procedure  bubblesort(A[1..n])
  Đầu vào:  Mảng A[1..n].
  Đầu ra:   Mảng A[1..n] đã được sắp xếp.
1:  for i := 1 to n − 1 do
2:      for j := 1 to n − i do
3:          if A[j] > A[j + 1] then
4:              swap(A[j], A[j + 1]);
5:          end if
6:      end for
7:  end for
end
```

Lời giải. Ta có thể đánh giá thuật toán theo trình tự như sau:

- Nhập vào và thoát khỏi hàm có độ phức tạp $O(1)$;

- Dòng 4 có độ phức tạp $O(1)$;
- Dòng 3 đến 5 có độ phức tạp $O(1)$;
- Vòng lặp từ dòng 2 đến dòng 6 có độ phức tạp $O(n-i)$ (nó thực hiện nhiều nhất $n-i$ lần);
- Vòng lặp từ dòng 1 đến dòng 7 có độ phức tạp $O(\sum_{i=1}^{n-1} n-i)$ (nó thực hiện nhiều nhất $n-i$ lần);

$$O(\sum_{i=1}^{n-1} n-i) = O(n(n-1) - \sum_{i=1}^{n-1} i) = O(n^2).$$

Do đó thuật toán này có độ phức tạp $O(n^2)$ trong trường hợp xấu nhất. Có thể chỉ ra bằng cách tương tự với $\Omega(n^2)$. ☺

Như vậy ta thấy rằng phân tích độ phức tạp của thuật toán được tiến hành lần lượt theo các dòng lệnh. Để thực hiện một cách nhanh chóng và trọng tâm chú ý vào những đoạn quan trọng, bỏ qua được những dòng không cần thiết khi chỉ một lần thực hiện có độ phức tạp $O(1)$. Ta tiến hành trên hai bước cơ bản sau:

1. Định danh được những *phép toán cơ bản* được dùng trong thuật toán và theo dõi độ phức tạp là bội số bất biến của số những phép toán cơ bản được dùng. (Không cần thiết phân tích theo từng dòng).
2. Phân tích chính xác số toán tử (làm việc trên con số hơn là làm việc trên các kí hiệu).

Ta trở lại ví dụ sắp xếp sủi bọt trên. Phép toán cơ bản là so sánh tại dòng 3. Độ phức tạp sẽ là O lớn của số những phép so sánh. Thế thì

- Dòng 3 có 1 phép so sánh;
- Vòng lặp for từ dòng 2 đến 6 dùng $n-i$ phép so sánh;

- Vòng lặp for từ dòng 1 đến 7 dùng $\sum_{i=1}^{n-1}(n-i)$ phép so sánh và khi đó

$$\sum_{i=1}^{n-1}(n-i) = n(n-1) - \sum_{i=1}^{n-1} i = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2}.$$

3.5. PHÂN TÍCH THUẬT TOÁN HỒI QUY

1. Phương pháp phân tích thuật toán hồi quy

Ta biết rằng một thuật toán hồi quy có một cỡ đã cho thường trong lòng nó gọi lại chính nó với cỡ nhỏ hơn. Phân tích những thuật toán hồi quy ta phải quy về các phân tích hàm thời gian chính nó, nhưng với cỡ nhỏ hơn. Ngoài ra trong thuật toán hồi quy thường có những giá trị khởi đầu của thuật toán và nhiều những thao tác khác ngoài hàm hồi quy. Từ cấu trúc thuật toán hồi quy, ta kí hiệu hàm $T(n)$ là hàm thời gian thực hiện của thuật toán ta đang xét có cỡ là n, thì ta phải tìm hàm thời gian theo những bước sau đây:

1. Hãy chỉ ra số n là gì, có phải là cỡ của thuật toán không?
2. Giá trị khởi đầu của n được dùng như cơ sở của hồi quy như thế nào? Thường thường đại lượng này là giá trị duy nhất ($n = 1$), nhưng có nhiều trường hợp nó bao gồm một số đại lượng khác nhau.
3. Hãy chỉ ra giá trị $T(n_0)$ là gì? Thường người ta cho đó là một hằng số c, nhưng với khả năng có thể được thì nên chỉ ra nó là một số cụ thể nếu cần.
4. Hàm thời gian tổng quát $T(n)$ là tổng của một số hàm $T(m)$ (những lời gọi trong thuật toán) và cộng thêm một số tổng những công việc khác đã làm trong thuật toán. Thường thường những lời gọi hồi quy là giải a bài toán con có cỡ $f(n)$, nghĩa là $a.f(n)$ trong mối quan hệ hồi quy.

Ta có công thức tổng quát cho hàm thời gian của thuật toán hồi quy:

$$T(n) = \begin{cases} c & \text{nếu } n = n_0 \\ a.T(f(n)) + g(n) & \text{nếu } n \neq n_0, \end{cases}$$

ở đây c thời gian cho bước cơ sở, n_0 là cơ sở của hồi quy, a là số lần gọi hàm hồi quy, $f(n)$ cỡ của bài toán khi gọi lại hàm hồi quy, $g(n)$ tất cả quá trình khác ngoài lời gọi hàm hồi quy. Phương trình hồi quy này giải bằng định lí tổng quát ở chương 1.

2. Ví dụ

Ví dụ 3.3. *Thiết lập hàm thời gian cho thuật toán 3.5 sau đây :*

Thuật toán 3.5 Ví dụ thuật toán hồi quy *bugs*(n)

```
procedure  bugs(n)
  Đầu vào:  Cho số n.
  Đầu ra:   Kết quả something.
  if n = 1 then
      do something;
  else
      bugs(n − 1);
      bugs(n − 2);
      for i := 1 to n do
          do something;
      end for
  end if
end  bugs
```

Lời giải. Theo quy tắc phần ở trên ta có

$$T(n) = \begin{cases} c & \text{nếu } n = 1 \\ T(n-1) + T(n-2) + g(n) & \text{nếu } n > 1. \end{cases}$$

☺

Ví dụ 3.4. *Thiết lập hàm thời gian cho thuật toán 3.6 (trang 85):*

Lời giải. Theo quy tắc ở phần trên ta có

$$T(n) = \begin{cases} c & \text{nếu } n = 1,2 \\ 2T(n-1) + g(n) & \text{nếu } n > 3. \end{cases}$$

☺

Thuật toán 3.6 Ví dụ thuật toán hồi quy $daffy(n)$

```
procedure  daffy(n)
  Đầu vào:  Cho số n.
  Đầu ra:   Kết quả something.
  if n = 1 or n = 2 then
     do something;
  else
     daffy(n − 1);
     for i := 1 to n do
        do something;
     end for
  end if
  daffy(n − 1);
end  daffy
```

Ví dụ 3.5. *Thiết lập hàm thời gian cho thuật toán 3.7 sau đây :*

Thuật toán 3.7 Ví dụ thuật toán hồi quy $elmer(n)$

```
procedure  elmer(n)
  Đầu vào:  Cho số n.
  Đầu ra:   Kết quả something.
  if n = 1 then
     do something;
  else
     if n = 2 then
        do something;
     else
        for i := 1 to n do
           elmer(n − 1)
           do something;
        end for
     end if
  end if
end  elmer
```

Lời giải. Theo quy tắc phần trên ta có

$$T(n) = \begin{cases} c_0 & \text{nếu } n = 1 \\ c_1 & \text{nếu } n = 2 \\ nT(n-1) + g(n) & \text{nếu } n > 3. \end{cases}$$

☺

Ví dụ 3.6. *Thiết lập hàm thời gian cho thuật toán 3.8 sau đây:*

Thuật toán 3.8 Ví dụ thuật toán hồi quy *yosemite*(n)

```
procedure   yosemite(n)
  Đầu vào:  Cho số n.
  Đầu ra:   Kết quả something.
  if n = 1 then
     do something;
  else
     for i := 1 to n − 1 do
        yosemite(i)
        do somethingnew;
     end for
  end if
end   yosemite
```

Lời giải. Theo quy tắc ở phần trên ta có

$$T(n) = \begin{cases} c & \text{nếu } n = 1 \\ T(1) + T(2) + \cdots + T(n-1) + g(n) & \text{nếu } n > 2. \end{cases}$$

☺

Ví dụ 3.7. *Thiết lập hàm thời gian cho thuật toán 3.9, nhân hai số nguyên* yz *:*

Lời giải. Kí hiệu thời gian thực hiện của thuật toán $T(n)$ trên, ở đây y, z có n-bit của số tự nhiên. Khi đó với $c, d \in \mathbb{R}$, ta có

$$T(n) = \begin{cases} c & \text{nếu } n = 1 \\ T(n-1) + d & \text{nếu } n > 2. \end{cases}$$

Ta có thể giải phương trình hồi quy này bằng nhiều cách, ví dụ bằng cách liệt kê rồi cộng lại : Ta biết rằng với mọi $n >$, $T(n) = T(n-1) + d$.

Thuật toán 3.9 Thuật toán nhân hai số nguyên hồi quy *multiply*(y,z)

```
function   multiply(y,z)
  Đầu vào:   Cho số y,z ∈ N.
  Đầu ra:    Tích yz.
  if z = 0 then
     return(0);
  else
     if z là số lẻ then
        return(multiply(2y, ⌊z/2⌋) + y)
     else
        return(multiply(2y, ⌊z/2⌋))
     end if
  end if
end   multiply
```

Do đó

$$
\begin{aligned}
T(n) &= T(n-1)+d \\
T(n-1) &= T(n-2)+d \\
T(n-2) &= T(n-3)+d \\
&\vdots \\
T(2) &= T(1)+d \\
T(1) &= c
\end{aligned}
$$

Cộng hai vế lại ta nhận được công thức

$$T(n) = dn + c - d.$$

Tất cả những biến đổi trên không phải là chứng minh. Ta có thể chứng minh đẳng thức trên bằng quy nạp toán học:

Đẳng thức đúng với $n = 1$, vì $T(1) = d + c - d = c$.

Giả sử đẳng thức đúng với n ta phải chứng minh rằng $T(n+1) = dn + c$. Thật vậy,

$$T(n+1) = T(n) + d = (dn + c - d) + d = dn + c.$$

Như vậy thuật toán trên có độ phức tạp $T(n) = O(n)$. ☺

3.6. BÀI TẬP

▷ **3.8.** Hàm sau đây trả về giá trị gì? Biểu diễn hàm này như một hàm của n. Dùng O lớn để đánh giá thời gian thực hiện thuật toán trong trường hợp xấu nhất.

```
function mystery(n)
  Đầu vào: Cho số n ∈ ℕ.
  Đầu ra: r = ?.
  r := 0;
  for i := 1 to n − 1 do
     for j := i + 1 to n do
        for k := 1 to j do
           r := r + 1;
        end for
     end for
  end for
  return(r)
end mystery
```

▷ **3.9.** Hàm sau đây trả về giá trị gì? Biểu diễn hàm này như một hàm của n. Dùng O lớn để đánh giá thời gian thực hiện thuật toán trong trường hợp xấu nhất.

```
function conundrum(n)
  Đầu vào: Cho số n ∈ ℕ.
  Đầu ra: r = ?.
  r := 0;
  for i := 1 to n − 1 do
     for j := i + 1 to n do
        for k := i + j − 1 to n do
           r := r + 1;
        end for
     end for
  end for
  return(r)
end conundrum
```

▷ **3.10.** Phân tích thuật toán hồi quy sau

function $g(n)$
Đầu vào: Cho số $n \in \mathbb{N}$.
Đầu ra: Giá trị của hàm g.
1: **if** $n <= 1$ **then**
2: **return**(n)
3: **else**
4: **return**$(5.g(n-1) - 6.g(n-2))$
5: **end if**
end g

Có bao nhiêu phép cộng được dùng trong trường hợp xấu nhất?

Chương 4
PHƯƠNG PHÁP CHIA ĐỂ TRỊ

Từ thời Đế chế La Mã đã áp dụng hiệu quả một nguyên lí chia để trị. Với tư tưởng bành trướng rộng lớn về đất đai và lãnh thổ, những nhà chính trị thời đó đã chia đất đai chiếm được thành những miền nhỏ và áp đặt ách cai trị một cách dễ dàng tại các miền nhỏ này. Ngày nay phương pháp này vẫn còn được áp dụng trong nhiều lĩnh vực của đời sống. Phương pháp này cũng rất hiệu quả khi ta đi thiết kế thuật toán cho những bài toán cỡ lớn, phức tạp. Từ những bài toán có đầu vào rất lớn ta chia ra thành những phần nhỏ hơn và đi tìm lời giải cho các bài toán nhỏ riêng biệt này, rồi sau đó tổng hợp những nghiệm bài toán nhỏ thành nghiệm bài toán toàn cục. Với tư tưởng như vậy, một bài toán ta có thể phải chia nhỏ nhiều lần, chia cho đến khi bài toán được

chia nhỏ chỉ còn lại rất nhỏ và lời giải khi đó là hiển nhiên. Tóm lại, những yếu tố chính của *phương pháp giải chia để trị* gồm:

1. **Chia bài toán:** Chia bài toán thành những bài toán nhỏ hơn.

2. **Trị bài toán nhỏ:** Giải những bài toán nhỏ này.

3. **Kết hợp các nghiệm:** Kết hợp những nghiệm của bài toán nhỏ cho nghiệm của bài toán lớn.

Có rất nhiều bài toán tính toán có thể giải hiệu quả bằng phương pháp chia để trị. Phương pháp này rất mạnh khi thiết kế thuật toán, thậm chí có người còn khuyên việc đầu tiên giải bài toán là hãy đặt ra câu hỏi: Có tồn tại phương pháp chia để trị cho bài toán này không?

Thuật toán thiết kế theo cách chia để trị điển hình là thuật toán hồi quy, vì phần "Trị các bài toán nhỏ" đòi hỏi cùng một kĩ thuật trên những bài toán nhỏ hơn. Phân tích thời gian tính toán của thuật toán hồi quy có phức tạp hơn, nhưng ta sẽ chỉ ra rằng công cụ toán học *phép truy toán* rất có ích cho việc phân tích thuật toán loại này theo một cách tự nhiên.

Phần sau đây ta xem xét một số bài toán theo phương pháp chia để trị với phương thức chia khác nhau, cách trị giải các bài toán con đa dạng và cuối cùng là kết hợp nghiệm một cách tối ưu như thế nào.

4.1. THUẬT TOÁN SẮP XẾP TRỘN

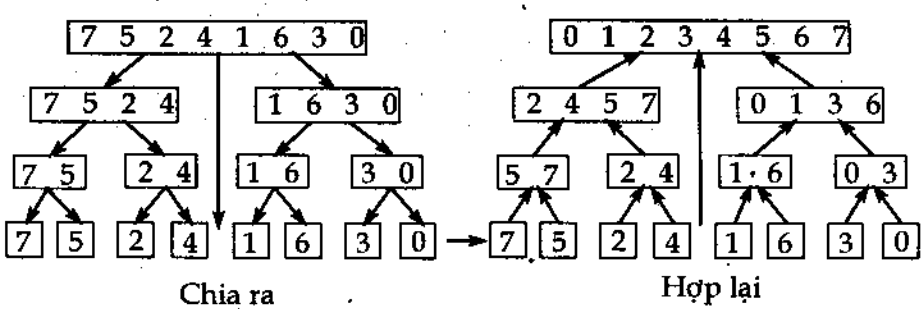
Phần trước ta đã xét một số cách sắp xếp những dãy số được lưu trong một mảng. Bằng cách thiết kế chia để trị ta xây dựng một thuật toán hiệu quả hơn nhiều. Bài toán sắp xếp đặt ra là: Cho A là một dãy n số, ta giả sử lưu các số này vào một mảng $A[1..n]$. Với đầu vào như vậy đòi hỏi hoán vị những dãy số trong mảng để đầu ra là một dãy đã được sắp với thứ tự tăng dần. Áp dụng phương pháp chia để trị để sắp xếp như thế nào? Ta gọi tên thủ tục sẽ thiết kế là *MergeSort*. Những thành phần cơ bản của thuật toán là

1. **Chia bài toán:** Chia A thành hai dãy con tại phần tử ở khoảng giữa, mỗi bài toán con có cỡ làm tròn của $\frac{n}{2}$.

2. **Trị bài toán nhỏ:** Sắp xếp mỗi bài toán con này, bằng cách gọi hồi quy *MergeSort* cho mỗi bài toán con.

3. **Kết hợp các nghiệm:** Trộn hai dãy con đã sắp xếp thành một dãy duy nhất được sắp xếp.

Quá trình chia nhỏ trên kết thúc khi ta có dãy con được chia chỉ còn một phần tử. Một dãy có độ dài bằng 1 thì hiển nhiên đã được sắp xếp. Một thao tác quan trọng trong các bước trên là tất cả công việc được kết thúc trong giai đoạn kết hợp trộn hai dãy đã sắp xếp thành một dãy đơn được sắp. Nhưng quá trình trộn được thực hiện rất dễ dàng, sẽ được lí giải ở phần sau.



Hình 4.1 Thuật toán mergesort

Hình 4.1 cho ta thấy các bước của thuật toán. Giai đoạn chia ra được thể hiện ở bên trái hình. Giai đoạn này được thực hiện từ trên xuống chia nhỏ danh sách thành danh sách con, để trị và kết hợp được mô tả ở bên phải hình. Giai đoạn này được thực hiện từ dưới lên, trộn những danh sách đã được sắp thành danh sách được sắp lớn hơn.

4.1.1. Thiết kế thuật toán

Ta thiết kế thuật toán từ trên xuống như phần nhận xét ở trên. Ta giả sử thủ tục trộn hai danh sách đã được xếp đã được thiết kế xong, kí hiệu là merge (phần sau ta tiến hành cụ thể). Vì thuật toán được gọi hồi

quy trên những danh sách con, được thể hiện bằng các mảng dữ liệu, nên ta truyền thủ tục bằng hai chỉ số, mà nó là chỉ số đầu tiên và chỉ số cuối cùng của mảng con để ta sắp xếp. Như vậy việc sắp xếp trên một mảng được gọi là thủ tục *mergesort*(A,p,q). Thủ tục này sẽ sắp xếp mảng con $A[p..q]$ và trả về kết quả đã sắp xếp trong mảng con này.

Ta có thể mô tả tổng quát thuật toán như sau: Nếu $p = q$, thì nghĩa là mảng chỉ có một phần tử và thủ tục trả về ngay kết quả đó. Nếu $p < q$, thì tồn tại ít nhất hai phần tử và ta sẽ thực hiện chia để trị. Nghĩa là ta tìm một chỉ số r ở giữa p và q, đó là $r = \frac{p+q}{2}$ (được làm tròn xuống số nguyên dưới nó). Khi đó ta chia mảng đã cho thành hai mảng con $A[p..r]$ và $A[r+1..q]$, sau đó gọi *mergesort* hồi quy để sắp xếp mỗi mảng con này. Cuối cùng, ta gọi thủ tục *merge* để trộn hai mảng con thành mảng đơn được sắp xếp.

Thuật toán 4.1 Sắp xếp trộn (mergesort)

procedure *mergesort*(A,p,q)
Đầu vào: Mảng $A[p..q]$
Đầu ra: Mảng A đã được sắp xếp
1: **if** $p < q$ **then**
2: $r := \left\lfloor \frac{p+q}{2} \right\rfloor$;
3: *mergesort*(A,p,r); {Sắp xếp $A[p..r]$}
4: *mergesort*$(A,r+1,q)$; {Sắp xếp $A[r+1..q]$}
5: *merge*(A,p,q,r); {Trộn hai mảng với nhau}
6: **end if**
end *mergesort*

Như vậy chỉ còn thiết kế thủ tục trộn hai mảng lại thành một mảng được sắp *merge*(A,p,q,r). Ta giả sử mảng bên trái $A[p,r]$ và mảng bên phải $A[r+1,q]$ đã được sắp. Ta trộn hai mảng con này bằng cách chép những phần tử của nó vào một mảng trung gian B. Ta lấy mảng B có cùng khoảng chỉ số như mảng A, đó là $B[p..q]$. Ta phải đánh chỉ số i và j chỉ ra những phần tử hiện thời thực hiện trong mỗi dãy con. Ta lấy phần tử nhỏ hơn đặt tiếp vào vị trí của B (được chỉ ra bằng chỉ số k) và sau đó tăng tương ứng chỉ số (hoặc là i hoặc là j). Khi ta đã chạy hết

những phần tử của một mảng con, thì ta chỉ việc chép những phần tử còn lại của mảng con kia vào B. Cuối cùng, ta chép toàn bộ nội dung của B vào A. (Việc dùng mảng trung gian là một điều hạn chế của thuật toán mà rất khó khắc phục. Đây cũng là một điểm yếu của thuật toán *mergesort* so với những thuật toán hiệu quả khác).

Thuật toán 4.2 Trộn hai mảng con (merge)

```
procedure   merge(A,p,r,q)
  Đầu vào:  Mảng A[p..q] và r
  Đầu ra:   Mảng A đã được sắp xếp
  array B[p..q];
  i := k = p; j := r + 1;      {Gán giá trị ban đầu i, j}
  while (i <= r and j <= q) do
     if A[i] <= A[j] then
        B[k+1] := A[i+1];      {Chép vào B từ mảng bên trái}
     else
        B[k+1] := A[j+1];      {Chép vào B từ mảng bên phải}
     end if
  end while{Lặp khi hai mảng cùng khác trống}
  while i <= r do
     B[k+1] := A[i+1]; {Chép vào B phần còn lại của mảng bên
     trái}
  end while
  while j <= q do
     B[k+1] := A[j+1]; {Chép vào B phần còn lại của mảng bên
     phải}
  end while
  for i := p do
     A[i]:=B[i];      {Chép B vào A}
  end for
end   merge
```

Ta chú ý rằng hai vòng lặp **while** cuối cùng chỉ có một trong hai vòng lặp này được thực hiện. Do tại vòng lặp trước đó khi thoát ra chỉ khi một mảng con đã vét hết số phần tử. Bạn đọc có thể có cảm nhận lẫn lộn trong các vòng lặp hồi quy của thuật toán, một cách tốt nhất là ta đối chiếu lại với ví dụ trong hình 4.1 ở trên.

Chú ý : 1. Một trong những mẹo nhỏ để giảm thời gian thực hiện của thuật toán trộn trên là tránh việc chép dữ liệu từ mảng A sang mảng B và ngược lại mỗi lần gọi hồi quy. Người ta thường thực hiện đồng thời trên hai mảng cùng cỡ một lúc. Tại bậc lẻ của lần gọi hồi quy ta trộn những mảng con của A vào những mảng con của B. Tại những bậc chẵn của phép gọi hồi quy ta trộn từ B vào A. Nếu tổng thể thuật toán có số bậc hồi quy lẻ, thì ta phải làm thêm một lần cuối cùng chép B vào A, tuy vậy còn nhanh hơn rất nhiều mỗi lần gọi hồi quy ta phải chép lại. Tất nhiên việc thực hiện này chỉ là yếu tố hằng số, nó không ảnh hưởng hàm đánh giá thời gian thực hiện.

2. Một mẹo khác để thực hiện thuật toán được nhanh hơn là tránh cho việc chia để trị lúc nào cũng có những mảng con có cỡ là 1, thay vào đó người ta dùng những thuật toán khác khi cỡ của mảng rơi vào một hằng số nào đó, ví dụ như cỡ 20 chẳng hạn. Có rất nhiều thuật toán sắp xếp với cỡ mảng nhỏ thì thời gian thực hiện rất nhanh và có hàm đánh giá là $\Theta(n^2)$. Ta chú ý rằng khi đó thuật toán trộn chạy với dãy con cỡ cao nhất là 20, thì thời gian chạy là $\Theta(20^2) = \Theta(1)$. Tuy rằng tất cả những thiết kế này không ảnh hưởng đến hàm đánh giá thời gian của thuật toán trộn.

4.1.2. Phân tích thuật toán

Ta xét thời gian thực hiện của thủ tục *merge*(A, p, r, q). Ta đặt độ dài của hai mảng con là $n = q - p + 1$. Thời gian thực hiện *merge* như là hàm của n là như thế nào? Thủ tục chứa bốn vòng lặp độc lập. Dễ thấy rằng mỗi vòng lặp thực hiện n lần. Như vậy thời gian thực hiện của thủ tục *merge* chỉ là $\Theta(n)$, vì theo tính chất tổng của các hàm đánh giá như chương trước ta đã biết. Ta có thể bỏ qua hằng số và viết thời gian chạy của thủ tục *merge* là n. Bây giờ ta xét thời gian thực hiện của hàm *mergesort*(A, p, q). Ta phân tích theo công thức truy hồi, nghĩa là một hàm định nghĩa hồi quy bởi đại lượng chính nó. Để tránh vòng lặp vô tận phép truy hồi một giá trị đã cho n được định bởi số hạng có nhận giá trị nhỏ hơn thực sự n. Cuối cùng phép truy hồi có một giá trị cơ sở nào đó, ví dụ như $n = 1$, được định nghĩa tường minh.

Sử dụng tư tưởng trên vào thủ tục *mergesort*. Kí hiệu $T(n)$ thời gian thực hiện trong trường hợp xấu nhất của *mergesort* trên mảng cỡ n. Trong thuật toán 4.1 ta có thể tính: Số phép toán gán, số phép so sánh, số truy nhập mảng, vì những những thao tác cơ bản chỉ khác nhau một yếu tố hằng số. Tất cả những phép toán cơ bản được dùng cho việc tính tổng thời gian thực hiện của thuật toán. Thêm vào đó là thời gian thực hiện của thủ tục *merge* đã tính ở trên.

Ta thấy rằng nếu thuật toán *mergesort* được thực hiện với mảng chỉ có một phần tử, thì thời gian thực hiện của nó là một hằng số. Vì ta bỏ qua nhân tố hằng số, ta có thể viết trong trường hợp này $T(n) = 1$. Khi ta gọi thủ tục *mergesort* với mảng có cỡ $n > 1$, nghĩa là $mergesort(A, p, q)$ với $n = q - p + 1$, thuật toán tính $r := \left\lfloor \frac{p+q}{2} \right\rfloor$. Mảng con $A[p..r]$ chứa $r - p + 1$ phần tử, ta có thể kiểm tra đó là $\left\lceil \frac{n}{2} \right\rceil$. Như vậy phần còn lại của dãy con $A[r+1..q]$ có $\left\lfloor \frac{n}{2} \right\rfloor$ phần tử. Thời gian để thực hiện mảng bên trái ta chưa biết được, nhưng vì $\left\lceil \frac{n}{2} \right\rceil < n$ với $n > 1$, nên ta có thể biểu diễn dưới dạng $T(\left\lceil \frac{n}{2} \right\rceil)$. Tương tự như vậy thời gian để thực hiện sắp xếp mảng con bên phải là $T(\left\lfloor \frac{n}{2} \right\rfloor)$. Tổng hợp lại với thủ tục trộn ở trên ta có mối liên hệ truy hồi

$$T(n) = \begin{cases} 1 & \text{nếu } n = 1 \\ T(\left\lceil \frac{n}{2} \right\rceil) + T(\left\lfloor \frac{n}{2} \right\rfloor) + n & \text{nếu } n > 1 \end{cases}$$

Như phần trên đã nói, ta lấy hàm đánh giá của thủ tục trộn là n, có thể lấy cn với c là một hằng số nào đó. Điều này có thể gây ra việc tính toán khó khăn một chút nhưng cuối cùng cũng không ảnh hưởng gì đến hàm đánh giá của thuật toán nói chung.

Ta có thể thử các giá trị của công thức truy hồi trên bằng các giá trị:

$$T(1) = 1$$
$$T(2) = T(1) + T(1) + 2 = 1 + 1 + 2 = 4$$
$$T(3) = T(2) + T(1) + 3 = 4 + 1 + 3 = 8$$
$$T(4) = T(2) + T(2) + 4 = 4 + 4 + 4 = 12$$

$$T(5) = T(3) + T(2) + 5 = 8 + 4 + 5 = 17$$

$$\ldots$$

$$T(8) = T(4) + T(4) + 8 = 12 + 12 + 8 = 32$$

$$\ldots$$

$$T(16) = T(8) + T(8) + 16 = 32 + 32 + 16 = 80$$

$$\ldots$$

$$T(32) = T(16) + T(16) + 32 = 80 + 80 + 32 = 192$$

$$\ldots$$

Với n bất kỳ thì chưa tìm ra một quy luật nào. Ta có thể thấy với n là lũy thừa của 2 thì khi đó ta có

$$\frac{T(1)}{1} = 1, \qquad \frac{T(8)}{8} = 4,$$
$$\frac{T(2)}{2} = 2, \qquad \frac{T(16)}{16} = 5,$$
$$\frac{T(4)}{4} = 3, \qquad \frac{T(32)}{32} = 6.$$

Thế thì ta có thể dự đoán là $\frac{T(n)}{n} = \lg n + 1$, hay tương đương với $T(n) = n\lg n + n$, nghĩa là $T(n) = \Theta(n\lg n)$. Nhưng đó không phải là chứng minh. Có rất nhiều cách chứng minh đẳng thức sau cùng này như chương trước đã nói tới. Ta sẽ chứng minh bằng phương pháp quy nạp, nhưng trước đó ta chú ý tới nhận xét về các phép toán $\left\lfloor \frac{n}{2} \right\rfloor$ và $\left\lceil \frac{n}{2} \right\rceil$. Trong những biểu thức có những hàm như thế này ta có thể bỏ đi các hàm này chỉ còn đại lượng n bằng cách giả thiết rằng n là bội của 2. Ta chú ý là như vậy việc phân tích của ta chỉ đúng cho một tập hợp hạn chế của giá trị n, nhưng những giá trị khác lại nằm trong một lũy thừa 2 nào đó, khi đó hàm đánh giá là một bất đẳng thức nên nó đúng với mọi n. Như vậy với n là lũy thừa của 2 thì biểu thức truy hồi có dạng

$$T(n) = \begin{cases} 1 & \text{nếu } n = 1, \\ 2T(\frac{n}{2}) + n & \text{nếu } n > 1. \end{cases}$$

Mệnh đề 4.1. *Với mọi $n \geq 1$, n là bội của 2, công thức sau đây đúng $T(n) = n\lg n + n$, ở đây* lg *là lôgarit cơ số* 2.

Chứng minh. Chứng minh bằng phương pháp quy nạp dạng 1 theo n.

Bước cơ sở: Với $n = 1$, ta có $T(1) = 1$ theo định nghĩa và công thức $1 \lg 1 + 1 = 1 = T(1)$, mệnh đề đúng.

Bước quy nạp: Cho $n > 1$ và giả sử công thức $T(k) = k \lg k + k$ đúng với mọi $k < n$. Ta phải chứng minh công thức đúng cho chính n. Muốn vậy ta phải biểu diễn $T(n)$ thành thừa số với những giá trị nhỏ hơn. Để làm điều đó ta áp dụng định nghĩa

$$T(n) = 2T(\frac{n}{2}) + n.$$

Do $\frac{n}{2} < n$, ta có thể áp dụng giả thiết quy nạp với $T(\frac{n}{2}) = \frac{n}{2} \lg \frac{n}{2} + \frac{n}{2}$. Tổng hợp lại ta có

$$\begin{aligned} T(n) &= 2\left(\frac{n}{2} \lg \frac{n}{2} + \frac{n}{2}\right) + n = (n \lg \frac{n}{2} + n) + n \\ &= n(\lg n - \lg 2) + 2n = (n \lg n - n) + 2n = n \lg n + n, \end{aligned}$$

đó là điều ta cần chứng minh. ☺

4.2. THUẬT TOÁN NHÂN HAI MA TRẬN

Nhân hai ma trận cùng cỡ là một ví dụ điển hình cho việc thiết kế thuật toán chia để trị. Ta có hai ma trận Y và Z cỡ $n \times n$. Theo định nghĩa phép nhân hai ma trận ta có công thức

$$x_{ij} = \sum_{k=1}^{n} y_{ik} z_{kj},$$

ở đây y_{ik} và z_{kj} là những phần tử của các ma trận tương ứng Y, Z, còn x_{ij} là phần tử trong ma trận tích kí hiệu là X. Bạn đọc có thể dễ dàng lập thành thuật toán như sau: Giả sử tất cả các phép toán trên số nguyên đều tốn thời gian là $O(1)$. Thuật toán nhân ma trận thông thường trên qua ba vòng lặp lồng nhau đều từ 1 đến n nên tổng thời gian phải là $T(n) = O(n^3)$.

Thuật toán trên rất đơn giản và hiển nhiên, một thời gian dài không ai đưa ra một phương pháp nhân hai ma trận khác và không ai thử đi tìm những phương pháp khác. Mãi đến cuối những năm sáu mươi của thế kỉ trước Strassen đưa ra một phương pháp làm hiệu quả hơn thuật

Thuật toán 4.3 Nhân hai ma trận theo định nghĩa (*matmultiply*)

```
procedure  matmultiply (X,Y,Z)
  Đầu vào:  Ma trận Y và Z cỡ n × n
  Đầu ra:   Ma trận X là tích của hai ma trận đầu vào.
  for i := 1 to n do
      for j := 1 to n do
          for k := 1 to n do
              x[i, j] := x[i, j] + y[i, k] * z[k, j];
          end for
      end for
  end for
end  matmultiply
```

toán trên và chứng minh được độ phức tạp tính toán chỉ còn $O(n^{\lg 7})$. Phát minh của Stressen làm ngạc nghiên giới khoa học và đưa phương pháp chia để trị là một phương pháp thiết kế thuật toán có hiệu quả, trước đó nhiều người đã đánh giá không cao về phương pháp này.

4.2.1. Cách tiếp cận chia để trị

Như mục trước ta giả sử cỡ n là lũy thừa của 2. Ta chia mỗi ma trận X, Y, Z thành bốn ma trận nhỏ có cỡ là $\frac{n}{2} \times \frac{n}{2}$ như sau:

$$X = \begin{bmatrix} I & J \\ K & L \end{bmatrix} \quad Y = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \quad Z = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Khi đó có những đẳng thức ma trận sau:

$$I = AE + BG, \quad J = AF + BH,$$
$$K = CE + DG, \quad L = CF + DH.$$

Đặt $T(n)$ là tổng thời gian nhân hai ma trận cỡ $n \times n$. Theo cách nhân như trên thì ta có quan hệ hồi quy như sau:

$$T(n) = \begin{cases} c & \text{nếu } n = 1 \\ 8T(\frac{n}{2}) + dn^2 & \text{ngược lại} \end{cases}$$

ở đây c, d là những hằng số.

Do đó

$$
\begin{aligned}
T(n) &= 8T(\frac{n}{2}) + dn^2 = 8(8T(\frac{n}{4}) + d(\frac{n}{2})^2) + dn^2 = \\
&= 8^2T(\frac{n}{4}) + 2dn^2 + dn^2 = 8^3T(\frac{n}{8}) + 4dn^2 + 2dn^2 + dn^2 = \\
&\vdots \\
&= 8^iT(\frac{n}{2^i}) + dn^2\sum_{j=0}^{i-1} 2^j = 2^{\lg n}T(1) + dn^2\sum_{j=0}^{\lg n-1} 2^j = \\
&= cn^3 + dn^2(n-1) = O(n^3).
\end{aligned}
$$

Như vậy hiệu quả của thuật toán tiếp cận chia để trị theo kiểu như ở trên không khá gì hơn kiểu nhân hai ma trận như định nghĩa.

4.2.2. Cách nhân hai ma trận của Strassen

Vẫn cách tiếp cận chia để trị như trên và những ma trận được chia làm bốn ma trận nhỏ cũng kí hiệu như trên. Nhưng để tránh gặp phải 8 phép nhân và 4 phép cộng như ở trên, Strassen đã đặt các ma trận bằng 7 phép nhân ma trận, còn ma trận kết quả được tính bằng các phép cộng hoặc trừ ma trận cho nhau, do các hàm tiệm cận không thay đổi với các phép cộng và phép trừ mà hàm đánh giá giảm đi đáng kể. Ta thể hiện bằng mệnh đề sau:

Mệnh đề 4.2. *Đặt những ma trận trung gian bằng các biểu thức:*

$$
\begin{aligned}
M_1 &:= (A+C)(E+F), & M_5 &:= (C+D)E, \\
M_2 &:= (B+D)(G+H), & M_6 &:= (A+B)H, \\
M_3 &:= (A-D)(E+H), & M_7 &:= D(G-E). \\
M_4 &:= A(F-H),
\end{aligned}
$$

Khi đó kết quả ma trận nhân được biểu diễn bằng:

$$
\begin{aligned}
I &= M_2 + M_3 - M_6 - M_7, & J &= M_4 + M_6, \\
K &= M_5 + M_7, & L &= M_1 - M_3 - M_4 - M_5.
\end{aligned}
$$

Chứng minh. Ta đi kiểm tra I, J, K, L có thực sự là tích của hai ma trận Y, Z hay không ?

$$\begin{aligned}
I &= M_2 + M_3 - M_6 - M_7 \\
&= (B+D)(G+H) + (A-D)(E+H) - (A+B)H - D(G-E) \\
&= (BG+BH+DG+DH) + (AE+AH-DE-DH) + \\
&\qquad + (-AH-BH) + (-DG+DE) = \boxed{BG+AE}. \\
J &= M_4 + M_6 = A(F-H) + (A+B)H \\
&= AF - AH + AH + BH = \boxed{AF+BH}. \\
K &= M_5 + M_7 = (C+D)E + D(G-E) \\
&= CE + DE + DG - DE = \boxed{CE+DG}.
\end{aligned}$$

$$\begin{aligned}
L &= M_1 - M_3 - M_4 - M_5 \\
&= (A+C)(E+F) - (A-D)(E+H) - A(F-H) - (C+D)E \\
&= AE + AF + CE + CF - AE - AH \\
&\qquad + DE + DH - AF + AH - CE - DE = \boxed{CF+DH}.
\end{aligned}$$

Như vậy, cách tính theo I, J, K, L cũng có kết quả như là cách tính của thuật toán trước đây. ☺

Thuật toán Strassen được mô tả như sau : Từ mệnh đề trên ta có thể dễ dàng thiết lập thuật toán 4.4 (trang 102).

Trong thuật toán này nếu những ma trận vuông, nhưng n không phải lũy thừa của 2, thì ta thêm vào những hàng, những cột chứa toàn số 0 cho đến số lũy thừa của 2 gần nhất với n. Kĩ thuật này gọi là phần thêm tĩnh. Điều đó có thể dẫn đến nhiều nhất là tăng gấp đôi cỡ của ma trận. Việc thực hiện thêm một lần tính bên trong theo quan hệ hồi quy có thể tăng số lượng phép nhân lên nhiều nhất 7 lần. Việc thêm những hàng và cột vào chỉ làm chậm việc thực hiện của thuật toán nhiều nhất với hằng số 7, mà nó không phụ thuộc vào n.

Với thuật toán này số lượng phép cộng đã tăng lên 18. Điều này dẫn đến việc nghi ngờ thuật toán Strassen trong các trường hợp độ chênh lệch tính toán của các phép tính với những bộ vi xử lí. Những bộ vi xử lí thời cũ thường thực hiện phép cộng nhanh gấp hai, ba lần thực hiện

Thuật toán 4.4 Nhân hai ma trận theo Strassen (*matproduct*)

procedure *matproduct*(Y, Z, n)
Đầu vào: Ma trận Y và Z cỡ n
Đầu ra: Ma trận X là tích của hai ma trận đầu vào.

```
if n không là lũy thừa của 2 then
    Thêm vào những hàng và những cột chứa toàn số 0
    tới khi cỡ của chúng đạt lũy thừa của 2 gần nhất, thay đổi n;
end if
if n = 1 then
    X := YZ;
else
    M1 := matproduct(A + C, E + F, n/2);
    M2 := matproduct(B + D, G + H, n/2);
    M3 := matproduct(A − D, E + H, n/2);
    M4 := matproduct(A, F − H, n/2);
    M5 := matproduct(C + D, E, n/2);
    M6 := matproduct(A + B, H, n/2);
    M7 := matproduct(D, G − E, n/2);
    I := M2 + M3 − M6 − M7;
    J := M4 + M6;
    K := M5 + M7;
    L := M1 − M3 − M4 − M5;
    X := Ghép I, J, K, L;
end if
end
```

phép nhân. Nhưng ngày nay với những bộ vi xử lí hiện đại việc thực hiện phép cộng và phép nhân có cùng một tốc độ. Thế thì thuật toán có cho ta tốc độ tính toán tốt hơn không? Ít nhất là không thấy trực tiếp. Strassen đã nhấn mạnh rằng thuật toán có thể ứng dụng hồi quy và cho kết quả tốt hơn.

Thuật toán trên đã sử dụng 7 phép nhân ma trận cỡ $\frac{n}{2}$, đáng lẽ ra là 8. Phép nhân hai ma trận được xác định với việc thực hiện nhiều hơn phép cộng. Thật vậy, nếu dùng thuật toán theo định nghĩa cơ bản của phép cộng và phép nhân hai ma trận, thì phép cộng hai ma trận cần có $(\frac{n}{2})^2$ phép toán cộng cơ bản, còn phép nhân cần tới $(\frac{n}{2})^3$ phép cộng cơ

bản cộng với $(\frac{n}{2})^3$ phép nhân cơ bản nữa. Vì vậy việc tiết kiệm phép nhân sẽ cho ta hiệu quả thực hiện thuật toán tốt hơn.

Phân tích thuật toán Strassen

Theo mệnh đề trên việc tính tích hai ma trận được đưa về 7 bài toán con nhân hai ma trận với cỡ được giảm đi một nửa và với 18 phép cộng ma trận. Ta có thể viết công thức truy hồi với $n > 2$

$$T(n) = 7T(\frac{n}{2}) + 18(\frac{n}{2})^2.$$

Tóm lại ta có công thức

$$T(n) = \begin{cases} c & \text{nếu } n = 1 \\ 7T(\frac{n}{2}) + dn^2 & \text{ngược lại}. \end{cases}$$

ở đây $c, d = \frac{9}{2}$ là những hằng số.

Do đó

$$\begin{aligned} T(n) &= 7T(\frac{n}{2}) + dn^2 = 7(7T(\frac{n}{4}) + d(\frac{n}{2})^2) + dn^2 = \\ &= 7^2T(\frac{n}{4}) + 7d\frac{n^2}{4} + dn^2 = 7^3T(\frac{n}{8}) + 7^2d\frac{n^2}{4} + 7d\frac{n^2}{4} + dn^2 = \\ &\cdots \\ &= 7^iT(\frac{n}{2^i}) + dn^2\sum_{j=0}^{i-1}\left(\frac{7}{4}\right)^j = 7^{\log n}T(1) + dn^2\sum_{j=0}^{\lg n-1}\left(\frac{7}{4}\right)^j = \\ &= cn^{\log 7} + dn^2\frac{(\frac{7}{4})^{\log n} - 1}{\frac{7}{4} - 1} = cn^{\log 7} + \frac{3}{4}dn^2(\frac{n^{\log 7}}{n^2} - 1) \\ &= O(n^{\log 7}) \approx O(n^{2,81}). \end{aligned}$$

Như vậy thuật toán này hiệu quả hơn rất nhiều so với thuật toán nhân ma trận bình thường.

Định lý 4.1. *Thuật toán Strassen cho phép nhân nhanh ma trận, số những phép nhân các số, số các phép cộng các số và số các phép trừ các số cần thiết để nhân hai ma trận cỡ $n \times n$ với nhau có độ phức tạp là $O(n^{2,81})$.*

4.2.3. Những thuật toán dựa trên phương pháp Strassen

Như vậy việc đặt và kết hợp các ma trận bằng phép cộng ma trận, nhân ma trận của Strassen có duy nhất không? Câu trả lời là không, mà có rất nhiều cách đặt khác nữa như:

$$M_1 := (C+D-A)(H-F+E), \qquad M_5 := (C+D)(F-E),$$
$$M_2 := AE, \qquad M_6 := (B-C+A-D)H,$$
$$M_3 := BG, \qquad M_7 := D(E+H-F-G).$$
$$M_4 := (A-C)(H-F),$$
$$I = M_1 + M_3, \qquad J = M_1 + M_2 + M_5 + M_6,$$
$$K = M_1 + M_2 + M_4 - M_7, \quad L = M_1 + M_2 + M_4 + M_5.$$

Bạn đọc tự kiểm tra như đối với thuật toán Strassen, có thể tham khảo trong tài liệu [1]. Cùng một kết quả với thuật toán Strassen còn có cách làm sau đây, bạn đọc có thể tham khảo ở [2].

$$M_1 := (B-D)(G+H), \qquad M_5 := A(F-H),$$
$$M_2 := (A+D)(E+H), \qquad M_6 := D(G-E),$$
$$M_3 := (A-C)(E+F), \qquad M_7 := (C+D)A.$$
$$M_4 := (A+B)H,$$
$$I = M_1 + M_2 - M_4 + M_6, \quad J = M_4 + M_5,$$
$$K = M_6 + M_7, \qquad L = M_2 - M_3 + M_5 - M_7.$$

Sau công bố thuật toán Strasen đã có hàng loạt những thử nghiệm thành công cũng như thất bại cho việc làm thuật toán chạy tốt hơn. Câu hỏi đặt ra là chỗ nào là chỗ yếu và khắc phục nó như thế nào trong thuật toán Strassen? Một vấn đề dễ thấy là khi thêm các hàng và các cột vào ma trận thì thuật toán bị chậm đi 7 lần. Vấn đề thứ hai cũng được quan tâm ngay từ đầu là với $n = 2$ thực tế thuật toán Strassen tồi đi chứ không phải tốt hơn thuật toán khác. Vậy những điều này có đứng vững trong trường hợp $n = 2, 3, \ldots$? Vậy thì giá trị nào của n để ta dùng thuật toán Strassen và thuật toán theo định nghĩa nhân ma trận là tốt?

1. Một khả năng giải quyết vấn đề thứ nhất (khi n không là lũy thừa của 2) là đặt thêm vào ma trận những hàng và cột những giá trị 0 đến khi nào có thể được, nghĩa là nếu đầu tiên n là chẵn thì ta có thể thực

hiện bên trong thuật toán một lần. Nếu $\frac{n}{2}$ là số chẵn, ta lại có thể thực hiện thuật toán thêm một lần, còn nếu nó là số lẻ thì cần làm thêm vào các hàng và cột, đến giai đoạn này ta mới thêm vào, và hiển nhiên tiết kiệm được rất nhiều những số 0 thêm vào. Kĩ thuật này đã biết đến như *thêm phần tử động*.

Tuy phương pháp thêm vào phần tử động có làm tốt hơn rất nhiều thuật toán Strassen, nhưng nó chưa phải là phương pháp tối ưu. Huss và Lederman chỉ ra rằng phương pháp thêm phần tử động là kỹ thuật tối ưu cho những cỡ ma trận lẻ. Ý tưởng trong trường hợp số chiều lẻ của ma trận là ta tách ra tương ứng một hàng và một cột. Ma trận vệt có số chiều chẵn n và với nó có thể áp dụng bước hồi quy của Strassen.

$$\left[\begin{array}{c|c} Y_{11} & y_{12} \\ \hline y_{21} & y_{22} \end{array}\right] \left[\begin{array}{c|c} Z_{11} & z_{12} \\ \hline z_{21} & z_{22} \end{array}\right]$$

Kết quả được kết hợp lại thành ma trận X_{11}:

$$\left[\begin{array}{c|c} X_{11} & x_{12} \\ \hline x_{21} & x_{22} \end{array}\right] = \left[\begin{array}{c|c} Y_{11}Z_{11} + y_{12}z_{21} & Y_{11}z_{12} + y_{21}z_{22} \\ \hline y_{21}Z_{11} + y_{22}z_{21} & y_{21}z_{12} + y_{22}z_{22} \end{array}\right]$$

2. Vấn đề thứ hai không thể thực hiện trực tiếp được, vì đòi hỏi so sánh những tốc độ của thuật toán cơ sở với thuật toán Strassen với cố định n. Điều này đòi hỏi thời gian đối với các thao tác trên phần tử của ma trận trong cả hai phương pháp đều phụ thuộc vào phần cứng máy tính. Nghiên cứu từ năm 1996 của Luoma và Pauca chỉ ra rằng dung lượng của bộ nhớ cash có ý nghĩa rất lớn và trong nhiều trường hợp thực tế thuật toán Strassen thực hiện tồi hơn thuật toán cơ sở. Các tác giả đã đề nghị một ý tưởng triển khai thuật toán, việc thực hiện những tính toán trung gian ngay trước khi cần thiết và được lưu trong bộ nhớ cash để dùng chúng cho những bước sau.

Một phương pháp làm tốt hơn thuật toán của Strassen là của Winograd. Ông đã giảm được số lượng các phép tính cộng/ phép trừ từ 18 xuống 15, đồng thời giữ nguyên được số lượng phép nhân. Sơ đồ hệ

thống của phương pháp này có dạng:

$$
\begin{aligned}
S_1 &:= C + D, & P_1 &:= AE, & U_1 &:= P_1 + P_2,\\
S_2 &:= S_1 - A, & P_2 &:= BG, & U_2 &:= P_1 + P_4,\\
S_3 &:= A - C, & P_3 &:= S_1T_1, & U_3 &:= U_2 + P_5,\\
S_4 &:= B - S_2, & P_4 &:= S_2T_2, & U_4 &:= U_3 + P_7,\\
T_1 &:= F - E, & P_5 &:= S_3T_3, & U_5 &:= U_3 + P_3,\\
T_2 &:= H - T_1, & P_6 &:= T_4H, & U_6 &:= U_2 + P_3,\\
T_3 &:= H - F, & P_7 &:= DT_4, & U_7 &:= U_6 + P_6.\\
T_4 &:= G - T_2,
\end{aligned}
$$

Giá trị của ma trận tích là

$$I := U_1, \quad J := U_7, \quad K := U_4, \quad L := U_5.$$

Bạn đọc có thể kiểm tra lại tính đúng của các công thức trên và phân tích độ phức tạp của thuật toán này.

Người ta đã cố gắng tìm cách giảm số lượng phép nhân và phép cộng trong sơ đồ nguyên thủy của Strassen với $n = 2$, nhưng tất cả đều thất bại. Đến năm 1971 Hopcoft và Kerr đã chứng minh được rằng điều đó thực sự không thể được nếu không dùng sự giao hoán của phép nhân (ta biết rằng phép nhân hai ma trận không phải là giao hoán). Sau đó người người ta phát hiện ra một phương pháp nhân nhanh cho ma trận 3×3 với nhiều nhất là 21 phép nhân, mà nó cho một thuật toán có độ phức tạp $O(n^{\log_3 21})$. Sau đó hàng loạt phát minh những cách làm thuật toán tốt hơn với cỡ ma trận càng lớn hơn, nhưng càng ít khả năng đưa vào ứng dụng thực tế. Ví dụ Pan-đa phát minh ra phương pháp nhân ma trận cỡ 70×70 với 143640 phép nhân cơ sở (đáng lẽ 343000 phép toán nhân theo thuật toán cổ điển). Muộn hơn nữa phát minh ra những phương pháp hiệu quả hơn trên lí thuyết và vẫn chưa đưa vào được ứng dụng những thuật toán này : Thuật toán có độ phức tạp $O(n^{2,521813})$ tìm ra năm 1979; Thuật toán có độ phức tạp $O(n^{2,521801})$ tìm ra năm 1980; Thuật toán có độ phức tạp $O(n^{2,376})$ tìm ra năm 1986 (do Coppersmith và Winograd).

4.3. NHÂN NHỮNG SỐ NGUYÊN LỚN

Ta biết rằng máy tính gán một số lượng cố định các bit cho việc lưu giữ biến nguyên. Các phép toán số học như phép cộng và phép nhân của số nguyên thường được thực hiện bằng cách chuyển số hạng nguyên trên thanh ghi có độ dài cố định và khi đó gọi các phép toán số học được xây dựng trong phần cứng. Tuy nhiên có những ứng dụng quan trọng như trong trường hợp ứng dụng mã hóa, khi số chữ số rất lớn được xử lí trực tiếp bằng phần cứng. Trong trường hợp như vậy, cần thiết thực hiện những những phép toán này bằng cách lưu trữ những số nguyên dùng những cấu trúc dữ liệu thích hợp (như là mảng hoặc danh sách liên kết) và khi đó viết những thuật toán cho việc thực hiện những phép toán số học này. Khi ta phân tích độ phức tạp của những thuật toán như vậy cỡ n của số nguyên y được lấy là số chữ số của y. Bất kỳ một cơ số nào $b \geq 2$ có thể chọn để biểu diễn một số nguyên. Ta ký hiệu chữ số khác không thứ i của y bởi y_i, $i = 0, 1, \ldots, n-1$. Nghĩa là

$$y = \sum_{i=0}^{n-1} y_i b^i.$$

Trong chương trình phổ thông thuật toán nhân hai số nguyên y và z cỡ n chữ số, dễ thấy phải làm n^2 phép nhân các chữ số. Phép nhân số nguyên trong máy tính cũng tương tự như vậy với cơ số 2. Cho y và z là hai số nguyên n bit, thì phép cộng thực hiện $O(n)$ phép toán trên bit. Thuật toán nhân hai số nguyên thực hiện $O(n^2)$ phép toán trên bit.

4.3.1. Phương pháp nhân nhanh hai số nguyên

Ta sẽ áp dụng phương pháp chia để trị cho thiết kế thuật toán hiệu quả hơn thuật toán thông thường về nhân hai số nguyên. Giả sử n là một số lũy thừa của 2 (phần cuối ta chỉ ra cách khắc phục thế nào khi điều giả thiết này không xảy ra). Trước tiên ta chia mỗi số thành hai khối mỗi khối $\frac{n}{2}$ bit. Sau đó phép nhân hai số nguyên n bit được thu về ba phép nhân có số nguyên $\frac{n}{2}$ cộng với phép dịch chuyển và các phép cộng. Giả sử y được chia ra thành hai phần $\{a, b\}$ và z được chia

ra thành hai phần $\{c,d\}$. Ta viết các số này dưới dạng cơ số hai thì

$$y = a2^{\frac{n}{2}} + b$$
$$z = c2^{\frac{n}{2}} + d$$

Từ đây ta có phép nhân

$$\begin{aligned} yz &= (a2^{\frac{n}{2}} + b)(c2^{\frac{n}{2}} + d) \\ &= ac2^n + (ad + bc)2^{\frac{n}{2}} + bd. \end{aligned}$$

Theo phép nhân này thì yz được tính bằng bốn phép nhân của những số $\frac{n}{2}$ bit và hai phép cộng những số có nhiều nhất $2n$ chữ số và hai phép chuyển dịch qua trái $\frac{n}{2}$ vị trí (vì nhân với 2^n và $2^{\frac{n}{2}}$). Nếu tính tích ac, ad, bc và bd ta dùng cùng công thức hồi quy, khi quá trình này tiếp tục đạt đến trường hợp đơn giản chỉ còn hai chữ số, ta nhận được phụ thuộc hồi quy như sau:

$$T(n) = \begin{cases} c & \text{nếu } n = 1 \\ 4T(\frac{n}{2}) + dn & \text{nếu } n > 1 \text{ và } n \text{ là lũy thừa của 2.} \end{cases}$$

Đây là phương trình hồi quy, theo định lí 1.3 ở chương trước cho ta kết quả $T(n) = O(n^2)$. Bằng cách như vậy thì thuật toán không khá hơn thuật toán nhân thông thường. Như vậy áp dụng phương pháp chia để trị thông thường không dẫn tới kết quả tốt hơn, mà cần thiết phải có một chút thông minh ở đây. Ta nhớ lại thuật toán Strassen, mà nó đưa đến kết quả tốt là do giảm số lượng phép nhân. Ta xét xem có thể áp dụng nguyên lí đó ở đây được không. Nếu ta xem xét kỹ trong công thức sẽ thấy rằng thực chất không cần thiết tính tích ad và bc: Ta chỉ cần tổng của chúng là đủ. Ta xét công thức

$$\begin{aligned} yz &= 2^n ac + 2^{\frac{n}{2}}[(a-b)(d-c) + ac + bd] + bd \\ &= (2^n + 2^{\frac{n}{2}})ac + 2^{\frac{n}{2}}(a-b)(d-c) + (2^{\frac{n}{2}} + 1)bd. \end{aligned}$$

Đẳng thức này làm ta ngạc nhiên, nhưng nó đúng bằng công thức ta đã xét trên. Chỉ có khác là nó chỉ đòi hỏi ba phép nhân, sáu phép cộng (kể cả hai phép trừ) và hai phép dịch chuyển. Khi đó $T(n)$ là tổng các các

phép toán nhị phân cần thiết để nhân hai số nguyên n bit, thì

$$T(n) = \begin{cases} c & \text{nếu } n = 1 \\ 3T(\frac{n}{2}) + dn & \text{nếu } n > 1 \text{ và } n \text{ là bội của } 2, \end{cases}$$

ở đây c, d là những hằng số.

Do đó, theo định lí 1.3 ở chương trước thuật toán nhân nhanh có độ phức tạp

$$T(n) = O(n^{\log 3}) = O(n^{1.59}).$$

Vậy thuật toán này tốt hơn cách nhân hai số nguyên theo cách truyền thống rất nhiều. Cũng kết quả như vậy khi ta dùng công thức

$$yz = 2^n ac + 2^{\frac{n}{2}}[(a+b)(d+c) - ac - bd] + bd.$$

4.3.2. Thuật toán nhân hai số nguyên

Thiết lập chương trình ví dụ mô tả nhân nhanh hai số nguyên rất khó. Phần này ta chỉ mô tả thuật toán bằng giả mã. Chú ý là cách phân tích phần trên cũng thể áp dụng cho các số nguyên có số chữ số ở cơ số 10. Bạn đọc sẽ thấy thuật toán 4.5 (trang 110) là rất chung cho phép nhân các số nguyên ở mọi cơ số. Nếu số nguyên chỉ có một chữ số $n = 1$ ta nhân trực tiếp hai số này.

Thuật toán trên đã dùng hàm lấy dấu *sign* của một số, hàm *abs* lấy giá trị tuyệt đối cũng như hàm << dịch chuyển chữ số.

Ta có thể dùng phương án thứ hai của công thức nhân nhanh ở trên bằng cách thay dòng thứ 12 và dòng thứ 14 trong thuật toán bằng:

$m_2 := mult(a+b, d+c, \frac{n}{2});$

return$(s * (m_1 << n + (m_2 - m_1 - m_3) << \frac{n}{2} + m_3));$

Trên kia ta đã giả thiết n là bội của 2. Nếu n không phải là bội của 2 thì làm thế nào? Nếu n là một số chẵn, thì bước đầu tiên không có vấn đề gì và ta có thể chia hai số ra thành hai phần có độ dài bằng nhau. Nhưng tiếp tục quá trình đến một lúc nào đó sẽ nhận được độ dài có số lẻ chữ số và việc thực hiện như thuật toán không thể thực hiện được. Một cách khắc phục tốt nhất là thêm vào bên trái một số 0 ở đầu. Tuy rằng đưa vào một chữ số 0 sẽ làm kém hiệu quả của thuật toán, nhưng người ta phân tích thấy rằng độ phức tạp của thuật toán vẫn là $O(n^{\log 3})$.

Thuật toán 4.5 Nhân hai số nguyên lớn (*mult*)

function $mult(x, y, n)$
Đầu vào: Hai số nguyên x và y cỡ n.
Đầu ra: xy là tích của hai ma trận đầu vào.

```
if n = 1 then
    return(xy)
else
    s := sign(x) * sign(y);
    x := abs(x);
    y := abs(y);
    a := n/2 số chữ số bên trái của x;
    b := n/2 số chữ số bên phải của x;
    c := n/2 số chữ số bên trái của y;
    d := n/2 số chữ số bên phải của y;
    m1 := mult(a, c, n/2);
    m2 := mult(a - b, d - c, n/2);
    m3 := mult(b, d, n/2);
    return(s * (m1 << n + (m1 + m2 + m3) << n/2 + m3));
end if
end mult
```

Một điểm chú ý khác khi phân tích thuật toán ta đã bỏ qua tổng của hai số $\frac{n}{2}$ chữ số không phải lúc nào cũng cho một số n chữ số, mà có khi cho $n+1$ chữ số. Khi đó tích $(a+b)(c+d)$ trong công thức phương án hai có thể là số có $n+1$ chữ số (những tích còn lại ac và bd không có hiện tượng này). Ta lấy ví dụ trong cơ số 10: $x = 9999$ và $y = 9998$. Ta có $a = 99, b = 99, c = 99$ và $d = 98$. Khi đó $ac = 99.99 = 9801$, $bd = 99.98 = 9702$ và $(a+b)(c+d) = 198.197 = 39006$.

Rất may mắn là với việc tăng nhiều nhất một chữ số, độ phức tạp cuối cùng của thuật toán trong cả trường hợp này vẫn là $O(n^{\log 3})$. Bạn đọc có thể thấy chi tiết trong [1].

Khi hai số có độ dài chữ số khác nhau lần lượt là m và n với $n \neq m$ thì

ta thực hiện như thế nào trong trường hợp này. Để tiện lợi ta cho trước $m < n$. Nếu ta thêm vào phía trước số có chữ số nhỏ hơn là những số 0 ta nhận được thuật toán với độ phức tạp $O(n^{\log 3})$, đáng lẽ là $O(mn)$ với thuật toán nhân cơ bản. Từ đây ta thấy rằng với $m < n^{\log 3 - 1}$ thuật toán nhân cơ bản tốt hơn thuật toán nhân ta vừa thiết kế. Một biện pháp giải quyết tốt hơn là ta chia những chữ số thành các nhóm các chữ số có độ dài m. Mỗi khối này nhân với số ban đầu bằng cách sử dụng thuật toán tốt nhất, sau đó với $\left\lceil \frac{n}{m} \right\rceil$ phép cộng thêm và chuyển dịch sẽ nhận được kết quả cuối cùng. Độ phức tạp chung của thuật toán sửa đổi là $O(mn^{(\log 3)-1})$.

Câu hỏi về vấn đề lôgic: Thuật toán thiết kế ở trên tốt như vậy sao không đưa vào thực hiện trong chương trình phổ thông ? Có hai nguyên nhân :

1. Thuật toán rất khó giải thích và thật sự khó khi sử dụng, nhất là với những số nguyên lớn. Khái niệm hồi quy là rất trừu tượng và rất khó nắm bắt.

2. Ta có thể chờ đợi thuật toán tốt hơn thuật toán nhân cơ bản với những số có ít nhất 500 chữ số trở lên. Hỏi có ai trong trường phổ thông lại thực hiện các phép tính với các số dài như thế ?

4.4. TÍNH TOÁN KÍ HIỆU TRÊN NHỮNG ĐA THỨC

Phép toán đại số trên những đa thức là công cụ trong rất nhiều ứng dụng và do đó cần thiết kế những thuật toán hiệu quả để thực hiện những phép toán cơ sở trên đa thức như phép cộng và phép nhân với những hiệu quả cao có thể được. Ta thấy bài toán nhân các số lớn, với những số này biểu diễn theo một cơ số nào đó đều là đa thức và phép toán nhân các số lớn đưa về bài toán các phép toán của đa thức có hiệu quả nhất.

Ngày nay trong khoa học xuất hiện một ngành tính toán kí hiệu, nghĩa là đầu vào gồm các kí hiệu và đầu ra cũng là những kí hiệu chứ không phải là một số hoặc một giá trị số. Ta cho đa thức $P(x) = a_{m-1}x^{m-1} + \cdots + a_1x + a_0$, đây là biểu diễn kí hiệu của đa thức

$P(x)$. Khác với biểu diễn số của $P(x)$ là một hàm đơn giản ánh xạ đầu vào gồm điểm x thành đầu ra một điểm $P(x)$. Như vậy hai đa thức $P(x)$ và $Q(x)$ thì tích giá trị là một hàm $Ptmult(P,Q)$ mà nó ánh xạ một điểm x thành điểm đầu ra $P(x) \times Q(x)$.

Biểu diễn kí hiệu của đa thức $P(x) = a_{m-1}x^{m-1} + \cdots + a_1x + a_0$ là mảng những hệ số của nó $[a_0, a_1, ..., a_{m-1}]$. Như vậy hai đa thức $P(x) = a_{m-1}x^{m-1} + \cdots + a_1x + a_0$ và $Q(x) = b_{n-1}x^{n-1} + \cdots + b_1x + b_0$, thì tích kí hiệu của hai đa thức $P(x)Q(x)$ là mảng hệ số $[c_0, c_1, ..., c_{m+n-2}]$, được định nghĩa bởi

$$c_k = \sum_{i+j=k} a_i b_j, \quad 0 \le i \le m-1, 0 \le j \le n-1, k = 0, ..., m+n-2. \qquad (4.1)$$

Ví dụ, đa thức $P(x) = 3x^2 + 2x - 5$ có mảng hệ số $[-5, 2, 3]$ và $Q(x) = x^3 - x + 4$ có mảng hệ số $[4, -1, 0, 1]$, tích kí hiệu của chúng là một đa thức $3x^5 + 2x^4 - 8x^3 + 10x^2 + 13x - 20$ có hệ số mảng $[-20, 13, 10, -8, 2, 3]$.

Biểu diễn kí hiệu của tích rất quan trọng trong thực tế cho việc xác định những tính chất khác nhau của tích các đa thức. Các hàm tính giá trị không cho ta cách tính đạo hàm của tích hai đa thức, khi đó biểu diễn kí hiệu dễ dàng thực hiện được phép tính này. Ngoài ra còn nhiều phép tính trên đa thức khi dùng kí hiệu sẽ dễ dàng hơn nhiều như tìm nghiệm, lấy tích phân, ... Phần sau đây ta sẽ thảo luận các phép toán thực hiện trên đa thức bằng các kí hiệu.

4.4.1. Nhân hai đa thức cùng bậc

Ta kí hiệu thuật toán nhân trực tiếp theo công thức (4.1) là *DPolymult* có độ phức tạp $O(mn)$, ở đây ta chọn phép nhân các hệ số là phép toán cơ sở. Bây giờ ta thiết kế một thuật toán hiệu quả hơn dựa trên cơ sở chia để trị. Tiết này ta giả thiết rằng $m = n$ cho hai đa thức đã cho ở phần trước. Đặt $d = \left\lceil \frac{n}{2} \right\rceil$, bước đầu tiên là ta chia tập các hệ số của đa thức thành hai phần: Một tập gồm các hệ số từ bậc cao nhất trở xuống đến d : $a_{n-1}, a_{n-2}, ..., a_d$ và một tập từ chỉ số $d-1$ trở xuống $a_{d-1}, a_{d-2}, ..., a_0$.

Khi đó ta đặt

$$P_1(x) = a_{d-1}x^{d-1} + a_{d-2}x^{d-2} + \cdots + a_1x + a_0,$$
$$P_2(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_{d+1}x + a_d.$$

Ta nhận được

$$P(x) = x^d P_2(x) + P_1(x).$$

Tương tự như vậy ta cũng chia tập các hệ số của đa thức $Q(x)$ thành hai đa thức $Q_1(x)$ và $Q_2(x)$ với cỡ đầu vào là d sao cho

$$Q(x) = x^d Q_2(x) + Q_1(x).$$

Ta nhân hai đa thức $P(x)$ và $Q(x)$ trực tiếp và biến đổi theo luật phân bố cho công thức

$$P(x)Q(x) = x^{2d}P_2(x)Q_2(x) + x^d(P_1(x)Q_2(x) + P_2(x)Q_1(x)) + P_1(x)Q_1(x). \tag{4.2}$$

Ta chú ý rằng đa thức $P_1(x)$ và $Q_1(x)$ có cỡ đầu vào là d và những đa thức $P_2(x)$ và $Q_2(x)$ có cỡ đầu vào là d hoặc $d-1$. Trong trường hợp $P_2(x)$ và $Q_2(x)$ có cỡ đầu vào là $d-1$ thì ta có thể thêm hệ số 0 vào mảng hệ số để cho đủ d. Như vậy bài toán nhân hai đa thức $P(x)$ và $Q(x)$ đưa về bài toán bốn tích hai đa thức có cỡ đầu vào là $d = \left\lceil \frac{n}{2} \right\rceil$ với hai phép nhân lũy thừa của x và ba phép cộng nữa. Ta thấy rằng hàm tính toán của thuật toán dựa trên công thức (4.2) là

$$T(n) = \begin{cases} c & \text{nếu } n = 1 \\ 4T(\left\lceil \frac{n}{2} \right\rceil) + dn & \text{nếu } n > 1. \end{cases}$$

với c, d là những hệ số hằng. Như các mục trước và định lí cơ bản hàm hồi quy cho ta độ phức tạp của thuật toán là $T(n) = O(n^2)$. Như vậy với phương pháp chia để trị bình thường thì độ phức tạp của thuật toán không hơn gì thuật toán nhân hai đa thức cơ bản. Ta vận dụng cách chia thông minh hơn một chút dựa trên cơ sở công thức sau đây :

$$P(x)Q(x) = x^{2d}P_2(x)Q_2(x) + x^d((P_1(x) + P_2(x))(Q_1(x) + Q_2(x)) - \\ - P_1(x)Q_1(x) - P_2(x)Q_2(x)) + P_1(x)Q_1(x). \tag{4.3}$$

Từ công thức này ta thiết lập thuật toán chia để trị *polymult* cho phép nhân đa thức. *polymult* chứa thủ tục *split*$(P(x), P_1(x), P_2(x))$ có đầu vào

là $P(x)$ và đầu ra là hai đa thức $P_1(x)$ và $P_2(x)$. Trong thuật toán có lệnh hồi quy phép nhân hai đa thức có cỡ nhỏ hơn.

Thuật toán 4.6 Nhân hai đa thức cùng bậc (*polymult*)

function *polymult*(P, Q, n)
Đầu vào: Số nguyên n và đa thức
$P(x) = a_{n-1}x^{n-1} + \cdots + a_1x + a_0$,
$Q(x) = b_{n-1}x^{n-1} + \cdots + b_1x + b_0$.
Đầu ra: Tích $P(x)Q(x)$.
1: **if** $n = 1$ **then**
2: **return**(a_0b_0)
3: **else**
4: $d := \left\lceil \frac{n}{2} \right\rceil$;
5: *split*$(P(x), P_1(x), P_2(x))$;
6: *split*$(Q(x), Q_1(x), Q_2(x))$;
7: $R(x) :=$ *polymult*$(P_2(x), Q_2(x), d)$;
8: $S(x) :=$ *polymult*$(P_1(x) + P_2(x), Q_1(x) + Q_2(x), d)$;
9: $T(x) :=$ *polymult*$(P_1(x), Q_1(x), d)$;
10: **return**$(x^{2d}R(x) + x^d(S(x) - R(x) - T(x)) + T(x))$;
11: **end if**
end *polymult*

Trong thuật toán 4.6 đáng lẽ ra phải thiết kế thủ tục nhân một lũy thừa của x với một đa thức như $x^iP(x)$. Nhưng để cho ngắn gọn theo cách viết này là gọi thủ tục đó rồi. Thuật toán cũng không thể hiện tường minh các hệ số của đa thức được triển khai thế nào như danh sách, mảng, ... vấn đề này dành cho các bạn lập trình cụ thể.

Ta phân tích thuật toán *polymult*, chọn phép nhân những hệ số là phép toán cơ sở và như vậy ta bỏ qua hai phép nhân với lũy thừa của x. Tuy nhiên phép toán nhân một đa thức với x^i có độ phức tạp tuyến tính và không ảnh hưởng đến độ phức tạp của thuật toán *polymult*. Cũng như các bài toán trước ta giả sử n là lũy thừa của 2. Vì thuật toán gọi ba lần đến chính nó với n thay bằng $d = \frac{n}{2}$, kí hiệu số phép nhân hệ số $T(n)$ được thực hiện bởi *polymult*, ta có quan hệ truy hồi sau đây:

$$T(n) = \begin{cases} c & \text{nếu } n = 1 \\ 3T(\frac{n}{2}) + dn & \text{nếu } n > 1 \text{ và } n \text{ là bội của } 2. \end{cases}$$

ở đây c, d là những hằng số.

Theo định lí truy hồi cơ bản ta suy ra $T(n) = O(n^{\log 3})$. Vì $\log 3 \approx 1,59 < 2$, nên ta thấy thuật toán này chạy tốt hơn thuật toán nhân cơ sở *DPolymult*.

4.4.2. Nhân hai đa thức khác bậc

Trong thực tế thường phải nhân hai đa thức $P(x)$ và $Q(x)$ có bậc khác nhau m và n tương ứng (nghĩa là có đầu vào khác nhau). Nếu $m < n$, thì ta có thể thêm vào phía trước $P(x)$ những $n - m$ số hạng có hệ số bằng 0. Như vậy sẽ không hiệu quả khi n rất lớn so với m. Khi đó tốt nhất là chia $Q(x)$ thành các khối có cỡ đầu vào m. Để cho thuận tiện ta giả sử n là bội của m, nghĩa là $n = km$ với một số nguyên dương k nào đó. Ta đặt $Q_i(x)$ là đa thức bậc m được cho bởi công thức

$$Q_i(x) = b_{im-1}x^{m-1} + b_{im-2}x^{m-2} + \cdots + b_{(i-1)m+1}x + b_{(i-1)m}, \quad i \in \{1, 2, ..., k\}.$$

Dễ thấy

$$Q(x) = Q_k(x)x^{m(k-1)} + Q_{k-1}(x)x^{m(k-2)} + \cdots + Q_2(x)x^m + Q_1(x).$$

Từ đó suy ra theo luật phân phối

$$P(x)Q(x) = P(x)Q_k(x)x^{m(k-1)} + P(x)Q_{k-1}(x)x^{m(k-2)} + \cdots + P(x)Q_1(x).$$

Sử dụng ý tưởng trên ta tạo ra thủ tục *polymult*1 cho việc nhân hai đa thức $P(x)$ và $Q(x)$ (thuật toán 4.7). Thủ tục *polymult*1 vẫn có hiệu quả khi bậc $m - 1$ của $P(x)$ thực sự nhỏ hơn $n - 1$ của $Q(x)$. Nếu n không phải là bội của m thì ta tính số k lớn nhất sao cho $n > km$ và $P(x)$ thêm vào các số hạng phía trước có hệ số 0 là $n - km$ thừa số.

Độ phức tạp của *polymult*1 nhân hai đa thức bậc $m - 1$ là $O(m^{\log 3})$. Vì *polymult*1 có gọi *polymult* tổng cộng $k = \frac{n}{m}$ lần, mỗi lần khi đầu vào là đa thức bậc $m - 1$. Do đó suy ra độ phức tạp tính toán của *polymult*1 là

$$O(km^{\log 3}) = O\left(\frac{nm^{\log 3}}{m}\right) = O\left(nm^{\log \frac{3}{2}}\right).$$

Thuật toán 4.7 Nhân hai đa thức khác bậc (*polymult*1)

function *polymult*1(P, Q, m, n)
Đầu vào: m, n là nguyên dương{ $n = km$ với k nguyên nào đó }
$P(x) = a_{m-1}x^{m-1} + \cdots + a_1x + a_0$,
$Q(x) = b_{n-1}x^{n-1} + \cdots + b_1x + b_0$.
Đầu ra: Tích $P(x)Q(x)$.
1: $prodpoly(x) := 0$; {đặt các hệ số của $prodpoly(x)$ bằng 0}
2: **for** $n := 1$ **to** k **do**
3: $Q_i(x) := b_{im-1}x^{m-1} + b_{im-2}x^{m-2} + \cdots + b_{(i-1)m+1}x + b_{(i-1)m}$
4: **end for**
5: **for** $n := 1$ **to** k **do**
6: $prodpoly(x) := prodpoly(x) + x^{(i-1)m}\, polymult(P(x), Q_i(x), m)$
7: **end for**
8: **return**$(prodpoly(x))$
end *polymult*1

4.5. CHỌN PHẦN TỬ NHỎ BẤT KÌ

Cho một danh sách các số $L[1..n]$, ta xét bài toán chọn phần tử nhỏ thứ k nằm trong L. Khi $k = 1$ hoặc $k = n$, bài toán này trùng với bài toán tìm phần tử cực tiểu hoặc cực đại trong L, trong trường hợp này có thuật toán với độ phức tạp tuyến tính. Trong trường hợp tổng quát bài toán tìm phần tử nhỏ thứ k có thể giải bằng thuật toán sắp xếp ví dụ như *mergesort*, những thuật toán này có độ phức tạp $O(n \log n)$. Việc tối ưu hóa thuật toán đơn giản này cũng không phải đơn giản, trong phần này ta đưa ra một thuật toán có độ phức tạp tuyến tính nhờ vào thủ tục *sắp đặt tại chỗ raplace*, mà nó được tạo ra dựa trên cơ sở *quicksort* cùng với chiến thuật tìm kiếm nhị phân.

4.5.1. Thuật toán lựa chọn

Để thuật tiện xây dựng thuật toán ta giả sử rằng $L[1..n]$ gồm những phần tử khác nhau. Giả sử khởi đầu ta gọi *raplace* với $L[1..n]$ và trả về giá trị $j = p$ sao cho trong mảng đã được sắp xếp lại mà $L[i] < L[j]$ với $1 \le i < j$ và $L[i] > L[j]$ với $j < i \le n$ và phần tử trục có giá trị bằng $L[1]$. Thực tế, phần tử trục là phần nhỏ thứ j trong L. Do đó, nếu $j = k$

thì phần tử nhỏ thứ k đã tìm được. Mặt khác, nếu $k < j$, thì mỗi phần tử trong $L[1..j-1]$ đều nhỏ hơn mỗi phần tử trong $L[j+1..n]$, thì phần tử nhỏ thứ k trong $L[1..n]$ trùng với phần tử nhỏ thứ k trong $L[1..j-1]$. Tương tự, nếu $k > j$, thì phần tử nhỏ thứ k trong $L[1..n]$ trùng với phần tử nhỏ thứ k trong $L[j+1..n]$. Do đó bằng cách gọi *raplace* ta có hoặc đã xác định được phần tử nhỏ thứ k hoặc một mảng con (có cỡ ít nhất nhỏ hơn 1 so với cỡ ban đầu) mà nó chứa phần tử nhỏ thứ k.

Khi $k \neq j$, ta lại gọi lại *raplace* thì hoặc là xác định được phần tử nhỏ thứ k hoặc là thu hẹp độ dài của mảng con có chứa phần tử nhỏ thứ k. Quá trình này tiếp tục sau $n-1$ lần gọi *raplace*, phần tử nhỏ thứ k sẽ được xác định. Thuật toán 4.8 hồi quy, thể hiện mô tả hình thức trên. Danh sách được đưa vào trong *select* là một mảng con $L[\ell..h]$ của mảng $L[1..n]$. Thủ tục *select* tính phần tử nhỏ nhất thứ $k-\ell+1$ trong mảng $L[\ell..h]$, khi khởi đầu chương trình ta gọi thủ tục với $\ell = 1$ và $h = n$.

Thuật toán 4.8 Chọn phần tử thứ k theo độ lớn (*select*)

```
procedure   select(L[ℓ..h], k, x)
  Đầu vào:  L[ℓ..h] mảng các phần tử đã cho
            k là một số nguyên sao cho ℓ ≤ k ≤ h.
  Đầu ra:   x là phần tử nhỏ thứ k − ℓ + 1 trong L[ℓ..h].
  raplace(L[ℓ..h], pos)
  if k=pos then
     x := L[pos];
  else
     if k<pos then
        select(L[ℓ..pos − 1], k, x);
     else
        select(L[pos + 1..h], k, x);
     end if
  end if
end   select
```

Ta lấy một ví dụ cụ thể để dễ thấy thuật toán thực hiện thế nào: Tìm phần tử nhỏ thứ $k = 5$.

1. Khởi động thuật toán: Cho một mảng với chỉ số và phần tử tương ứng như sau:

Chỉ số:	1	2	3	4	5	6	7
Phần tử:	22	9	23	52	15	19	47

2. Gọi thủ tục *raplace*($L[1..7], pos$): Thủ tục này lấy $L[1] = 22$ làm trục, mảng được sắp xếp lại và tìm trả về giá trị $pos = 4$.

Chỉ số:	1	2	3	4	5	6	7
Phần tử:	15	9	19	22	52	23	47

3. Gọi thủ tục *raplace*($L[5..7], pos$): Thủ tục này lấy $L[5] = 52$ làm trục, mảng được sắp xếp lại và tìm trả về giá trị $pos = 7$.

Chỉ số:	1	2	3	4	5	6	7
Phần tử:	15	9	19	22	47	23	52

4. Gọi thủ tục *raplace*($L[5..6], pos$): Thủ tục này lấy $L[5] = 47$ làm trục, mảng được sắp xếp lại và tìm trả về giá trị $pos = 6$.

Chỉ số:	1	2	3	4	5	6	7
Phần tử:	15	9	19	22	23	47	52

5. Gọi thủ tục *raplace*($L[5..5], pos$): Thủ tục này lấy $L[5] = 23$ làm trục, trả về giá trị $pos = 5$.

Chỉ số:	1	2	3	4	5	6	7
Phần tử:	15	9	19	22	23	47	52

Như vậy $x = 23$.

Ta phân tích thuật toán *select* với mảng đầu vào cỡ n, ta giả sử $\ell = 1$ và $h = n$. Ta sẽ dùng phép toán so sánh trong thủ tục *raplace* làm phép toán cơ bản để chúng ta so sánh. Kí hiệu $T(n,k)$ và $A(n,k)$ là độ phức tạp trong trường hợp xấu nhất và trường hợp trung bình của thuật toán *select* với sự hạn chế đầu vào thông số k cố định. Khi đó

$$T(n) = \max\{T(n,k) | 1 \le k \le n\}$$

và giả sử mỗi k đều có vai trò như nhau,

$$A(n) = \frac{\sum_{k=1}^{n} A(n,k)}{n}.$$

Vì thủ tục *raplace* có độ phức tạp $n+1$ với mảng đầu vào n (lấy phép so sánh làm phép toán cơ sở để tính), và vì trường hợp xấu nhất là mảng

chỉ thu hẹp một phần tử với mỗi lần gọi, nên ta có

$$T(n,k) = (n+1) + n + (n-1) + \cdots + 2 = \frac{(n+1)(n+2)}{2} - 1 \in \Theta(n^2).$$

Như vậy, thuật toán *select* có độ phức tạp tính toán theo đa thức bậc hai như là thuật toán nhanh nhất *quicksort*.

Khi phân tích độ phức tạp của *select* trong trường hợp trung bình ta thường giả sử rằng đầu vào của *select* là tất cả những hoán vị của $1,2,\ldots,n$ và mỗi hoán vị này là như nhau. Ta sẽ chứng minh bằng quy nạp toán học theo n mệnh đề sau đây:

$$A(n,k) \le 4n \text{ với mọi } n \text{ và } k \in \{1,2,\ldots,n\}. \tag{4.4}$$

Thật vậy, với $n = 1$ và $k = 1$ ta có $A(1,1) = 2$ mệnh đề (4.4) đúng.

Giả sử rằng (4.4) đúng với mọi cặp m,k với $k \in \{1,2,\ldots,m\}$ và $m < n$, ta phải chứng minh rằng (4.4) đúng cho n và tất cả $k \in \{1,2,\ldots,n\}$.

Lấy k là một số bất kỳ sao cho $1 \le k \le n$. Ta đã giả thiết rằng tất cả các hoán vị của $1,2,\ldots,n$ là như nhau cho đầu vào của *select*, nên kéo theo sau khi gọi *raplace* với $L[1..n]$, mỗi giá trị $1,2,\ldots,n$ như nhau tại giá trị đầu ra của thông số *pos*. Có ba trường hợp ta phải xét phụ thuộc vào *pos* nhỏ hơn, bằng hoặc lớn hơn k. Để cho gọn ta đặt $i = pos$.

a) Nếu $k = i$, thì *select* dừng sau một lần gọi *raplace*, như vậy sẽ có $n+1$ phép so sánh được thực hiện.

b) Nếu $k < i$, thì *select* gọi lại chính nó với mảng $L[1..i-1]$, như vậy trong trường hợp này có số các phép so sánh sau :

$$n + 1 + A(i-1,k) \le n + 1 + 4(i-1) \text{ (theo giả thiết quy nạp).}$$

c) Nếu $k > i$, thì *select* gọi lại chính nó với mảng $L[i+1..n]$, như vậy trong trường hợp này có số các phép so sánh sau :

$$n + 1 + A(n-i,k-i) \le n + 1 + 4(n-i) \text{ (theo giả thiết quy nạp).}$$

Ta tổng hợp cả ba trường hợp trên

$$A(n,k) = \frac{1}{n}\left[n + 1 + \sum_{i=1}^{k-1}(n+1+A(n-i,k-i)) + \sum_{i=k+1}^{n}(n+1+A(i-1,k))\right]$$

$$= \frac{1}{n}\left[n(n+1) + \sum_{i=1}^{k-1}A(n-i,k-i) + \sum_{i=k+1}^{n}A(i-1,k)\right]$$

$$\leq n+1+\frac{4}{n}\left[\sum_{i=1}^{k-1}(n-i)+\sum_{i=k+1}^{n}(i-1)\right].$$

Như một hàm của k thì vế phải của bất đẳng thức trên đạt cực đại tại $k=\lceil\frac{n+1}{2}\rceil$. Ta giả sử n là số lẻ (với n là số chẵn chứng minh hoàn toàn tương tự), khi đó bất đẳng thức cuối cùng cho ta

$$\begin{aligned}
A(n,k) \leq A\left(n,\frac{n+1}{2}\right) &\leq n+1+\frac{4}{n}\left[2\left(\sum_{i=1}^{n-1} i+\sum_{i=1}^{\frac{n-1}{2}} i\right)\right] \\
&= n+1+\frac{4}{n}\left[n(n-1)-\left(\frac{n-1}{2}\right)\left(\left(\frac{n-1}{2}\right)+1\right)\right] \\
&= n+1+\frac{4}{n}\left[n(n-1)-\frac{(n-1)(n+1)}{4}\right] \\
&= n+1+\frac{4}{n}\left[(n-1)\left(n-\frac{n+1}{4}\right)\right] \\
&= n+1+\frac{4}{n}\left[(n-1)\left(\frac{3n-1}{4}\right)\right] \\
&= n+1+\frac{(n-1)(3n-1)}{n} \leq 4n.
\end{aligned}$$

Như vậy đã chứng minh quy nạp đầy đủ.

Ta suy ra ngay $A(n) \leq 4n$. Vì *select* thực hiện ít nhất một lần gọi *raplace* cho mọi đầu vào, nên $A(n,k)$ và $A(n)$ có $n+1$ phép so sánh, do đó ta có

$$n+1 \leq A(n); A(n,k) \leq 4n, \text{ với mọi } n \text{ và } k \in \{1,2,\ldots,n\}.$$

Do vậy, $A(n)$ và $A(n,k)$ đều là $\Theta(n)$ (độ phức tạp tuyến tính).

4.5.2. Thuật toán chọn với trục điểm giữa

Thuật toán ở phần trên cho ta độ phức tạp trung bình tuyến tính, nhưng độ phức tạp trong trường hợp xấu nhất lại là đa thức bậc hai. Ta có thể dùng thuật toán *raplace* trong *select* một cách thông minh hơn để cho độ phức tạp trong trường hợp xấu nhất là tuyến tính được không? Ta thấy rằng, nguyên nhân độ phức tạp tính toán xấu nhất của thuật toán phần trước là độ phức tạp đa thức bậc hai, vì phần tử trục luôn luôn nằm tại điểm đầu của mảng sắp xếp. Mặt khác, nếu phần tử trục

luôn luôn rơi vào trung điểm của mảng, thì khi đó mảng được chia làm hai mảng con có cỡ gần bằng nhau và lúc bấy giờ số những phép so sánh sẽ phải thực hiện xấp xỉ như sau:

$$(n+1)\left(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots\right)=2(n+1)\in\Theta(n).$$

Như vậy, nếu ta biết trung điểm ở mỗi giai đoạn của *select*, ta sẽ dùng nó như phần tử trục trong khi gọi *raplace* để tính phần tử nhỏ thứ k theo thời gian độ phức tạp tuyến tính. Tuy nhiên vấn đề tính chính xác trung điểm là trường hợp đặc biệt $k=\frac{n+1}{2}$ phải xem xét (với n là số lẻ).

Ta xét một phương pháp *trung điểm của trung điểm* để tính xấp xỉ trung điểm trong quá trình tính toán. Quá trình xấp xỉ này được gọi là *giả trung điểm*, được định nghĩa như sau: Giả sử rằng n là một số lẻ bội của 5, như vậy $n=5q$ với số lẻ q nào đó. Ta chia mảng $L[1..n]$ thành q mảng con $L_1, L_2, ..., L_q$, mỗi mảng có cỡ là 5, nghĩa là

$$L_i = L[5(i-1)+1..5i], \quad i=1,2,...,q.$$

Bây giờ ta sắp xếp mỗi mảng $L_1, L_2, ..., L_q$ và khi đó sắp xếp lại chúng theo trung điểm tăng dần và nhận được danh sách $L'_1, L'_2, ..., L'_q$, như vậy

$$L'_i = [a_i, b_i, c_i, d_i, e_i], \quad a_i < b_i < c_i < d_i < e_i, \quad i=1,2,..,q,$$

ở đây $c_1 < c_2 < \cdots < c_q$. Chú ý rằng c_i là trung điểm của mảng L'_i, $i=1,2,...,q$, sao cho với hoán vị thích hợp π của $1,2,...,p$, $c_{\pi(i)}$ là trung điểm của $L_i, i=1,...,q$. *Giả trung điểm* kí hiệu là *pm*, là trung điểm $c_{\frac{q+1}{2}}$ của những trung điểm $c_1, c_2, ..., c_q$.

Ví dụ cụ thể sau đây thể hiện cách tìm giả trung điểm $pm=48$ trong mảng $L[1..35]$, nhưng trung điểm của dãy này lại là 43: Ta chia $L[1..35]$ thành $q=7$ mảng con cỡ 5.

$$\begin{array}{llll} L[1..35] = & \overset{L_1}{[[87,23,91,25,66]} & \overset{L_2}{[12,55,8,77,67]} & \overset{L_3}{[88,32,11,44,57]} \\ & \overset{L_4}{[75,21,6,89,80]} & \overset{L_5}{[43,84,34,61,48]} & \overset{L_6}{[14,10,37,27,29]} \\ & \overset{L_7}{[36,16,15,78,83]]} & & \end{array}$$

Sắp xếp lại các mảng con và xếp tăng dần theo trung điểm của chúng

	a_i	b_i	c_i	d_i	e_i	
$L'_1=$	[10	14	$\underline{27}$	29	37]	$=L_6$
$L'_2=$	[15	16	$\underline{36}$	78	83]	$=L_7$
$L'_3=$	[11	32	$\underline{44}$	57	88]	$=L_3$
$L'_4=$	[34	43	$\boxed{\underline{48}}$	61	84]	$=L_5$
$L'_5=$	[8	12	$\underline{55}$	67	77]	$=L_2$
$L'_6=$	[23	25	$\underline{66}$	87	91]	$=L_1$
$L'_7=$	[6	21	$\underline{75}$	80	89]	$=L_4$

Mệnh đề 4.3. *Số những phần tử của $L[1..n]$ lớn hơn (nhỏ hơn) giả trung điểm pm nhiều nhất là $\frac{7}{10}n-\frac{3}{2}$.*

Chứng minh. Theo định nghĩa *pm*, mọi phần tử trong $\frac{3(q+1)}{2}$ phần tử sau đây nhỏ hơn hoặc bằng *pm*:

$$a_1, b_1, c_1, a_2, b_2, c_2, \ldots, a_{\frac{q+1}{2}}, b_{\frac{q+1}{2}}, c_{\frac{q+1}{2}}$$

Như vậy số phần tử trong $L[1..n]$ lớn hơn *pm* nhiều nhất là

$$n-3\left(\frac{q+1}{2}\right)=n-3\left(\frac{\frac{n}{5}+1}{2}\right)=\frac{7}{10}n-\frac{3}{2}.$$

Tương tự ta cũng có nhiều nhất $\frac{7}{10}n-\frac{3}{2}$ phần tử nhỏ hơn *pm*. ☺

Mệnh đề 4.3 là cơ sở để ta thiết kế thuật toán *medianselect* dưới đây tìm phần tử nhỏ thứ k trong mảng $L[1..n]$. Thuật toán này có độ phức tạp trong trường hợp xấu nhất là $\Theta(n)$. Thuật toán này hoạt động như là thuật toán *select*, ngoại trừ ta tính thêm giả trung điểm và dùng nó như là phần tử trục khi gọi thủ tục *raplace*2. Thủ tục *raplace*2 là phiên bản của *raplace* sửa đổi sao cho thêm đối số đầu vào *phần tử trục* và sắp xếp mảng đối với phần tử trục này.

Trung điểm của 5 phần tử có thể tính được bằng 6 phép so sánh. Như vậy ta có thể tính các trung điểm $c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(q)}$ từ $L_1, L_2, \ldots, L_q$ bằng cách thực hiện $6q=\frac{6}{5}n$ phép so sánh. Giả trung điểm *pm* có thể tính bởi cách gọi hồi quy *medianselect* với đầu vào $c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(q)}$ và $k=\frac{q+1}{2}$.

Giả mã cho thuật toán *medianselect* có thủ tục *median*5 được gọi với

đối số đầu vào $L[\ell..h]$ và đầu ra $M[1..q]$, ở đây $M[i] = c_{\pi(i)}, i = 1,2,...,q$ Khi khởi đầu gọi *medianselect* với $\ell = 1$ và $h = n$.

Thuật toán 4.9 Chọn phần tử thứ k theo độ lớn (*medianselect*)

procedure *medianselect*$(L[\ell..h], k, x)$
Đầu vào: $L[\ell..h]$ mảng các phần tử đã cho
k là một số nguyên sao cho $\ell \le k \le h$.
Đầu ra: x là phần tử nhỏ thứ $k-\ell+1$ trong $L[\ell..h]$.

```
if h − ℓ + 1 <= 5 then
    {Dùng phương pháp đặc biệt để giải}
end if
q := (h − ℓ)/5;
median5(L[ℓ..h], M[1..q]);
medianselect(M[1..q], (q + 1)/2, pivot);{giả trung điểm là pivot}
raplace2(L[ℓ..h], pivot, pos)
if k=pos then
    x := L[pos];
else
    if k<pos then
        medianselect(L[ℓ..pos − 1], k, x);
    else
        medianselect(L[pos + 1..h], k, x);
    end if
end if
```

end *medianselect*

Ta phân tích thuật toán *medianselect* trong trường hợp xấu nhất: $T(n)$. Giả sử *medianselect* đầu vào $L[1..n]$ với những phần tử khác nhau. Khi thực hiện *median5* sinh ra nhiều nhất $\frac{5}{6}n$ phép so sánh. Thực hiện *raplace2* sinh ra $n+1$ phép so sánh. Thực hiện hồi quy *medianselect* với $M[1..q]$ sinh ra nhiều nhất $T(q) = T(\frac{n}{5})$ phép so sánh. Vì theo mệnh đề 4.3 mảng để dùng gọi hồi quy có cỡ nhiều nhất là $\frac{7}{10}n - \frac{3}{2}$, do đó thực hiện *medianselect* sinh ra nhiều nhất là $T(\frac{7}{10}n - \frac{3}{2})$ phép so sánh. Ta nhận được bất phương trình truy hồi

$$T(n) \le T\left(\frac{1}{5}n\right) + T\left(\frac{7}{10}n - \frac{3}{2}\right) + \frac{11}{5}n + 1,$$

với điều kiện ban đầu là $T(5) = 6$. Từ bất đẳng thức này ta có thể chứng minh trực tiếp bằng quy nạp theo n bất đẳng thức $T(n) \leq 22$ (bạn đọc tự kiểm tra). Như vậy thuật toán *medianselect* có độ phức tạp tuyến tính trong trường hợp xấu nhất.

4.6. MỘT ỨNG DỤNG CHO XẾP LỊCH THI ĐẤU THỂ THAO

Phương pháp chia để trị đã được ứng dụng từ lâu để lập bảng các giải thi đấu, mỗi một đối thủ đấu lần lượt các đối thủ khác để tính điểm. Trong phần này ta xét

Giải thi đấu bóng đá: Giải thi đấu bóng đá ngoại hạng Việt Nam được tổ chức với n đội bóng (được đánh số từ 1 đến n). Giải vô địch cần tiến hành trong n vòng đấu, nếu n là số lẻ và $n - 1$ vòng đấu nếu n là số chẵn. Mỗi cặp đội bóng được đấu với nhau đúng một lần.

Bài toán đặt ra: Hãy lập một chương trình máy tính mà nó đưa ra lịch trình thi đấu của giải vô địch với cấu trúc bảng.

Đầu vào: Nhập số lượng đội bóng n ($2 \leq n \leq 24$) được nhập vào theo cấu trúc chuẩn.

Đầu ra: Kết quả ra là một bảng những số nguyên gồm n dòng. Một số tại cột j và hàng i trong bảng có nghĩa là số của đội bóng, mà nó đấu với đội số i trong vòng đấu thứ j. (Nếu đội bóng i đấu với đội bóng k tại vòng thứ j, hiển nhiên nghĩa là đội bóng k đấu với đội bóng i cũng ở vòng này). Nếu đội i nghỉ tại vòng thứ j, khi đó số j tại hàng thứ i bằng 0.

Ví dụ: + Đầu vào: 3 hoặc 4

+ Đầu ra:

2	3	0
1	0	3
0	1	2

hoặc

2	3	4
1	4	3
4	1	2
3	2	2

Lời giải: Giải bài này ta thay đổi ý nghĩa một chút tại đầu ra. Nhận được bảng đầu ra đáng lẽ là như bài ra, nhưng ta quy định lại bảng số lúc nào kết xuất cũng là cỡ $n \times n$ (thậm chí cả khi n là số chẵn). Mỗi số tại dòng thứ i và cột thứ j là vòng đấu của hai đội bóng có số i và j. Từ

bảng số như vậy đưa về bảng số như trong đề bài đã ra, bạn đọc thử tính như một bài tập.

Như vậy bài toán không có ý nghĩa khi $n = 1$. Với $n = 1$ thì chỉ có một vòng đấu duy nhất. Bảng cần tìm có dạng:

Đội bóng	1	2
1	0	1
2	1	0

Ta kí hiệu $i : j$ nghĩa là đội bóng i đấu với đội bóng j ($1 \le i, j \le n$). Trong trường hợp $n = 4$, thì sẽ có ba vòng đấu. Nhưng thực hiện ghi như thế nào vào bảng thi đấu? Ta có thể chia các đội ra làm đôi bằng cách vòng thứ nhất có các trận 1 : 2 và 3 : 4, vòng thứ hai có các trận 1 : 3 và 2 : 4, còn vòng thứ ba có các trận 1 : 4 và 2 : 3. Ta nhận được bảng

Đội bóng	1	2	3	4
1	0	1	2	3
2	1	0	3	2
3	2	3	0	1
4	3	2	1	0

Ta xét kỹ bảng trên thấy rằng trong bảng này có chứa hai bảng trường hợp $n = 2$ và hai lần bảng

2	3
3	2

. Ta còn thấy thêm rằng bảng này bằng bảng

0	1
1	0

cộng thêm 2 vào mỗi phần tử.

Ta có thể có hướng cảm nhận để lập bảng thi đấu cho trường hợp $n = 8$. Với mục đích này ta phải xây dựng bảng có cỡ gấp đôi trường hợp cỡ $n = 4$. Ta tiến hành lập bảng bằng cách chép 4 lần bảng với $n = 4$. Sau đó cộng thêm vào mỗi phần tử với 4 ở những bảng có cỡ 4 ngoài đường chéo chính. Ta nhận được bảng

Đội bóng	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7
2	1	0	3	2	5	4	7	6
3	2	3	0	1	6	7	4	5
4	3	2	1	0	7	6	5	4
5	4	5	6	7	0	1	2	3
6	5	4	7	6	1	0	3	2
7	6	7	4	5	2	3	0	1
8	7	6	5	4	3	2	1	0

Dễ kiểm tra thấy rằng bảng trên là đúng, vì mỗi hàng mỗi cột đều chứa hoán vị của số từ 1 đến 7 không lặp lại, nghĩa là toàn giải thi đấu có 7 vòng và mỗi đội thi đấu với mỗi đội khác đúng một lần. Ta thấy rằng bảng trên đối xứng nếu $i : j$ thì cũng là vòng đấu của $j : i$ và ngược lại ($1 \leq i, j \leq n$), đường chéo chính của bảng gồm những số 0. Bằng cách hoàn toàn tương tự, ta lập được bảng thi đấu cho $n = 16, 32, 64, \ldots$ và nói chung n là lũy thừa của 2. Điều này có thể chứng minh bằng phương pháp quy nạp (tuy hơi cồng kềnh bạn đọc kiểm tra lấy). Ta có thể viết chương trình thông qua hai thủ tục sau: Trước nhất trong chương trình phải có:

$max := 100;$

$m : arrayof[1..max, 1..max];$

Ta lập thủ tục chép các số liệu vào ma trận như bảng sau :

Thuật toán 4.10 Chép số tự nhiên vào bảng

```
procedure   copymatrix(stX, stY, cnt, add)
  Đầu vào:   stX, stY, cnt, add
  Đầu ra:    Bảng m được cộng thêm add
      for i := 0 to cnt-1 do
         for j := 0 to cnt-1 do
            m[i + stX][j + stY] := m[i + 1][j + 1] + add;
         end for
      end for
end   copymatrix
```

Khi đó lập bảng bằng thủ tục: Để có một chương trình đầy đủ bạn đọc chỉ còn lập một thủ tục in ra mảng hai chiều được gán ở trên, điều này quá dễ đối với các ngôn ngữ lập trình ta đã học, ta không triển khai cụ thể ở đây.

Nhưng rất tiếc thuật toán này rất không thuận lợi cho n không phải lũy thừa của 2. Bạn đọc có thể thay đổi thủ tục này sao cho nó chạy với mọi n.

Sau đây ta tiếp cận bằng cách khác cho thuật toán cũng đúng với mọi n. Với số n lẻ ta tiến hành như sau: Ta không phải điền các số

Thuật toán 4.11 Lập bảng thi đấu

```
procedure   findsolution(n)
  Đầu vào:   n số đội bóng tham dự giải.
  Đầu ra:    Bảng m đầy đủ giải thi đấu.
     m[1][1] := 0;
     for i := 0 to n do
        copymatrix(i+1, 1, i, i);
        copymatrix(i+1, i+1, i, 0);
        copymatrix(1, i+1, i, i);
     end for
end   findsolution
```

vào bảng cỡ $n \times n$ mà là vào bảng mới có thêm một cột, nghĩa là bảng $n \times (n+1)$. Việc điền số thực hiện theo hàng bắt đầu từ cột trái thứ nhất và tiếp tục về bên phải. Khi ta đã điền vào cột thứ $n+1$ thì chuyển sang dòng sau. Bắt đầu điền vào giá trị 1 tại hàng 1, cột 1 lần lượt từng ô được điền các giá trị liên tiếp trong khoảng $[1,n]$, khi đạt đến giá trị n thì lại bắt đầu từ số 1 bất kể đang ở hàng nào, cột nào trong khi điền. Ví dụ với $n = 5$ cho kết quả:

Đội bóng	1	2	3	4	5	6
1	1	2	3	4	5	1
2	2	3	4	5	1	2
3	3	4	5	1	2	3
4	4	5	1	2	3	4
5	5	1	2	3	4	5

Bỏ cột cuối cùng và cho bằng 0 ở đường chéo chính trong bảng và phần còn lại là những số vòng các đội gặp nhau:

Đội bóng	1	2	3	4	5	6
1	0	2	3	4	5	1
2	2	0	4	5	1	2
3	3	4	0	1	2	3
4	4	5	1	0	3	4
5	5	1	2	3	0	5

Với n là số chẵn thì ta có thể sử dụng thuật toán đã nói ở phần trước.

Nhưng ở đây ta có thể thực hiện chiến thuật khác. Ta giải bài toán với $n-1$ là một số lẻ. Như thuật toán trên đến khi ta đã gán đường chéo chính của bảng bằng 0 và không bỏ cột thêm vào, ta ghép thêm một hàng với số liệu của cột ta đã thêm vào.

Ví dụ $n = 4$, trước tiên ta điền vào với $n = 3$

Đội bóng	1	2	3	4
1	1	2	3	1
2	2	3	1	2
3	3	1	2	3

Đội bóng	1	2	3	4
1	0	2	3	1
2	2	0	1	2
3	3	1	0	3
4	1	2	3	0

Sau đó ta thêm hàng thứ $n = 4$ như bảng trên (bên phải).

Thuật toán 4.12 Lập bảng thi đấu sửa đổi

```
procedure   findsolution(n)
  Đầu vào:   n số đội bóng tham dự giải.
  Đầu ra:    Bảng m đầy đủ lịch thi đấu.
     if (n  mod 2 = 0 then
        n := n − 1; {Nếu n chẵn thì đưa về n−1}
     end if
     {Điền vào bảng với n, biết chắc n là số lẻ.}
     for i := 0 to  n∗(n+1) − 1 do
        m[i  mod (n+1)][i div (n+1)] := i  mod n+1;
     end for
     if (n  mod 2 = 1 then
        n := n+1; {Lấy lại giá trị n chẵn}
     end if
     for i := 0 to  n do
        if n  mod 2 = 0 then
           m[i][n − 1] := m[n − 1][i] = m[i][i];{Điền hàng cuối cùng}
        end if
        m[i][i]:=0; {Điền đường chéo chính bằng 0}
     end for
end   findsolution
```

Cách giải trên rất hiệu quả nhưng không cho câu trả lời trực tiếp câu hỏi: Tại vòng thi đấu nào khi hai đội i và j gặp nhau. Điều này đòi hỏi thêm một chút quan sát bảng thi đấu đã lập để có công thức tính.

Thủ tục *findsolution* được sửa đổi như sau: Trường hợp ta có số chẵn đội tham dự $n = 2k$. Cho hai đội A và B có số s và t, mà chúng khác n. Khi đó hai đội sẽ đấu tại vòng thứ $s+t-1$, nếu $s+t \leq n$, và tại vòng $s+t-n$, nếu $s+t > n$. Sơ đồ này áp dụng cho giải đấu cờ quốc tế còn có ý nghĩa nữa là ai sẽ chơi quân đen và ai sẽ chơi quân trắng: Nếu $s+t$ là số lẻ, thì người chơi quân trắng sẽ là người có số nhỏ hơn, còn nếu $s+t$ là số chẵn thì ngược lại, người có số lớn hơn sẽ chơi quân trắng.

Thuật toán 4.13 Lập bảng thi đấu phương án cuối cùng

```
procedure   findsolution(n)
  Đầu vào:   n số đội bóng tham dự giải.
  Đầu ra:    Bảng m đầy đủ lịch thi đấu.
     if (n  mod 2 = 0 then
        n := n - 1; {Nếu n chẵn thì đưa về n - 1}
     end if
     {Điền vào bảng với n, biết chắc n là số lẻ}
     for i := 0 to  n do
        for j := 0 to  n do
           if i + j < n then
              m[i][j] := i + j + 1;
           else
              m[i][j] := i + j + 1 - n;
           end if
        end for
     end for
     if (n  mod 2 = 1 then
        n := n + 1; {Lấy lại giá trị n chẵn}
     end if
     for i := 0 to  n do
        if n  mod 2 = 0 then
           m[i][n - 1] := m[n - 1][i] = m[i][i];{Điền hàng cuối cùng}
        end if
        m[i][i]:=0; {Điền đường chéo chính bằng 0}
     end for
end   findsolution
```

Đội tham gia có số n đấu với đội có số ℓ tại vòng $2\ell - 1$, nếu $2\ell \leq n$ và tại vòng $2\ell - n$, nếu $2\ell > n$. Trong trường hợp đấu cờ quốc tế, người

tham gia mang số từ 1 đến $\frac{n}{2}$ chơi quân đen và số còn lại được chơi quân trắng.

Trường hợp số đội tham gia là lẻ $n = 2k + 1$ ta thêm một đội đấu giả và các đội đấu với đội này được nghỉ tại vòng tương ứng.

Ta có thể bỏ được đội chơi giả và điền trực tiếp những giá trị tương ứng của cột cuối cùng. Như vậy ta sửa đổi một chút như trong thuật toán 4.13.

4.7. BÀI TẬP

▷ **4.1.** Thuật toán tìm phần tử lớn nhất trong danh sách $L[1..n]$ được thiết kế theo phương pháp chia để trị *maximum*, khi sử dụng thuật toán thì gọi $maximum(1, n)$ như thuật toán **4.14** :

Thuật toán 4.14 Tìm giá trị lớn nhất theo chia để trị $maximum(x, y)$

function $maximum(x, y)$
Đầu vào: Chỉ số $x, y \in \mathbb{N}$ của $L[1..n]$.
Đầu ra: Giá trị lớn nhất trong $L[1..n]$.

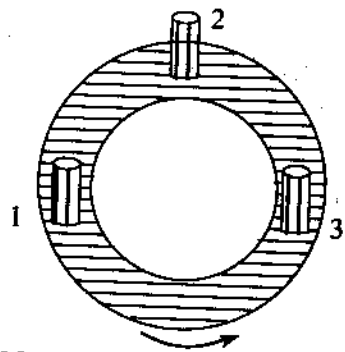
```
if y − x <= 1 then
    return(max{L[x], L[y]});
else
    max1 := maximum(x, ⌊(x+y)/2⌋);
    max2 := maximum(⌊(x+y)/2⌋ + 1, y);
    return(max{max1, max2});
end if
end maximum
```

1. Chứng minh rằng thuật toán 4.14 đúng. Bạn có thể giả thiết rằng n là lũy thừa của 2.

2. Viết công thức truy hồi theo số trường hợp xấu nhất của phép so sánh khi dùng $maximum(1, n)$. Giải phương trình hồi quy vừa thiết lập. Bạn có thể giả thiết n là lũy thừa của 2.

3. Đánh giá thời gian thực hiện thuật toán 4.14.

▷ **4.2.** Bài toán Tháp Hà Nội được nhắc tới trong chương 1, khi ta nói về hàm hồi quy. Cách tiếp cận giải bài toán như trong chương 1 cũng là cách thực hiện chia để trị. Nhưng định hướng hồi quy như thế nào thì ta hãy chứng minh thuật toán được mô tả sau đây giải được bài toán tháp Hà Nội:



Ngược chiều kim đồng hồ

Hình 4.2

Ta tưởng tượng có ba cột sắp xếp quanh vòng tròn với chiều di chuyển theo chiều kim đồng hồ là từ cột 1 sang cột 2 sang cột 3 lại sang cột 1. Như vậy việc một lần chuyển đĩa là chuyển theo chiều kim đồng hồ hoặc ngược chiều kim đồng hồ. Ta xét hai trường hợp sau:

1. Nếu ta có số lẻ đĩa ở cột thứ nhất thì:

- Bắt đầu chuyển đĩa nhỏ nhất theo chiều *ngược chiều kim đồng hồ*.
- Luân phiên nhau giữa cách làm như trên và chỉ đi những bước đi hợp pháp khác.

2. Nếu ta có số chẵn đĩa thì việc chuyển như trên được thay từ ngược chiều kim đồng hồ thành *theo chiều kim đồng hồ*.

CHƯƠNG 5
PHƯƠNG PHÁP QUY HOẠCH ĐỘNG

Phương pháp phân tích và thiết kế *chia để trị* ở chương trước phát huy tác dụng rất mạnh, do nó xuất phát từ việc chia nhỏ bài toán ban đầu thành hai bài toán con có cỡ gần bằng nhau để giải. Trong thực tế không có giới hạn chia bài toán thành bao nhiêu bài toán con để giải, cũng như cỡ của các bài toán con khác nhau như thế nào. Nhưng có một giới hạn quan trọng mà ta không để ý là không hai bài toán con nào giao nhau và việc giải chúng cho ta nghiệm trực tiếp của bài toán ban đầu. Phương pháp quy hoạch động nghiên cứu ở chương này thường liên quan tới những bài toán con có phần giao nhau là *bắt buộc*. Phần

sau đây ta sẽ xem xét hai điều kiện cần để áp dụng phương pháp quy hoạch động.

5.1. GIỚI THIỆU PHƯƠNG PHÁP QUY HOẠCH ĐỘNG

Một cách định nghĩa trừu tượng cho phương pháp quy hoạch động là tìm những giá trị cực trị (cực đại hoặc cực tiểu) trên một hàm, mà hàm này có đối số là một hàm khác. Phương pháp quy hoạch động nói chung có thể xem như phương pháp suy nghĩ hoặc quy hoạch, với nó thông qua việc tìm những nghiệm tối ưu cho một phần (hoặc tất cả) những tập con của một tập hợp, thì có thể tìm được nghiệm tối ưu của cả tập hợp.

Định nghĩa trên mang tính chất toán học ở mức trừu tượng. Nhưng còn đối với kỹ thuật lập trình và thiết kế thuật toán thì sao? Phương pháp này tương tự như phương pháp chia để trị: Bài toán ban đầu được chia thành những bài toán con, những bài toán này đến lượt lại chia thành bài toán nhỏ nữa, ... đến lúc đạt được bài toán đủ đơn giản để giải trực tiếp được. Sau đó những nghiệm của bài toán con được tổng hợp lại theo một phương pháp thích hợp để nhận được nghiệm của bài toán ban đầu. Khác với phương pháp chia để trị, phương pháp quy hoạch động không đòi hỏi những bài toán con không giao nhau, điều này có nghĩa là phương pháp quy hoạch động tổng quát hơn phương pháp chia để trị. Chính những đòi hỏi không giao nhau của các bài toán con dẫn đến những vấn đề thuật toán không hiệu quả. Trong trường hợp chung những bài toán con có độ phức tạp đa thức của bài toán ban đầu, nhưng quá trình kết hợp lại thì cùng những bài toán con được giải lại một số lần, cuối cùng dẫn đến độ phức tạp hàm mũ cho thuật toán.

5.1.1. Ví dụ so sánh với phương pháp chia để trị

Ta biết rằng hệ số Newton biểu diễn một khái niệm tổ hợp : Chọn r vật trong tổng n vật đã cho thì có C_n^r cách khác nhau. Bằng quy nạp ta cũng có thể dễ dàng chứng minh đẳng thức sau:

$$C_n^r = C_{n-1}^{r-1} + C_{n-1}^r.$$

Như vậy để thiết lập thuật toán tính những tổ hợp của việc chọn r vật từ n vật đã cho ta có thể dùng phương pháp chia để trị trực tiếp theo công thức trên:

Thuật toán 5.1 Tính hệ số Newton (choose)

```
function   choose(n, r)
  Đầu vào:  Hai số nguyên dương n, r.
  Đầu ra:   Kết quả C_n^r.
  if r = 0 ∨ n = r then
      return(1);
  else
      return(choose(n − 1, r − 1) + choose(n − 1, r));
  end if
end   choose
```

Tính đúng đắn của thuật toán có thể chứng minh bằng quy nạp theo n dễ dàng (giống như thuật toán Fibonacci ở chương 2).

Phân tích thuật toán: Cho $T(n)$ là thời gian tính toán $choose(n,r)$ trong trường hợp xấu nhất trên tất cả giá trị có khả năng của r. Khi đó dễ thấy rằng

$$T(n) = \begin{cases} c & \text{nếu } n = 1 \\ 2T(n-1)+d & \text{ngược lại} \end{cases},$$

ở đây c, d là những hằng số nào đó.

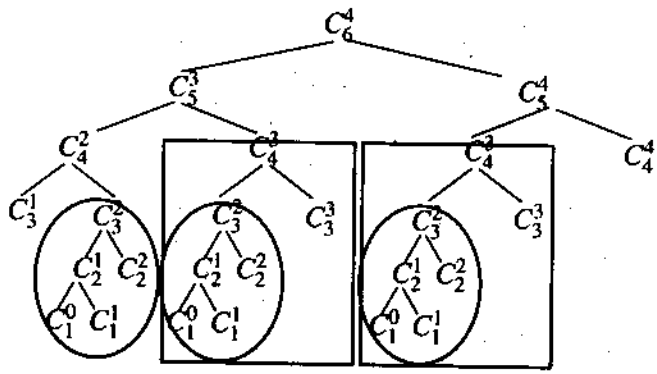
Do đó,

$$\begin{aligned} T(n) &= 2T(n-1)+d = 2(2T(n-2)+d)+d = 4T(n-2)+2d+d \\ &= 4(2T(n-3)+d)+2d+d = 8T(n-3)+4d+2d+d = \cdots \\ &= 2^iT(n-i)+d\sum_{j=0}^{i-1}2^j = 2^{n-1}T(1)+d\sum_{j=0}^{n-2}2^j \\ &= (c+d)2^{n-1}-d. \end{aligned}$$

Do đó, $T(n) = \Theta(2^n)$. Như vậy với thuật toán được thiết kế bằng phương pháp chia để trị thì thời gian tính toán là hàm mũ rất lớn khi n tăng cao.

Nguyên nhân nào đã dẫn đến độ phức tạp tính toán của thuật toán tăng vọt như vậy, đó chính là những bài toán con được giải đi giải lại

nhiều lần. Ta mô tả thuật toán trên bằng hình cây với $n = 6$ và $r = 4$. Khi đó cây tính toán trong trường hợp thuật toán chia để trị này có những phần giống nhau được khoanh như hình 5.1:



Hình 5.1

Để khắc phục tình trạng phải tính lại người ta ghi kết quả ra một bảng để lấy vào cho các lần tính sau chứ không tính lại nữa.

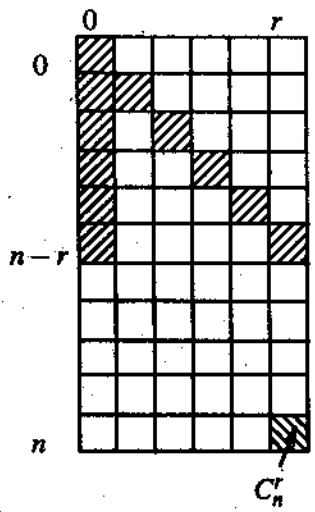
Thuật toán 5.2 Tính hệ số Newton theo quy hoạch động (choose)

```
function   choose(n, r)
  Đầu vào:  Hai số nguyên dương n, r.
  Đầu ra:   Kết quả B[n, r] = C_n^r.
  Khai báo mảng B[1..n, 1..n]
  for i := 0 to n − r do
      B[i, 0] := 1;
  end for
  for i := 0 to r do
      B[i, i] := 1;
  end for
  for j := 1 to r do
      for i := j + 1 to n − r + j do
          B[i, j] := B[i − 1, j − 1] + B[i − 1, j];
      end for
  end for
  return(B[n, r])
end   choose
```

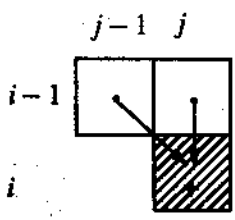
Một trong những biện pháp theo ý tưởng trên được thực hiện như sau : Ta dùng mảng hai chiều $B[0..n, 0..n]$ như một bảng để ghi lại kết quả đang tính, mỗi giá trị $B[i, j] = C_i^j$. Ta sử dụng tam giác Pascal để tính các hệ số Newton tiếp theo bằng thuật toán 5.2:

Thuật toán được khởi động bằng cách tạo các cạnh của tam giác Pascal và đòi hỏi kết quả ở cuối bảng như hình 5.2.

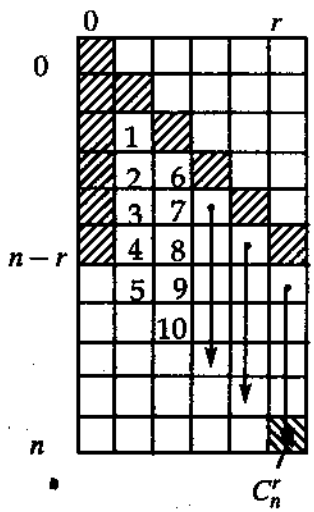
Điền những ô trống $B[i, j]$ phải cần các ô đã có $B[i-1, j-1]$ và $B[i-1, j]$. Nghĩa là cộng hai ô trên điền xuống ô dưới như sơ đồ hình 5.3.



Hình 5.2



Hình 5.3



Hình 5.4

Thuật toán thực hiện điền theo cột từ trái qua phải. Điền vào mỗi ô của các cột từ trên xuống dưới (hình 5.4).

Phân tích thuật toán có bảng: Thuật toán hoạt động bằng cách điền hết vào các ô trong bảng, kích thước số ô bao nhiêu thì thuật toán phải gán từng ấy lần. Dễ thấy có tất cả số ô phải điền là

$$(n-r+1)(r+1) = nr + n - r^2 + 1 \leq n(r+1) + 1.$$

Mỗi ô điền vào được tính thời gian là $O(1)$ thì tổng tất cả thời gian đòi hỏi của thuật toán là $O(n^2)$. Thời gian đòi hỏi này còn tốt hơn nhiều $O(2^n)$ như thuật toán trước đã tính.

5.1.2. Một số vấn đề trong phương pháp quy hoạch động

Như vậy ta có thể định nghĩa *phương pháp quy hoạch động* cho những người lập trình và thiết kế thuật toán là *điền vào bảng những kết quả lời giải của bài toán con đã giải với mục đích thoát khỏi việc tính toán thừa.*

Tuy nguyên lý quy hoạch động được áp dụng cho những bài toán có một nghiệm duy nhất, như bài toán tính hệ số Newton ở trên, nhưng ứng dụng cơ bản của nó còn dùng để giải những bài toán tối ưu. Câu hỏi nảy sinh với những bài toán có nhiều nghiệm khác nhau, trong những nghiệm đó ta phải chọn lấy một. Trên mỗi nghiệm dự bị ta sẽ đặt tương ứng với một số nào đó nhờ sự định nghĩa hàm trợ giúp. Mục đích là tìm nghiệm mà với nó những hàm này nhận giá trị cực trị (cực đại hoặc cực tiểu). Những hàm này ta gọi là hàm mục đích. Có thể bài toán có nhiều nghiệm tối ưu, nhưng trong trường hợp chung áp dụng phương pháp quy hoạch động chỉ cho ta một nghiệm trong chúng.

Ví dụ phần trước thể hiện mặt mạnh của quy hoạch động. Nhưng phương pháp cũng phải trả giá: Phần cơ bản của phương pháp quy hoạch động là giải những bài toán con của bài toán ban đầu với cỡ nhỏ hơn và lưu giữ những kết quả đã tính, nghĩa là được về vận tốc tính toán đối với bộ nhớ. Trong một số trường hợp ta cần sẵn một số bộ nhớ (như bài toán được khai báo mảng ở trên). Nhưng một số thuật toán bộ nhớ có thể tăng tuyến tính, hoặc tăng theo đa thức bậc hai, hoặc đôi khi cần bộ nhớ rất lớn, những phần sau ta sẽ xem xét kĩ hơn.

Phải nhấn mạnh rằng phương pháp quy hoạch động không luôn luôn ứng dụng được. Một mặt, nghiệm của bài toán ban đầu không phải lúc nào cũng có thể nhận được từ tổ hợp những kết quả giải của một phần hoặc tất cả những bài toán con của nó. Mặt khác, thậm chí cả khi ta tổ hợp được nghiệm, có thể lại thấy số lượng những bài toán con vô cùng lớn không thể chấp nhận được. Thêm vào đó nhiều khi thiếu những tiêu chuẩn rõ ràng đặc trưng cho bài toán để giải bằng phương pháp này: Ta biết rằng hàng loạt các bài toán mà những thuật toán cơ bản hiệu quả hơn rất nhiều thuật toán quy hoạch động, còn một số bài toán khác nói chung không áp dụng được.

Tồn tại hai điều kiện cần để áp dụng phương pháp quy hoạch động: *cấu trúc con tối ưu của nghiệm* và *sự phủ nhau của những bài toán con*.

1. Những cấu trúc tối ưu con của nghiệm nghĩa là nghiệm tối ưu của bài toán ban đầu có thể tìm được thông qua tổ hợp những nghiệm tối ưu của các bài toán con. Thường thường thỏa mãn những tiêu chuẩn nào đó đưa đến tìm những không gian thích hợp tự nhiên của những bài toán con. Một ví dụ như vậy trong việc giải bài toán nhân những ma trận. Trong bài toán ta chỉ cần các dãy con những ma trận từ dãy các ma trận ban đầu là đủ (không phải là những tập hợp con bất kì). Điều này đưa đến giảm rất nhiều số lượng bài toán con, từ đó phương pháp tìm nghiệm có hiệu quả hơn.

2. Điều kiện cần để sử dụng phương pháp quy hoạch động là sự phủ nhau của các bài toán con. Phương pháp quy hoạch động đưa ra cách khắc phục những bài toán con phủ nhau này, đồng thời tính nghiệm cho mỗi bài toán con chỉ một lần và hạn chế số lượng phải giải những bài toán con. Càng hạn chế giải những bài toán con mới thì phương pháp càng có hiệu quả. Sự mất đi phần phụ thuộc nhau của những bài toán con (phần giao của hai bài toán con) đưa đến sử dụng quy hoạch động không hiệu quả. Trong trường hợp này ta nên giải bằng phương pháp chia để trị, chắc chắn sẽ cho kết quả tốt hơn.

5.1.3. Kĩ thuật lập thuật toán theo quy hoạch động

Phương pháp quy hoạch động thực chất là giải bài toán tương tác từ dưới lên, nghĩa là trước tiên giải những bài toán con hiển nhiên (giải trực tiếp được), sau đó những bài toán như vậy lớn dần lên, những bài toán con này được giải và ... tuy cách tiếp cận này phải giải tất cả những bài toán con với cỡ nhỏ hơn. Trong thực tế nhược điểm trên không ảnh hưởng đến tốc độ thuật toán. Một hướng khác là giải theo hàm hồi quy. Phương pháp này lại có cách tiếp cận từ trên xuống. Nhưng người ta thiết lập bảng chứa những nghiệm bài toán con đã giải và từ đó giải tiếp các bài toán con tiếp theo. Qua ví dụ thuật toán tìm hệ số Newton ở trên và kinh nghiệm thực tế người ta đưa ra kĩ thuật để thiết kế thuật toán quy hoạch động như sau:

1. Xác định các định tính của thuật toán

- Thiết lập thuật toán chia để trị cho bài toán (chương 4).
- Phân tích thuật toán, thấy rằng thời gian chạy là hàm mũ.
- Một số bài toán con được giải đi giải lại nhiều lần.

2. Xây dựng thuật toán quy hoạch động

Thuật toán thiết kế theo chia để trị

```
function choose (n,r)
  if r=0 or r=n then return(1) else
      return(choose(n-1,r-1),choose(n-1,r))
```

Thuật toán thiết kế theo quy hoạch động

```
function choose (n,r)
  for i:=0 to n-r do B[i,0]:=1
  for i:=0 to r do B[i,i]:=1
  for j:=1 to r do
      for i:=j+1 to n-r+j do
          B[i,j]:=B[i-1,j-1]+B[i-1,j]
  return(B[n,r])
```

Hình 5.5

- Lấy những phần thuật toán chia để trị, nhất là phần gọi hồi quy và thay lời gọi hồi quy bằng những tìm kiếm trên bảng.
- Lập mảng ghi giá trị trả về vào một ô của mảng.
- Sử dụng cơ sở khởi tạo của chia để trị để điền vào những ô bắt đầu của bảng.
- Tạo ra những *tìm kiếm trung gian* giữa các phần tử của bảng.
- Tạo ra những vòng lặp để điền hết bảng có dùng *tìm kiếm trung gian.*

Hình 5.5 mô tả quá trình trên thông qua ví dụ tìm hệ số Newton. Những phần sau mô tả đầy đủ theo sơ đồ này.

5.2. BÀI TOÁN NHÂN MA TRẬN

5.2.1. Giới thiệu bài toán

Bài toán nhân dãy ma trận bao hàm vấn đề xác định tối ưu một dãy thực hiện chuỗi các thao tác. Lớp bài toán tổng quát quan trọng này trong thiết kế bộ biên dịch cho mã tối ưu và trong các cơ sở dữ liệu cho tối ưu truy vấn. Ta chỉ xét ở đó bài toán rất hạn chế mà ở đây có thể sử dụng phương pháp quy hoạch động một cách dễ thấy nhất.

Ta xét bài toán nhân n ma trận hình chữ nhật thích hợp với nhau:

$$M = M_1 \times M_2 \times \cdots \times M_n,$$

ở đây M_i có cỡ $r_{i-1} \times r_i$ (r_{i-1} hàng và r_i cột), $i = 1, 2, ..., n$.

Theo định nghĩa phép nhân hai ma trận A và B có cỡ tương ứng là $p \times q$ và $q \times r$ thực hiện theo công thức

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}, i = 1, 2, ..., p; j = 1, 2, ..., r$$

và kết quả nhận được một ma trận C có cỡ $p \times r$.

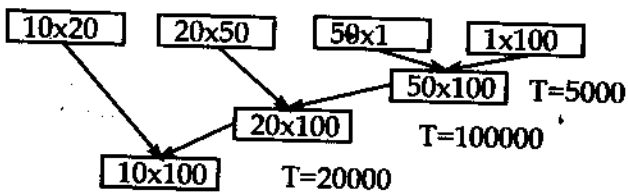
Dễ thấy số những phép nhân cơ bản trong công thức trên là pqr. Ta biết rằng tồn tại những phương pháp nhân mà phép nhân cơ bản giảm đáng kể, đặc biệt là với những ma trận có cỡ lớn. Một phương pháp đó là của Strassen đã xét ở chương 4.

Cho hai ma trận A và B có cỡ tương ứng là $p \times q$ và $s \times r$. Ta nhớ lại rằng nếu số lượng cột của A là q bằng với số lượng hàng s của B, thì ta có thể thực hiện phép nhân AB (nhưng không thể BA vì nó đòi hỏi $r = p$). Như vậy phép nhân không phải là phép toán giao hoán giữa hai ma trận. Do đó bài toán đã cho nhân dãy ma trận không thể chuyển chỗ các ma trận được, mà chỉ có kết hợp tại chỗ với nhau mà thôi. Mặt khác phép nhân ma trận có tính chất kết hợp, nghĩa là cho ba ma trận A, B, C có các cỡ thích hợp thì ta có $(AB)C = A(BC)$. Ta có thể chứng minh theo quy nạp thứ tự phép nhân những ma trận không ảnh hưởng đến kết quả khi ta kết hợp các phép nhân khác nhau.

Tuy thứ tự phép nhân không ảnh hưởng đến ma trận kết quả, nhưng thời gian thực hiện phép nhân có thể thực sự khác nhau với những sơ đồ

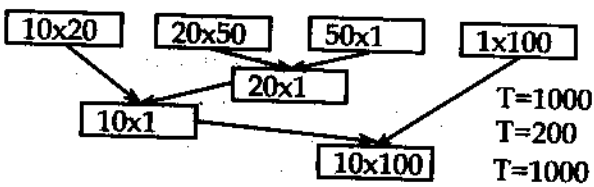
khác nhau. Nguyên nhân là do số lượng những phép nhân cơ bản khác nhau. Ta xét ví dụ: Cho những ma trận có cỡ tương ứng là $M_1(10 \times 20)$, $M_2(20 \times 50)$, $M_3(50 \times 1)$, $M_4(1 \times 100)$, kí hiệu hàm số lượng phép nhân giữa các ma trận là $T[AB] = pqr$ với A và B có các cỡ tương ứng $p \times q$ và $q \times r$. Khi đó ta xét những thứ tự nhân khác nhau:

Phép nhân $M = M_1 \times (M_2 \times (M_3 \times M_4))$ cho kết quả trong hình 5.6.



Hình 5.6 $T[(M_1(M_2(M_3M_4))] = 5000 + 100000 + 20000 = 125000$

Tuy nhiên nếu bắt đầu từ ở giữa sơ đồ $M = (M_1 \times (M_2 \times M_3)) \times M_4$, thì kết quả trong hình 5.7.



Hình 5.7 $T[(M_1(M_2M_3)M_4)] = 1000 + 200 + 1000 = 2200$

Bài toán nhân dãy ma trận: Cho dãy ma trận $M = M_1 \times M_2 \times \cdots \times M_n$ và số chiều $r_0, r_1, \ldots, r_n$, ở đây M_i có cỡ $r_{i-1} \times r_i$, $i = 1, 2, \ldots, n$. Hãy xác định thứ tự phép nhân sao cho số phép nhân cơ bản nhỏ nhất.

Chú ý: Thuật toán này không thực hiện những phép nhân ma trận, mà chỉ ra thứ tự tốt nhất để thực hiện những phép nhân này.

Một cách thiết kế thuật toán vụng về nhất là ta cố gắng thử tất cả các cách đóng mở ngoặc cho n ma trận. Nhưng chính cách thử này cũng có thời gian hàm số mũ, vì tồn tại hàm số mũ cách bỏ dấu ngoặc giữa những ma trận này.

Thật vậy, Gọi $X(n)$ là số cách bỏ dấu ngoặc tròn cho n ma trận. Dễ

thấy rằng $X(1) = X(2) = 1$ và với $n > 2$,

$$X(n) = \sum_{k=1}^{n-1} X(k).X(n-k).$$

Điều này tự nhiên như một tích được bẻ ra làm đôi như:

$$\underbrace{(M_1 \times M_2 \times \cdots \times M_k)}_{X(k)\text{ cách}} \times \underbrace{(M_{k+1} \times M_{k+2} \times \cdots \times M_n)}_{X(n-k)\text{ cách}}$$

Do đó,

$$X(3) = \sum_{k=1}^{2} X(k).X(3-k) = X(1).X(2) + X(2).X(1) = 2.$$

Dễ kiểm tra chỉ có $M(MM)$, $(MM)M$.

$$X(4) = \sum_{k=1}^{3} X(k).X(4-k) = X(1).X(3) + X(2)X(2) + X(3)X(1) = 5.$$

Đó là $M((MM)M)$, $M(M(MM))$, $(MM)(MM)$, $((MM)M)M$, $(M(MM))M$.

Mệnh đề 5.1. *Chứng minh rằng* $X(n) \geq 2^{n-2}$ *với mọi* $n > 2$.

Chứng minh. Ta chứng minh bằng quy nạp toán học theo n. Qua những ví dụ trên bất đẳng thức đúng với $n \leq 4$. Giả sử bất đẳng thức đúng với mọi $n-1$ và $n \geq 5$. Khi đó

$$X(n) = \sum_{k=1}^{n-1} X(k).X(n-k) \geq \sum_{k=1}^{n-1} 2^{k-2}2^{n-k-2} \qquad \text{(theo giả thiết quy nạp)}$$

$$= \sum_{k=1}^{n-1} 2^{n-4} = (n-1)2^{n-4} \geq 2^{n-2} \qquad (\text{vì } n \geq 5).$$

Người ta có thể chỉ ra rằng $X(n) = \frac{1}{n}C_{2(n-1)}^{n-1}$. ☺

5.2.2. Thiết kế thuật toán theo cách chia để trị

Theo cách thiết kế thuật toán chia để trị thì ta chia nhỏ bài toán ra thành các bài toán con, giải các bài toán con đó và cuối cùng là lấy lời giải tốt nhất từ các bài toán con. Cho $recmatrix(p, i, j)$ là số phép tính nhỏ nhất để tính tích

$$M_i \times M_{i+1} \times \cdots \times M_j,$$

ở đây mảng $p[1..n]$ chứa cỡ của ma trận $r_0, r_1, ..., r_n$. Khi tách tích trên làm hai tích tại ma trận M_k

$$(M_i \times M_{i+1} \times \cdots \times M_k) \times (M_{k+1} \times \cdots \times M_j)$$

và sau đó nhân hai phần lại thì tổng của các phép tính là bao nhiêu? Dễ thấy tại mỗi đoạn ngắt ra có phép tính là $recmatrix(p,i,k)$, $recmatrix(p,k+1,j)$ và cuối cùng ta nhân hai ma trận của mỗi đoạn là ma trận cỡ $r_{i-1} \times r_k$ với ma trận cỡ $r_k \times r_j$.

Do đó, tổng của số phép tính được ngắt tại ma trận M_k bằng

$$recmatrix(p,i,k) + recmatrix(p,k+1,j) + p[i-1]p[k]p[j].$$

Từ lí luận phần trên ta có thể lập một thuật toán để tính giá trị nhỏ nhất của phép nhân khi ta ngắt lần lượt từ 1 đến n khi gọi hàm $recmatrix(p,1,n)$.

1. Thuật toán kiểu chia để trị

Thuật toán 5.3 Nhân dãy ma trận theo hồi quy (*recmatrix*)

function *recmatrix*$(p[1..n], i, j)$
  Đầu vào: Mảng $p[1..n]$ và hai số nguyên i, j.
  Đầu ra: Kết quả $recmatrix(p,i,j)$ giá trị nhỏ nhất.
  if $i = j$ **then**
    $m[i,j] := 0$;
  else
    $m[i,j] := \infty$;
    for $k := i$ **to** $j-1$ **do**
      $cost := recmatrix(p,i,k) + recmatrix(p,k+1,j)$
        $+ p[i-1]p[k]p[j]$;
      if $cost < m[i,j]$ **then**
        $m[i,j] := cost$;
      end if
    end for
  end if
  return$(m[i,j])$;
end *recmatrixchain*

2. Chứng minh tính đúng đắn của thuật toán

Đây là thuật toán hồi quy dễ dàng chứng minh bằng phương pháp

quy nạp toán học theo $n = j - i + 1$ (bạn đọc tự chứng minh).

3. Phân tích thuật toán

Gọi $T(n)$ là thời gian chạy thủ tục *recmatrix*(p, i, j) trong trường hợp xấu nhất khi mà $j - i + 1 = n$. Từ thuật toán dễ thấy

$$T(n) = \begin{cases} c & \text{nếu } n = 1, \\ \sum_{m=1}^{n-1} (T(m) + T(n-m)) + dn & \text{nếu } n \geq 2 \end{cases}$$

ở đây c, d là những hằng số.

Do đó, với $n > 0$,

$$T(n) = 2\sum_{m=1}^{n-1} T(m) + dn.$$

Vì thế,

$$\begin{aligned} T(n) - T(n-1) &= (2\sum_{m=1}^{n-1} T(m) + dn) - (2\sum_{m=1}^{n-2} T(m) + d(n-1)) \\ &= 2T(n-1) + d. \end{aligned}$$

Nghĩa là,

$$T(n) = 3T(n-1) + d,$$

và như vậy bằng cách thay liên tiếp, ta có

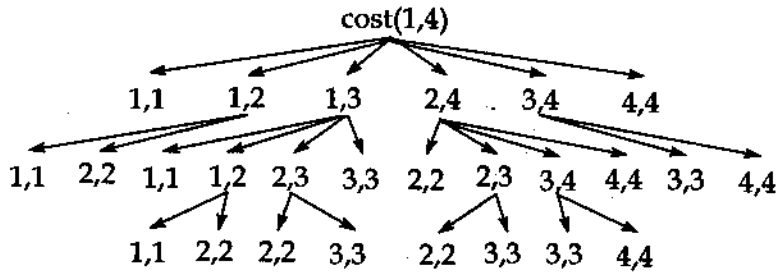
$$\begin{aligned} T(n) &= 3T(n-1) + d = 3(3T(n-2) + d) + d = 9T(n-2) + 3d + d \\ &= 9(3T(n-3) + d) + 3d + d = 27T(n-3) + 9d + 3d + d \\ &\cdots \\ &= 3^m T(n-m) + d\sum_{\ell=0}^{m-1} 3^\ell = 3^n T(0) + d\sum_{\ell=0}^{n-1} 3^\ell \\ &= c3^n + d\frac{3^n - 1}{2} = (c + \frac{d}{2})3^n - \frac{d}{2}. \end{aligned}$$

Do đó $T(n) \in \Theta(3^n)$. Thuật toán theo chia để trị có độ phức tạp thời gian là hàm mũ.

4. Ví dụ: Vẽ sơ đồ một ví dụ nhỏ để thấy rằng trong tính toán thuật toán đã lặp đi lặp lại nhiều lần một số thủ tục (hình 5.8).

5.2.3. Thiết kế theo quy hoạch động

Như vậy thuật toán chia để trị cho việc nhân một số ma trận dẫn đến hàm thời gian là hàm mũ và trong thủ tục hồi quy gọi lặp lại nhiều



Hình 5.8

lần một số chu trình con. Do vậy điều kiện tốt cho việc ứng dụng thiết kế theo quy hoạch động, nghĩa là ghi lại các kết quả tính toán trung gian vào một bảng.

Ta ghi giá trị $cost(i,j)$ vào mảng $m[i,j]$. Nghĩa là $m[i,j]$ chứa số nhỏ nhất những phép nhân của chuỗi ma trận $M_i \times M_{i+1} \times \cdots \times M_j$. Giá trị tối ưu để tính đại lượng này được mô tả như sau :

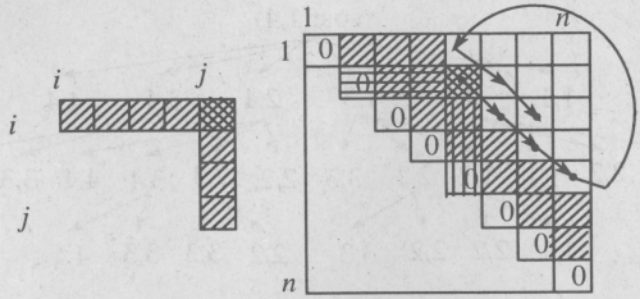
Bước cơ sở : Nếu $i = j$ thì dãy ma trận chỉ có một ma trận và vì thế không có phép nhân nào. Như vậy, $m[i,i] = 0$.

Bước thực hiện : Nếu $i < j$, thì ta có thể tách dãy đã cho thành hai dãy tại k với $i \leq k < j$: $(M_i \times M_{i+1} \times \cdots \times M_k) \times (M_{k+1} \times \cdots \times M_j)$.

Số những phép nhân nhỏ nhất trong $M_i \times M_{i+1} \times \cdots \times M_k$ được ghi vào $m[i,k]$ và số những phép nhân nhỏ nhất trong $M_{k+1} \times \cdots \times M_j$ được ghi vào $m[k+1,j]$. Dễ thấy rằng cỡ ma trận của đoạn thứ nhất là $r_{i-1} \times r_k$ và cỡ ma trận của đoạn thứ hai là $r_k \times r_j$ và khi nhân hai ma trận kết quả của phân chia tại k phải thực hiện $r_{i-1} \cdot r_k \cdot r_j$ phép nhân. Tóm lại ta có

$$m[i,j] = \begin{cases} 0 & \text{nếu } i = j \\ \min\limits_{i \leq k < j} \{m[i,k] + m[k+1,j] + r_{i-1} r_k r_j\} & \text{ngược lại .} \end{cases}$$

Như vậy, nhìn vào công thức trên để điền hết bảng ta phải biết những giá trị $m[i,k]$ và $m[k+1,j]$ đã được điền vào từ trước. Để tính một ô trong bảng ta tăng các thông số, có thể nói tính $m[i,j]$ là đã tính được hàng và cột này từ các ô $m[i,i]$ và $m[j,j]$, như sơ đồ hình 5.9:



Hình 5.9

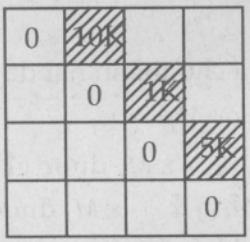
Ví dụ : Như ví dụ ban đầu ta có

$$r_0 = 10, r_1 = 20, r_2 = 50, r_3 = 1, r_4 = 100.$$

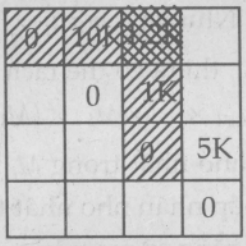
Theo thiết kế trên điền vào bảng như sau:

$$m[1,2] = \min_{1 \le k < 2} \{m[1,k] + m[k+1,2] + r_0 r_k r_2\} = r_0 r_1 r_2 = 10000$$

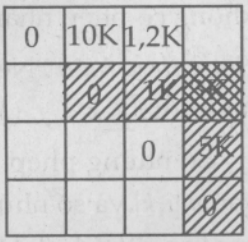
$$m[2,3] = r_1 r_2 r_3 = 1000; m[3,4] = r_2 r_3 r_4 = 5000 \qquad \text{(hình 5.10).}$$



Hình 5.10



Hình 5.11



Hình 5.12

$$m[1,3] = \min_{1 \le k < 3} \{m[1,k] + m[k+1,3] + r_0 r_k r_3\} \qquad \text{(hình 5.11)}$$

$$= \min\{m[1,1] + m[2,3] + r_0 r_1 r_3, m[1,2] + m[3,3] + r_0 r_2 r_3\}$$

$$= \min\{0 + 1000 + 200, 10000 + 0 + 500\} = 1200$$

$$m[2,4] = \min_{2 \le k < 4} \{m[2,k] + m[k+1,4] + r_1 r_k r_4\} \qquad \text{(hình 5.12)}$$

$$= \min\{m[2,2] + m[3,4] + r_1 r_2 r_4, m[2,3] + m[4,4] + r_1 r_3 r_4\}$$

$$= \min\{0 + 5000 + 100000, 1000 + 0 + 2000\} = 3000$$

$$
\begin{aligned}
m[1,4] &= \min_{1\le k<4}\{m[2,k]+m[k+1,4]+r_0r_kr_4\}\\
&= \min\{m[1,1]+m[2,4]+r_0r_1r_4,\\
&\quad m[1,2]+m[3,4]+r_0r_2r_4,\\
&\quad m[1,3]+m[4,4]+r_0r_3r_4\}\\
&= \min\{0+3000+20000,\\
&\quad 10000+5000+50000,\\
&\quad 1200+0+1000\} = 2200 \quad \text{(hình 5.13,)}
\end{aligned}
$$

0	10K	1,2K	3K
	0	1K	3K
		0	5K
			0

Hình 5.13

Thuật toán : Không khó để chuyển đổi mô tả trên thành thuật toán. Trong quá trình tính $m[i,j]$ ta cần tới những giá trị $m[i,k]$ và $m[k+1,j]$ với k nằm giữa i và j. Điều này gợi ý ta phải tính theo số ma trận trong một mảng con. Ta đặt $d = j-i+1$ là độ dài dãy ma trận con phải nhân. Dãy có độ dài bằng 1 như $m[i,i]$ là hiển nhiên. Ta sẽ tính cho dãy con có độ dài $2,3,\ldots,n$. Cuối cùng là $m[1..n]$. Ta cần cẩn thận trong khi đặt vòng lặp. Nếu một chuỗi có độ dài d bắt đầu từ vị trí i, thì $j = i+d-1$. Vì ta muốn $j \le n$, nghĩa là $i+d-1 \le n$ hoặc nói một cách khác $i \le n-d+1$. Như vậy vòng lặp cho i chạy từ 1 đến $n-d+1$.

Mảng $s[i,j]$ mục đích là ghi lại vị trí bỏ dấu ngoặc sẽ được giải thích sau. Ta thấy thuật toán thực hiện tính toán trực tiếp theo các dãy lệnh. Và dễ thấy độ phức tạp của thuật toán này là $\Theta(n^3)$. Ta có thể chứng minh dễ dàng điều này vì khối tính toán lớn nhất của thuật toán là ba vòng lặp lồng nhau, mà mỗi vòng lặp đều chạy nhiều nhất là n lần. Như vậy thuật toán 6.4 (trang 148) chạy nhanh hơn rất nhiều thuật toán được thiết kế bằng hồi quy ở trên.

Dãy kết quả ma trận được chèn dấu : Tư tưởng chính là để lại dấu ngoặc tại những vị trí chia dãy con cho kết quả tốt nhất, nghĩa là giá trị của k mà nó cho giá trị nhỏ nhất của $m[i,j]$. Ta có thể thực hiện song song điều này bằng cách ghi lại tại $s[i,j]$ giá trị k cho phép chia tối ưu nhất. Ví dụ giả sử $s[i,j] = k$. Điều này nói với ta rằng cách tốt nhất để nhân dãy con các ma trận $M_i \times \cdots \times M_j$ là trước nhất hãy nhân $M_i \times M_{i+1} \times \cdots \times M_k$ và sau đó là nhân $M_{k+1} \times \cdots \times M_j$ và cuối cùng nhân chúng với nhau. Ta chú ý rằng chúng ta chỉ cần thiết ghi vào $s[i,j]$ khi

Thuật toán 5.4 Nhân dãy ma trận theo quy hoạch động (matrixchain)

```
function  matrixchain(p[1..n], n)
  Đầu vào:  Mảng p[1..n] chứa cỡ các ma trận và số nguyên n.
  Đầu ra:   Kết quả m[1,n] giá trị nhỏ nhất và s dấu chỗ chia dãy.
  Khai báo mảng s[1..n − 1, 2..n];
  for i:=1 to n do
     m[i,i] := 0;
  end for
  for d := 2 to n  do
     for i := 1 to n − d + 1 do
        j := i + d − 1;
        m[i, j] := ∞;
        for k := i to j − 1 do
           q := m[i,k] + m[k + 1, j] + p[i − 1] * p[k] * p[j];
           if q < m[i, j] then
              m[i, j] := q;
              m[i, j] := k;
           end if
        end for
     end for
  end for
  return( m[1,n] và s)
end  matrixchain
```

mà có ít nhất hai ma trận, nghĩa là $j > i$.

Bảng số liệu trong ví dụ số cụ thể được thể hiện ở hình 5.14 :

	$j \rightarrow$ 1	2	3	4	
$r_0 = 10$	0	10K	1,2K	2.2K	1
M_1 $r_1 = 20$		0	1K	3K	2
M_2 $r_2 = 50$			0	5K	3
M_3 $r_3 = 1$				0	4
M_4 $r_4 = 100$					

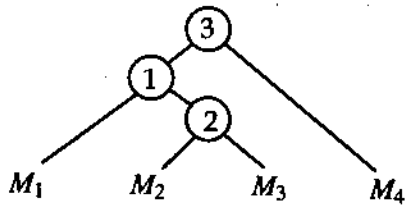
$m[i, j]$ i

$j \rightarrow$ 1	2	3	4	
	1	1	3	1
		2	3	2
			3	3
				4

$s[i, j]$ i

Hình 5.14

Từ sơ đồ trên ta thấy thứ tự cuối cùng của phép nhân $(M_1(M_2M_3))M_4$ được thể hiện trên hình cây (hình 5.15).



Hình 5.15

Khi đã có $s[i,j]$ ta dễ dàng thiết kế thủ tục đưa dãy tượng trưng các ma trận và các ngoặc tròn tại vị trí tối ưu như đã mô tả ở trên (vấn đề này bạn đọc tự làm lấy).

Một thuật toán nhân cần thiết sử dụng giá trị $s[i,j]$ để xác định cách nhân cho kết quả một ma trận. Giả sử những ma trận được ghi vào mảng các ma trận $A[1..n]$ và $s[i,j]$ đã được tính từ thủ tục trên. Thủ tục để nhận được ma trận kết quả cuối cùng là:

Thuật toán 5.5 Ma trận kết quả nhân một dãy ma trận (multmatrix)

function *multmatrix*(i,j)
  Đầu vào: Mảng ma trận $A[1..n]$ và $s[i,j]$.
  Đầu ra: Ma trận tích của các ma trận mảng trên.
  Đọc vào giá trị $s[1..n-1, 2..n]$;
  Đọc vào ma trận $A[1..n]$
  if i=j **then**
    return$(A[i])$;
  else
    $k := s[i,j]$;
    $X :=$ *multmatrix*(i,k);
    $Y :=$ *multmatrix*$(k+1,j)$;
    return$(X * Y)$
  end if
end *multmatrix*

5.3. GHI NHỚ HÓA VÀ TAM GIÁC HÓA

Ta đã mô tả quy hoạch động như một phương pháp bao hàm những tính toán từ dưới lên của một bảng. Tuy nhiên phương pháp hồi quy lại hướng ta đến việc tính từ trên xuống như phần thiết kế chia để trị trong mục trước. Tại đó ta đã chỉ ra rằng việc thiết lập như vậy dẫn đến độ phức tạp tính toán hàm mũ. Nguyên nhân chính là nhiều thủ tục con được gọi lại nhiều lần. Khi thiết kế thuật toán từ dưới lên thì mỗi thủ tục chỉ thực hiện có một lần. Câu hỏi đặt ra là có khả năng giữ nguyên cấu trúc từ trên xuống của cách giải hồi quy mà vẫn giữ nguyên được độ phức tạp của thuật toán là $O(n^3)$ như phương án từ dưới lên? Câu trả lời được chỉ ra trong phần sau đây.

5.3.1. Ghi nhớ hóa

Trả lời câu hỏi phần trước là ta có thể thực hiện được nhờ kĩ thuật *ghi nhớ hóa.* Ta xem lại thuật toán *recmatrix* ở phần trước. Công việc của thuật toán là tính $m[i, j]$ và trả về giá trị của nó. Như chú ý ở trên là thuật toán cũ luôn luôn phải tính lại giá trị này nhiều lần. Ta phải cố định đại lượng này bằng cách chỉ cho phép tính chúng một lần. Một cách làm là khởi tạo mọi phần tử này *một đại lượng đặc biệt* nào đó (ví dụ như *UNDEFINED*). Khi gặp một đại lượng, khác với đại lượng đặc biệt này thì ta không tính lại nữa, vậy thuật toán được thiết kế lại:

Phiên bản này của thuật toán có độ phức tạp tính toán là $O(n^3)$. Bằng trực quan điều trên là đúng vì mỗi phần tử của bảng chỉ tính có một lần duy nhất và thuộc lớp $O(n^2)$, và những công việc liên quan cần tính một phần tử bảng nhiều nhất là $O(n)$ (nhiều nhất thêm một vòng lặp).

Phương pháp ghi nhớ hóa này rất ít khi được dùng, vì nó chạy chậm hơn những phương pháp thiết kế từ dưới lên. Tuy nhiên một số bài toán quy hoạch động phần nhiều những phần tử bảng là đơn giản và không cần đến, và như vậy tính toán từ dưới lên có thể tính những phần tử này không cần thiết. Trong trường hợp như vậy phương pháp ghi nhớ hóa phát huy tác dụng.

Thuật toán 5.6 Nhân dãy ma trận ghi nhớ (*memomatrix*)

```
function   memomatrix(p[1..n], i, j)
  Đầu vào:  Mảng p[1..n] và hai số nguyên i, j.
  Đầu ra:   Kết quả memomatrix(p, i, j) giá trị nhỏ nhất.
  if m[i, j]! = UNDEFINED then
     return(m[i, j]);{đã xác định rồi}
  else
     if i = j then
        m[i, j] := 0; { trường hợp cơ sở}
     else
        m[i, j] := ∞; {Khởi tạo}
        for k := i to j − 1 do
           cost := recmatrix(p, i, k) + recmatrix(p, k + 1, j) + p[i −
           1]p[k]p[j];
           if cost < m[i, j] then
              m[i, j] := cost; {Thay số lớn hơn}
           end if
        end for
     end if
  end if
  return(m[i, j]);
end   recmatrixchain
```

5.3.2. Đa giác và tam giác hóa

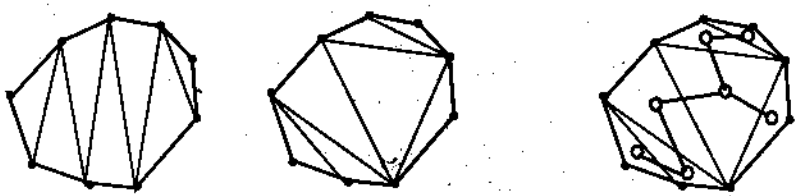
Ta xét một bài toán hình học xem ra rất khác với bài toán nhân dãy ma trận, nhưng thực tế tương tự nhau. Ta bắt đầu bằng một số định nghĩa. *Đa giác* là một đường gấp khúc khép kín những đoạn thẳng trong mặt phẳng. Những đoạn thẳng gọi là *cạnh* của đa giác và những điểm chung của hai đầu đoạn thẳng gọi là *đỉnh* của đa giác. Một đa giác gọi là *đa giác đơn* nếu các cạnh của nó không giao nhau, nghĩa là hai cạnh của nó chỉ có một điểm chung là đỉnh (hình 5.16). Một đa giác đơn chia mặt phẳng ra thành *phần trong, biên* và *phần ngoài*. Một đa giác đơn gọi là *lồi* nếu mọi góc trong của nó không vượt quá 180^0. Những đỉnh có góc trong đúng bằng 180^0 bình thường có thể cho phép, nhưng trong bài toán ta sẽ xét không chấp nhận đó là đỉnh của đa giác.

Đa giác Đa giác đơn Đa giác lồi

Hình 5.16 Những dạng đa giác

Cho một đa giác lồi, giả sử đỉnh của nó được gán nhãn theo chiều ngược chiều kim đồng hồ $P = \{v_0, v_1, ..., v_{n-1}\}$. Ta giả sử chỉ số của đỉnh theo modulo n, nghĩa là $v_n = v_0$. Đa giác như vậy sẽ có n cạnh $\overline{v_{i-1}v_i}$.

Ta lấy hai đỉnh không kề nhau v_i và v_j với $i < j$ thì đoạn thẳng $\overline{v_iv_j}$ là *đường chéo* (nếu đa giác đơn thì đường chéo được đòi hỏi thêm, đoạn nối hai đỉnh này phải hoàn toàn nằm trong miền trong của đa giác đơn). Dễ thấy đường chéo một đa giác lồi chia đa giác thành hai đa giác $\{v_i, v_{i+1}, ..., v_j\}$ và $\{v_j, v_{j+1}, ..., v_i\}$. *Tam giác hóa* một đa giác lồi là một tập cực đại T của những đường chéo không giao nhau chia đa giác thành những tam giác. Như vậy tập những đường chéo này chia đa giác thành những tam giác không có điểm trong chung. Không khó chứng minh rằng mọi phép tam giác hóa một đa giác lồi n cạnh bao gồm $n-3$ đường chéo và $n-2$ tam giác (chứng minh bằng quy nạp toán học).



Một tam giác hóa Giải pháp khác Cây đối ngẫu

Hình 5.17 Phương pháp tam giác hóa

Tam giác hóa được quan tâm do rất nhiều nguyên nhân. Rất nhiều thuật toán hình học được thực hiện bằng cách chia hình đa giác phức tạp thành những tam giác. Khi đó thuật toán có thể áp dụng từ tam giác này đến tam giác khác. Định nghĩa *đồ thị đối ngẫu* của tam giác hóa

là một đồ thị mà đỉnh của nó là những tam giác và trong nó hai đỉnh chung một cạnh nếu hai tam giác có chung một đường chéo (hìn 5.17). Ta thấy rằng đồ thị đối ngẫu là một cây tự do. Do đó thuật toán duyệt các cây có thể dùng cho những tam giác của tam giác hóa.

Một cách tổng quát, cho một đa giác lồi, tồn tại rất nhiều khả năng tam giác hóa. Thực chất đó là hàm mũ số cách tam giác hóa một đa giác lồi theo số cạnh n. Tam giác hoá như nào là tốt nhất? Có rất nhiều tiêu chuẩn để dùng phụ thuộc vào từng ứng dụng cụ thể. Ví dụ như ta phải trả tiền mực cho các đường kẻ trong tam giác hóa đa giác và ta muốn dùng số mực ít nhất. Hoặc bài toán thực tế là có một mảnh kim loại hình đa giác lồi, ta phải cắt thành những hình tam giác sao cho tổng số các đường cắt là nhỏ nhất, ... còn rất nhiều ví dụ khác nữa.

Bài toán : Tam giác hóa đa giác lồi trọng lượng nhỏ nhất

Cho một đa giác lồi xác định một tam giác hóa sao cho tổng của những chu vi tam giác đạt giá trị nhỏ nhất.

Cho ba đỉnh khác nhau v_i, v_j, v_k, ta định nghĩa *trọng lượng* của tam giác tương ứng bằng hàm số trọng lượng

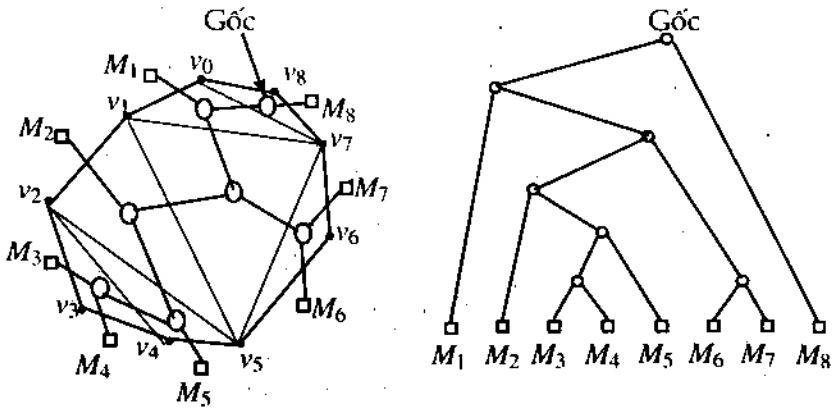
$$w(v_i, v_j, v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|,$$

ở đây $|v_i v_j|$ là độ dài của đoạn thẳng $\overline{v_i v_j}$.

Tương ứng với cây nhị phân

Một sự giải thích đằng sau sự tương tự nhau của tam giác hóa và bài toán nhân dãy các ma trận là cả hai bài toán này liên quan cơ bản tới cây nhị phân. Trong trường hợp nhân dãy ma trận, cây nhị phân tương ứng là cây đánh giá cho phép nhân, ở đây lá của cây tương ứng với những ma trận và mỗi nút của cây tương ứng là phép nhân của dãy hai hoặc nhiều hơn những ma trận. Để thấy sự tương tự ở đây, ta xét đa giác lồi $n+1$ cạnh $P = \{v_0, v_1, ..., v_n\}$ và cố định một cạnh của đa giác lại (cho đó là $\overline{v_0 v_n}$). Bây giờ ta xét gốc của cây nhị phân là nút tam giác chứa cạnh $\overline{v_0 v_n}$, còn những nút bên trong là nút của cây đối ngẫu và những lá của chúng tương ứng với những cạnh của đa giác. Ta thấy rằng chia nhỏ đa giác thành tam giác là tương đương với cây nhị phân n lá, và ngược lại. Ta quan sát hình 5.18 thể hiện điều môt tả trên. Chú

ý rằng mỗi tam giác tương ứng với nút trong của cây và mỗi cạnh của đa giác ban đầu trừ $\overline{v_0v_n}$ tương ứng với nút lá của cây.



Hình 5.18 Tam giác hóa và cấu trúc cây

Những kết luận sau đây suy ra dễ ràng: ứng với những cây có n lá và do đó nó có $n-1$ nút bên trong. Vì mỗi nút bên trong ngoại trừ nút gốc có một cạnh đi vào nó, nên tồn tại $n-2$ cạnh giữa những nút bên trong. Mỗi nút bên trong ứng với một tam giác và mỗi cạnh giữa các nút bên trong tương ứng với một đường chéo của tam giác hóa.

Ta thấy rằng mỗi cạnh cho ứng với mỗi ma trận thì cây trên giống hệt như cây nhân dãy những ma trận. Vậy tiến hành thiết kế cho bài toán tam giác hóa hoàn toàn như bài toán nhân dãy ma trận.

Giải pháp quy hoạch động cho tam giác hóa

Để đưa về công thức gần giống với bài toán nhân dãy ma trận, ta xét đa giác lồi $n+1$ cạnh $P = \{v_0, v_1, ..., v_n\}$. Để đưa về dạng công thức cho quy hoạch động ta cần đi định nghĩa những bài toán con ta có thể có giải pháp tối ưu nhất. Với $1 \le i \le j \le n$, định nghĩa $t[i, j]$ là trọng lượng tối ưu của tam giác hóa cho đa giác con $\{v_{i-1}, v_i, ..., v_j\}$. Giả sử rằng đường chéo $\overline{v_{i-1}v_j}$ nằm trong tập đường chéo tam giác hóa (ta bắt đầu từ $i-1$ vì để sau này công thức cấu trúc giống như trong trường hợp nhân dãy ma trận). Ta thấy rằng nếu ta có thể tính đại lượng $m[i, j]$ với mọi i, j thì bài toán của ta phải tính là $m[1, n]$.

Như là bước cơ sở ta định nghĩa trọng lượng của đa giác hai cạnh

hiển nhiên là 0, kéo theo $t[i,i] = 0$ (đây là đoạn thẳng $\overline{v_{i-1}v_i}$). Một cách tổng quát, tính $m[i,j]$ ta xét đa giác $\{v_{i-1}, v_i, ..., v_j\}$ với $i < j$. Một trong những đường chéo của đa giác này là cạnh $\overline{v_{i-1}v_j}$. Ta có thể chia đa giác con này thành những đa giác con nhỏ hơn nữa bằng cách dựng một tam giác mà cạnh đáy là chính đường chéo này, và đỉnh thứ ba là v_k, ở đây $i \le k \le j-1$. Bằng cách chia này ta chia đa giác thành đa giác con $\{v_{i-1}, v_i, ..., v_k\}$ và $\{v_k, ..., v_j\}$ mà trọng lượng cực tiểu của nó đã biết như $t[i,k]$ và $t[k+1,j]$. Cộng thêm vào đó trọng lượng của tam giác mới tạo ra $v_{i-1}v_kv_j$ cho ta toàn bộ $t[i,j]$. Ta có công thức hồi quy sau:

$$m[i,j] = \begin{cases} 0 & \text{nếu } i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + w(v_{i-1}, v_k, v_j)\} & \text{nếu } i < j. \end{cases}$$

Công thức này hoàn toàn trùng với cấu trúc hồi quy của bài toán nhân dãy ma trận. Ta thực hiện thuật toán giống hệt như với trường hợp nhân dãy ma trận và độ phức tạp tính toán của thủ tục tính tam giác hóa đa giác cũng là $\Theta(n^3)$.

Những bài toán tương tự tam giác hóa đa giác

Ta thấy rằng hai bài toán trên vẻ ngoài hoàn toán khác nhau, nhưng thực chất chúng tương tự nhau. Một nguyên nhân đã nói ở trên là chúng có cấu trúc cây nhị phân tương ứng như nhau. Khi phân tích hai bài toán này đều cho kết quả một số Katalan. Mà một trong những khả năng định nghĩa số Katalan này là số lượng những cách chia (tam giác hóa) đa giác lồi n cạnh ra thành $n-2$ tam giác. Số Katalan có công thức

$$T(n) = \frac{1}{n+1} C_{2n}^{n}.$$

Số lượng tam giác hóa có liên quan đến số lượng của những phương pháp khác nhau nhân dãy n ma trận. Từ bài toán nhân dãy ma trận ta có thể đưa về công thức trên. Để nhân một dãy ma trận ta có thể chia làm hai dãy con tại vị trí k mà $k = 1, 2, ..., n$. Từ đó suy ra số lượng những cách đặt dấu ngoặc cho dãy ma trận được tính theo công thức hồi quy

$$P(n) = \begin{cases} 1 & n = 1, \\ \sum\limits_{k=1}^{n-1} P(k)P(n-k) & n \ge 2. \end{cases}$$

Bằng cách đơn giản hóa ta tìm được mối liên hệ $P(n) = T(n-1)$.

Số Katalan có nhiều cách thể hiện khác nhau. Một số bài toán sau liên quan đến số Katalan và nghĩa là chúng được giải tương tự :

1. Số lượng những cách đặt đúng dấu ngoặc trong tích không kết hợp của $n+1$ thừa số.

2. Số lượng những cách cắt đa giác $n+2$ cạnh thành n tam giác với $n-1$ đường chéo của nó không cắt nhau.

3. Số lượng những phương pháp nhân n ma trận.

4. Thương của hệ số Newton trung bình $\frac{(2n)!}{n!n!}$ và $n+1$.

5. Công thức cho những số hạng của dãy c_j được cho bằng đẳng thức: $c_0 = 1$ và

$$c_{n+1} = c_0c_n + c_1c_{n-1} + \cdots + c_nc_0, n \geq 0.$$

6. Những số hạng của dãy hồi quy được cho bằng đẳng thức $c_0 = 1$ và $(n+1)c_{n+1} = (4n+2)c_n, \quad n \geq 0$.

7. Hệ số trước những lũy thừa của x trong việc khai triển theo hàm sinh $\frac{1-\sqrt{1-4x}}{2x}$.

8. Số lượng những cây nhị phân có n lá ngoài.

9. Số lượng gốc của những cây nhị phân có n lá.

10. Số lượng những con đường khác nhau từ tọa độ $(0,0)$ đến (n,n), nhưng với mỗi bước đi chỉ đúng một trong hai tọa độ x và y được tăng 1 đơn vị và không được phép cắt đường chéo chính cho bởi phương trình $y = x$ tại bất cứ một điểm trong nào.

11. Số lượng những con đường khác nhau từ tọa độ $(0,0)$ đến $(n+1,n+1)$, nhưng với mỗi bước đi chỉ đúng một trong hai tọa độ x và y được tăng 1 đơn vị và không được phép chạm tới đường chéo chính cho bởi phương trình $y = x$ tại bất cứ một điểm trong nào.

12. Số lượng những cách sắp xếp $2n$ người ngồi quanh một bàn tròn, có thể bắt tay nhau mà không có hai đôi nào bị chéo tay nhau.

5.4. DÃY CON CHUNG DÀI NHẤT

Một lĩnh vực rất quan trọng cho thiết kế thuật toán là nghiên cứu các thuật toán về chuỗi kí tự. Có rất nhiều bài toán ở đây, trong số đó quan trọng nhất là tìm một cách hiệu quả một dãy con hoặc tổng quát hơn tìm một mẫu cho trước trong một văn bản (vấn đề này các chương trình soạn thảo văn bản nào cũng phải quan tâm). Trong nhiều trường hợp thực tế ta không phải tìm chính xác mẫu nào đó mà thường đi tìm những mẫu tương tự. Như vấn đề tìm mã gien. Những mã gien được lưu trong một phân tử *DNA* rất dài. *DNA* bao gồm chuỗi rất dài gồm bốn loại kí hiệu cơ bản C, G, T, A. Cách sắp xếp theo chuỗi này trong sinh học thành mẫu giống nhau rất hiếm. Vì vậy người ta chỉ đi tìm những chuỗi kí tự tương tự mà thôi. Một phương pháp chung là đo mức độ tương tự giữa hai chuỗi có chuỗi con chung dài nhất.

Ta quan niệm rằng chuỗi kí tự như một dãy các kí tự. Cho hai dãy $X =< x_1, x_2, ..., x_m >$ và $Y =< y_1, y_2, ..., y_n >$. Ta nói rằng Y là *dãy con* của X nếu tồn tại một dãy tăng thực sự k chỉ số $\{i_1, i_2, ..., i_k\}$ (nghĩa là $1 \leq i_1 < i_2 < \cdots < i_k \leq n$) sao cho $Y =< x_{i_1}, x_{i_2}, ..., x_{i_k} >$.

Ví dụ, cho $X =< abracadabra >$ và $Y =< aadaa >$, khi đó Y là dãy con của X.

Bây giờ cho hai chuỗi X và Y, *dãy con chung dài nhất* (viết tắt là LCS) của X và Y là dãy dài nhất Z mà nó đồng thời là dãy con của X và Y.

Ví dụ, cho X như ở trên và $Y =< yabbadabbadoo >$. Khi đó dãy con chung dài nhất là $Z =< abadaba >$.

Bài toán dãy con chung dài nhất

Cho hai dãy $X =< x_1, x_2, ..., x_m >$ và $Y =< y_1, y_2, ..., y_n >$ hãy tìm dãy con chung dài nhất.

Ta chú ý rằng bài toán không phải có nghiệm duy nhất, ví dụ như LCS của $< abc >$ và $< bac >$ hoặc là $< ac >$ hoặc là $< bc >$. Bài toán có

nhiều nghiệm là những dãy con khác nhau.

Một cách đơn giản để giải bài toán LCS là đem tất cả các dãy con của chuỗi X kiểm tra xem nó có phải là dãy con của Y không và sau đó đánh giá dãy con dài nhất. Làm như vậy thì mỗi dãy con của chuỗi con của X là tương ứng với tập hợp con gồm các chỉ số $\{1,2,...,m\}$. Ta phải tính toán trên 2^m dãy con của X, như vậy giải pháp này đưa đến thời gian thực hiện tìm kiếm là hàm mũ, điều này khó thực hiện được khi dãy đã cho là rất lớn.

Giải pháp quy hoạch động

Như các trình bày ở phần trước ta cần chia bài toán thành những phần nhỏ để giải. Có rất nhiều cách chia một chuỗi thành các chuỗi nhỏ hơn, nhưng ta trở lại bài toán với các cặp *tiền tố* đủ cho chúng ta. *Tiền tố* thứ i của một dãy $X =< x_1, x_2, ..., x_m >$ là một dãy con của X và bao gồm từ phần tử ban đầu đến phần tử thứ i của X. Nghĩa là $X_i =< x_1, ..., x_i >$ với $i = 0, 1, 2, ..., m$. Ta cho X_0 là dãy trống.

Cho $X =< x_1, x_2, ..., x_m >$ và $Y =< y_1, y_2, ..., y_n >$ là các dãy, và $Z =< z_1, z_2, ..., z_k >$ là một LCS bất kỳ của X và Y. Khi đó ý tưởng giải bài toán là tính LCS cho mọi khả năng các cặp của tiền tố hai dãy đã cho. Ta kí hiệu $c[i, j]$ là độ dài của dãy con chung dài nhất của X_i và Y_i. Khi đó thì đại lượng $c[m, n]$ là LCS của toàn bộ hai chuỗi đã cho. Như vậy, ý tưởng là tính $c[i, j]$ mà giả sử ta đã biết trước đó những giá trị $c[i', j']$ với $i' \leq i$ và $j' \leq j$. Ta tiến hành các bước như sau :

1. *Bước cơ sở :* $c[i, 0] = c[0, j] = 0$, nếu một trong những dãy đã cho là rỗng, thì dãy con chung dài nhất cũng rỗng và khi đó chúng có độ dài bằng 0.

2. *Kí tự cuối cùng của tiền tố bằng nhau :* Giả sử $x_i = y_j$. Khi đó phần tử cuối cùng của LCS $z_k = x_i = y_j$ và Z_{k-1} là một LCS của X_{m-1} và Y_{n-1}. Thật vậy, nếu $z_k \neq x_i$, thì ta có thể chắp thêm $x_i \neq y_j$ vào Z để có được dãy con chung của X và Y có chiều dài $k+1$, mâu thuẫn với giả thiết cho rằng Z là dãy con chung dài nhất của X và Y. Như vậy ta phải có $z_k = x_i = y_j$. Vậy thì tiền tố Z_{k-1} là dãy con chung dài nhất của X_{i-1} và Y_{j-1} có độ dài $k-1$.

Như vậy, nếu $x_i = y_j$ thì $c[i,j] = c[i-1, j-1] + 1$.

Ví dụ: $X_i =< abca >$ và $Y_i =< daca >$, hai tiền tố này đều có a cuối cùng, thì dãy con chung dài nhất của chúng phải có a ở cuối. Vì a là một thành phần của LCS ta có thể tìm LCS bằng cách bỏ a ở hai dãy đi và tìm LCS của $X_{i-1} =< abc >$ và $Y_{j-1} =< dac >$ đó là $Z_{k-1} =< ac >$ và khi đó ta thêm a vào cuối được $Z_k =< aca >$ là LCS của hai tiền tố đã cho.

3. *Kí tự cuối cùng của tiền tố không bằng nhau :* Giả sử $x_i \neq y_j$. Khi đó phần tử $z_k \neq x_i$ (hoặc $z_k \neq y_j$) nghĩa là Z là LCS của X_{i-1} và Y_j (hoặc Z là LCS của X_i và Y_{j-1}). Thật vậy, nếu $z_k \neq x_i$, thì Z là dãy con chung của X_{i-1} và Y_j. Nếu có một dãy con chung W khác của X_{i-1} và Y_j có chiều dài lớn hơn k, thì W cũng là dãy con chung của X_i và Y_j, điều này mâu thuẫn với giả thiết Z là LCS của X_i và Y_j (tương tự chứng minh cho trường hợp $z_k \neq y_j$).

Như vậy, x_i và y_j không đồng thời nằm trong dãy con chung dài nhất, nghĩa là hoặc x_i nằm trong LCS, hoặc y_j nằm trong LCS (và có khả năng cả hai đều không nằm trong LCS). Trong trường hợp thứ nhất LCS của X_i và Y_j là LCS của X_{i-1} và Y_j, nghĩa là bằng $c[i-1, j]$. Trong trường hợp thứ hai LCS là LCS của X_i và Y_{j-1} , nghĩa là bằng $c[i, j-1]$. Ta không biết trường hợp nào sẽ xảy ra, nhưng ta có thể lấy giá trị lớn nhất trong chúng.

Như vậy, nếu $x_i \neq y_j$ thì $c[i,j] = \max(c[i-1,j], c[i,j-1])$.

Cuối cùng ta có công thức hồi quy sau:

$$c[i,j] = \begin{cases} 0 & \text{nếu } i = 0 \text{ hoặc } j = 0, \\ c[i-1, j-1] + 1 & \text{nếu } i, j > 0 \text{ và } x_i = y_j, \\ \max(c[i-1,j], c[i,j-1]) & \text{nếu } i, j > 0 \text{ và } x_i \neq y_j. \end{cases}$$

Thuật toán tính dãy con chung dài nhất

Dựa trên công thức truy hồi trên ta có thể viết được thuật toán. Thuật toán tiếp nhận hai chuỗi hữu hạn, mỗi giá trị $c[i,j]$ được ghi vào mảng $c[1..m, 1..n]$. Việc tính toán sẽ được thực hiện theo các hàng của mảng và cột trước đó theo công thức trên. Ta lập mảng thứ hai $b[1..m, 1..n]$ để ghi những giải pháp tối ưu. Thuật toán trả về bảng c và

Thuật toán 5.7 Tìm dãy chung dài nhất (*lengthLCS*)

```
function  lengthLCS(X[1..m], Y[1..n])
  Đầu vào:  Mảng X[1..m] và Y[1..n].
  Đầu ra:   c bảng độ dài LCS của tiền tố và b giải pháp tối ưu.
  Khai báo c[1..m, 1..n] và b[1..m, 1..n];
  m := length(X); n := length(Y);
  for i := 0 to m do
      c[i,0] := 0; b[i,j] :=←;
  end for
  for j := 0 to n do
      c[0,j] := 0; b[i,j] :=↑;
  end for
  for i := 1 to m do
      for j := 1 to n do
          if x_i = y_j then
              c[i,j] := c[i-1,j-1] + 1; b[i,j] :=↖;
          else
              if c[i-1,j] >= c[i,j-1] then
                  c[i,j] := c[i-1,j]; b[i,j] :=↑;
              else
                  c[i,j] := c[i,j-1]; b[i,j] :=←;
              end if
          end if
      end for
  end for
  return(c và b);
end  lengthLCS
```

b, và $c[m,n]$ chứa chiều dài của LCS của X và Y (thuật toán 5.7).

Thời gian thực hiện thuật toán là $O(mn)$ vì tồn tại hai vòng lặp với m và n phép lặp tương ứng. Thuật toán này cũng dùng mất $O(mn)$ không gian.

Thể hiện nghiệm của thuật toán tìm được:

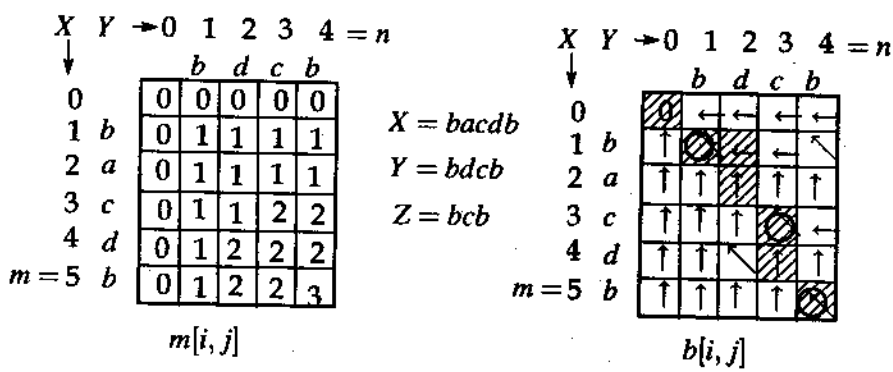
Mảng b của thuật toán (hình 5.19) trả về là nhằm mục đích tạo ra chuỗi con chung dài nhất. Ta bắt đầu từ $b[m,n]$ và ra theo mũi tên. Khi nào gặp ↖ trong mảng b (hình 5.19) nghĩa là $x_i = y_j$ và là một thành

phần của LCS. Phương pháp này gặp các thành phần LCS theo thứ tự đảo ngược. Thủ tục sau đây có thể in ra dãy LCS.

Thuật toán 5.8 In ra dãy con chung dài nhất (*printLCS*)

```
function  printLCS(b, X[1..m], i, j)
  Đầu vào:  Mảng X[1..m] và b.
  Đầu ra:   Mảng LCS của X và Y.
  if i := 0 or j = 0 then
     return(∅);
  end if
  if b[i, j] :=↖ then
     printLCS(b, X, i − 1, j − 1); print x_i;
  else
     if b[i, j] :=↑ then
        printLCS(b, X, i − 1, j);
     else
        printLCS(b, X, i, j − 1);
     end if
  end if
end  printLCS
```

Ví dụ:



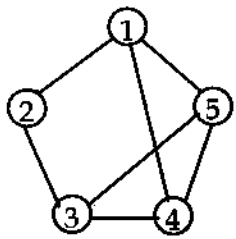
Hình 5.19 Tam giác hóa và cấu trúc cây

5.5. ĐƯỜNG ĐI NGẮN NHẤT

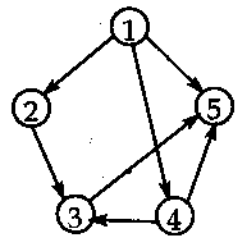
5.5.1. Nhắc lại một số khái niệm trong lí thuyết đồ thị

1. Một *đồ thị* là một cặp thứ tự $G = (V, E)$, ở đây V là một tập hữu hạn *những đỉnh* và $E \subseteq V \times V$ là một tập *những cạnh*. Nghĩa là $V = \{v_1, ..., v_n\}$ thì $E = \{(v_i, v_j) : 1 \le i, j \le n\}$. Ví dụ: (hình 5.20)

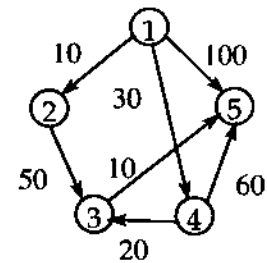
$V = \{1,2,3,4,5\}$ và $E = \{(1,2),(1,4),(1,5),(2,3),(3,4),(3,5),(4,5)\}$



Hình 5.20



Hình 5.21



Hình 5.22

2. *Đồ thị định hướng* là một đồ thị với các hướng trên các cạnh của đồ thị. Ví dụ: (hình 5.21)

$V = \{1,2,3,4,5\}$ và $E = \{(1,2),(1,4),(1,5),(2,3),(3,4),(3,5),(4,5)\}$

3. *Đồ thị định hướng gán nhãn* là một đồ thị định hướng với chi phí dương trên các cạnh của đồ thị. Ví dụ: (hình 5.22)

$V = \{1,2,3,4,5\}$ và $E = \{(1,2),(1,4),(1,5),(2,3),(3,4),(3,5),(4,5)\}$

Ví dụ thực tế như các thành phố là đỉnh và khoảng cách các con đường nối các thành phố là cạnh.

Ta kí hiệu n là số đỉnh và e là số cạnh của đồ thị.

4. *Một đường* trong đồ thị $G = (V, E)$ là một dãy những cạnh liên tiếp

$$(v_1, v_2), (v_2, v_3), ..., (v_n, v_{n+1}) \in E.$$

+ *Độ dài* của đường là số các cạnh.

+ *Chi phí* của đường là tổng những chi phí của các cạnh.

Ví dụ: Đồ thị nhãn định hướng (hình 5.22)

$V = \{1,2,3,4,5\}$, và $E = \{(1,2),(1,4),(1,5),(2,3),(3,4),(3,5),(4,5)\}$

Đường $(1,2),(2,3),(3,5)$: Độ dài đường này là 3 và chi phí là 70.

5.5.2. Đường đi ngắn nhất giữa hai cặp đỉnh

Cho một đồ thị định hướng gán nhãn $G = (V,E)$. Bài toán đặt ra là với mỗi cặp đỉnh $u, v \in V$ tìm chi phí cho đường ngắn nhất đi từ u tới v. Nghĩa là tìm chi phí nhỏ nhất cho con đường đi từ u tới v.

Ta biết rằng có hai thuật toán Dijkstra và Bellman-Ford tìm đường đi ngắn nhất từ một đỉnh đến một đỉnh khác trong đồ thị gán nhãn, những thuật toán này có độ phức tạp tương ứng là $\Theta(n^2)$ và $\Theta(nm)$. Ta có thể lặp lại cách làm của những thuật toán này cho việc tìm đường ngắn nhất giữa hai đỉnh bất kỳ u và v có độ phức tạp tương ứng là $\Theta(n^3)$ và $\Theta(n^2m)$. Ta dùng quy hoạch động để thiết kế một thuật toán cũng có độ phức tạp $\Theta(n^3)$.

Trong đồ thị G ta có các đỉnh $V = \{1,2,...,n\}$ và tập cạnh E với các cạnh có nhãn w. Cho P là đường nối giữa hai đỉnh khác nhau $i, j \in V$ và giả sử k là một đỉnh bên trong của P (nghĩa là đỉnh này khác với i, j). Như vậy nguyên lí cấu trúc thể hiện là có hai đường ngắn nhất: P_1 là đường ngắn nhất đi từ i tới k và P_2 là đường ngắn nhất đi từ k tới j.

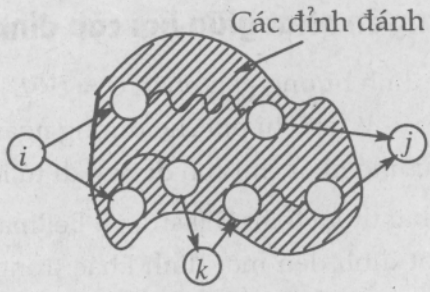
Cho một số nguyên dương k, $k \le n$, và kí hiệu $V_k = \{1,2,...,k\}$ (quy định $V_0 = \emptyset$). Ký hiệu $S_k(i,j)$ là độ dài của con đường ngắn nhất P từ i tới j, mà những đỉnh trong của nó nằm trong tập V_k (nếu không có đường nào như vậy thì $S_k(i,j) = \infty$) :

$$S_k(i,j) = \begin{cases} \text{Chi phí của cạnh từ } i \text{ đến } j, & \text{nếu } i \neq j \text{ và } (i,j) \in E; \\ \infty & \text{nếu } i \neq j \text{ và } (i,j) \notin E; \\ 0 & \text{nếu } i = j. \end{cases}$$

Ta chú ý rằng $S_n(i,j)$ là độ dài của đường ngắn nhất trong G từ i tới j. Nếu k không nằm trong những đỉnh trong của P, thì P là đường ngắn nhất từ i tới j mà những đỉnh trong của nó nằm trong V_{k-1} (hình 5.23), như vậy thì $S_k(i,j) = S_{k-1}(i,j)$.

Trường hợp ngược lại, nếu k nằm trong những đỉnh trong của P, khi đó những đỉnh trong của P_1 từ i tới k và những đỉnh trong của P_2 từ k tới j tất cả đều nằm trong V_{k-1}.

Vì P_1 là đường ngắn nhất từ i đến k và P_2 là đường ngắn nhất từ k đến j, ta có $S_k(i,j) = S_{k-1}(i,k) + S_{k-1}(k,j)$. Như vậy hoặc là $S_k(i,j) = S_{k-1}(i,j)$



Hình 5.23

hoặc là $S_k(i,j) = S_{k-1}(i,k) + S_{k-1}(k,j)$ phụ thuộc vào P có chứa k hay không. Vì P là đường ngắn nhất từ i đến j, suy ra độ dài của P là cực tiểu của hai giá trị này. Vậy ta có công thức truy hồi sau:

$$S_k(i,j) = \min\{S_{k-1}(i,j), S_{k-1}(i,k) + S_{k-1}(k,j)\}. \tag{5.1}$$

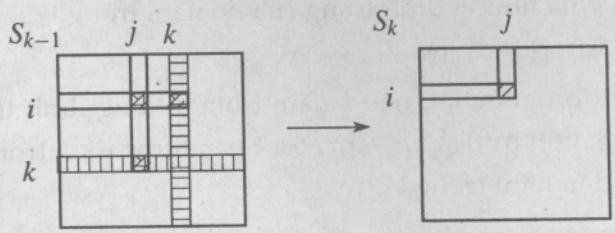
Theo định nghĩa $S_0(i,j)$ là độ dài của đường không chứa một đỉnh ở bên trong nào từ i tới j, nên S_0 bằng *ma trận trọng lượng* W được định nghĩa như sau: $W[i,j] = w(e)$ nếu G chứa cạnh e từ đỉnh i tới j, $W[i,j] = 0$ nếu $i = j$ và $W[i,j] = \infty$ khi không có đường từ đỉnh i tới j.

5.5.3. Thuật toán Floyd

Thuật toán Floyd dựa trên cơ sở công thức (5.1). Ta kí hiệu lại S_k là ma trận chứa độ dài của đường ngắn nhất và đường không ngắn nhất. Do đó

$$S_k[i,j] = \min\{S_{k-1}[i,j], S_{k-1}[i,k] + S_{k-1}[k,j]\}.$$

Như vậy những phần tử của ma trận S_k được tính từ S_{k-1} bằng cột k và hàng k của nó (hình 5.24).



Hình 5.24

Những phần tử của hàng k và cột k của S_k giống hệt như trong S_{k-1}. Thật vậy, tại hàng k ta có

$$\begin{aligned} S_k[k,j] &= \min\{S_{k-1}[k,j], S_{k-1}[k,k] + S_{k-1}[k,j]\} \\ &= \min\{S_{k-1}[k,j], 0 + S_{k-1}[k,j]\} = S_{k-1}[k,j]. \end{aligned}$$

Tại cột k ta có

$$\begin{aligned} S_k[i,k] &= \min\{S_{k-1}[i,k], S_{k-1}[i,k] + S_{k-1}[k,k]\} \\ &= \min\{S_{k-1}[i,k], S_{k-1}[i,k] + 0\} = S_{k-1}[i,k]. \end{aligned}$$

Để lưu lại dấu vết của của đường đi qua các đỉnh ta đưa vào một ma trận nữa $P_k[i,j]$ định nghĩa bằng công thức hồi quy sau:

$$P_k[i,j] = \begin{cases} P_{k-1}[i,j] & \text{nếu } P_k[i,j] = P_{k-1}[i,j]; \\ k & \text{ngược lại} \end{cases}$$

và điều kiện ban đầu là $p[i,j] = 0$ với mọi $i, j \in V$.

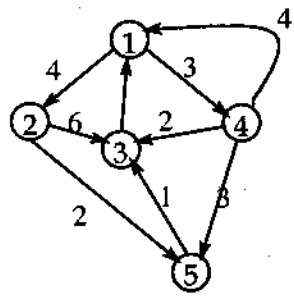
Thuật toán 5.9 Thuật toán Floyd (*Floyd*)

```
procedure   Floyd(W[1..n,1..n], P[1..n,1..n], S[1..n,1..n])
  Đầu vào:  Mảng W[1..n,1..n] ma trận trọng số.
  Đầu ra:   P[1..n,1..n] ma trận lưu đỉnh đường ngắn nhất.
            S[1..n,1..n] lưu khoảng cách giữa hai đỉnh.
  for i := 1 to n do
     for j := 1 to n do
        S[i,j] := W[i,j]; P[i,j] := 0;
     end for
  end for
  for k := 1 to n do
     for i := 1 to n do
        for j := 1 to n do
           if S[i,j] > S[i,k] + S[k,j] then
              S[i,j] := S[i,k] + S[k,j]; P[i,j] := k;
           end if
        end for
     end for
  end for
end Floyd
```

Dễ thấy rằng thuật toán này có độ phức tạp tính toán là $\Theta(n^3)$, vì nhiều nhất có ba vòng lặp, mỗi vòng lặp nhắc lại n lần.

Đường ngắn nhất từ i tới j trong G có thể dựng lại được từ thông tin P_n như sau: nếu $P_n[i, j] = 0$ thì đường ngắn nhất từ i đến j là cạnh trực tiếp nối hai đỉnh i và j; ngược lại nếu $P_n[i, j] = k$, thì k là đỉnh trong đường ngắn nhất đi từ i đến j, còn những đỉnh bên trong khác được tìm trong $P_n[i,k]$ và $P_n[k, j]$.

Ví dụ: Ta lấy đồ thị



$$S_0 = \begin{bmatrix} 0 & 4 & \infty & 3 & \infty \\ \infty & 0 & 6 & \infty & 2 \\ 1 & \infty & 0 & \infty & \infty \\ 4 & \infty & 2 & 0 & 3 \\ \infty & \infty & 1 & \infty & 0 \end{bmatrix}$$

$$S_1 = \begin{bmatrix} 0 & 4 & \infty & 3 & \infty \\ \infty & 0 & 6 & \infty & 2 \\ 1 & 5 & 0 & 4 & \infty \\ 4 & 8 & 2 & 0 & 3 \\ \infty & \infty & 1 & \infty & 0 \end{bmatrix} \qquad P_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$S_2 = \begin{bmatrix} 0 & 4 & 10 & 3 & 6 \\ \infty & 0 & 6 & \infty & 2 \\ 1 & 5 & 0 & 4 & 7 \\ 4 & 8 & 2 & 0 & 3 \\ \infty & \infty & 1 & \infty & 0 \end{bmatrix} \qquad P_2 = \begin{bmatrix} 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 2 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$S_3 = \begin{bmatrix} 0 & 4 & 10 & 3 & 6 \\ 7 & 0 & 6 & 10 & 2 \\ 1 & 5 & 0 & 4 & 7 \\ 4 & 7 & 2 & 0 & 3 \\ 2 & 6 & 1 & 5 & 0 \end{bmatrix} \qquad P_3 = \begin{bmatrix} 0 & 0 & 2 & 0 & 2 \\ 3 & 0 & 0 & 3 & 0 \\ 0 & 1 & 0 & 1 & 2 \\ 3 & 3 & 0 & 0 & 0 \\ 3 & 3 & 0 & 3 & 0 \end{bmatrix}$$

$$S_4 = \begin{bmatrix} 0 & 4 & 5 & 3 & 6 \\ 7 & 0 & 6 & 10 & 2 \\ 1 & 5 & 0 & 4 & 7 \\ 4 & 7 & 2 & 0 & 3 \\ 2 & 6 & 1 & 5 & 0 \end{bmatrix} \qquad P_4 = \begin{bmatrix} 0 & 0 & 4 & 0 & 2 \\ 3 & 0 & 0 & 3 & 0 \\ 0 & 1 & 0 & 1 & 2 \\ 3 & 3 & 0 & 0 & 0 \\ 3 & 3 & 0 & 3 & 0 \end{bmatrix}$$

$$S_5 = \begin{array}{|ccccc|} \hline 0 & 4 & 5 & 3 & 6 \\ 4 & 0 & 3 & 7 & 2 \\ 1 & 5 & 0 & 4 & 7 \\ 4 & 7 & 2 & 0 & 3 \\ \hline 2 & 6 & 1 & 5 & 0 \\ \hline \end{array} \qquad P_5 = \begin{array}{|ccccc|} \hline 0 & 0 & 4 & 0 & 2 \\ 5 & 0 & 5 & 5 & 0 \\ 0 & 1 & 0 & 1 & 2 \\ 3 & 3 & 0 & 0 & 0 \\ 3 & 3 & 0 & 3 & 0 \\ \hline \end{array}$$

Giả sử ta muốn tìm đường ngắn nhất từ đỉnh 3 đến đỉnh 5 ta có thể dựng lại từ P_5 như sau: Ta có $P_5[3,5] = 2$, vậy đỉnh 2 nằm trên đường ngắn nhất. Ta tìm tiếp $P_5[2,5] = 0$, vậy không có đỉnh trong giữa hai đỉnh này mà là đường trực tiếp nối hai đỉnh 2 và 5. Còn $P_5[3,2] = 1$, vậy 1 là đỉnh trong của đường ngắn nhất nối 3 và 2. Ta lại xét $P_5[3,1] = 0$ và $P_5[1,2] = 0$, vậy không có đỉnh giữa 3 và 1 cũng như giữa 1 và 2. Như vậy đường ngắn nhất từ đỉnh 3 đến 5 là bao gồm các đỉnh: 3, 1, 2, 5. Thủ tục in ra tất cả các đỉnh của đường ngắn nhất là :

Thuật toán 5.10 Thuật toán in ra các đỉnh đường ngắn nhất (*shortest*)

procedure *shortest*(i, j)
  Đầu vào: Hai số nguyên $i, j \leq n$.
  Đầu ra: Dãy các đỉnh.
  1: $k := P[i, j]$;
  2: **if** $k > 0$ **then**
  3: *shortest*(i,k); *print*(k); *shortest*(k, j);
  4: **end if**
end *shortest*

Mệnh đề 5.2. *Thủ tục shortest (i, j) được gọi sẽ in ra những nút bên trong của đường ngắn nhất từ i đến j.*

Chứng minh. Chứng minh bằng quy nạp theo độ dài của đường. ☺

5.5.4. Thuật toán Warshall

Bài toán: *Cho một đồ thị định hướng $G = (V, E)$. Với mỗi cặp đỉnh u, v có tồn tại một đường từ u đến v hay không?* (Bài toán gọi là đường đóng kín bắc cầu).

Lời giải: Ta đặt chi phí của tất cả các cạnh bằng 1 và chạy thuật toán floyd. Nếu cuối cùng ta nhận được bảng tại $S[i, j] \neq \infty$, thì tồn tại đường từ đỉnh i đến j.

Do đó ta có thể sửa đổi thuật toán 5.9 với những giá trị trong mảng đúng hoặc sai của mảng khởi đầu W:

Thuật toán 5.11 Thuật toán Warshall (*Warshall*)

```
procedure   Warshall(W[1..n, 1..n], S[1..n, 1..n])
  Đầu vào:   Mảng W[1..n, 1..n] ma trận đúng và sai.
  Đầu ra:    S[1.., 1..n] lưu khoảng cách giữa hai đỉnh.
  for i := 1 to n do
     for j := 1 to n do
        S[i, j] := W[i, j];
     end for
  end for
  for k := 1 to n do
     for i := 1 to n do
        for j := 1 to n do
           S[i, j] := (S[i, j] ∨ (S[i, k] ∧ S[k, j]));
        end for
     end for
  end for
end   Warshall
```

5.6. BÀI TOÁN ĐƯỜNG ĐI CỦA NGƯỜI BÁN HÀNG

Giả sử người bán hàng phải đến n thành phố và có giá chi phí cho đường đi từ thành phố này sang thành phố khác. Người đó xuất phát từ thành phố nhà mình và đi đến $n-1$ thành phố khác còn lại và sau đó trở về nhà theo một thứ tự sao cho chi phí phải trả là ít nhất trong các khả năng đi được.

Ta có thể áp dụng trực tiếp thuật toán *tìm kiếm vét cạn* để đếm tất cả $(n-1)!$ chuyến đi và giữ dấu vết đường đi ngắn nhất. Trong thực tế để giải bài toán đường đi của người bán hàng thì những người này đi đến một thành phố nào đó rồi họ hỏi để biết con đường ngắn nhất đến các thành phố khác. Trong mục này ta dùng phương pháp quy hoạch động để thiết kế một thuật toán cho bài toán có độ phức tạp tính toán $\Theta(n2^n)$ trong trường hợp xấu nhất. Vậy độ phức tạp tốt hơn rất nhiều so với $(n-1)!$, tuy đây là độ phức tạp mũ, nhưng không thể khắc phục

được, vì bài toán đường đi của người bán hàng thuộc lớp những bài toán khó (*NP* khó), ta tin rằng không có thuật toán có độ phức tạp đa thức nào dành cho nó.

Bài toán đường đi người bán hàng tương đương với việc tìm giá nhỏ nhất của chu trình Hamilton trong một đồ thị định hướng trọng lượng G. Ta có thể giả thiết mà không mất tính tổng quát rằng G là một đồ thị định hướng đầy đủ $\hat{K}_n = (V, \hat{E})$, ở đây $|V| = n$ và $\hat{E}$ bao gồm tất cả những cặp đỉnh khác nhau $(u,v), u,v \in V$. Cho một đồ thị định hướng trọng lượng $G = (V,E)$, ta đơn thuần mở rộng trọng lượng c cho $\hat{K}_n$ bằng cách đặt $c(u,v) = \infty$ với mọi cạnh $(u,v) \notin E$. Khi đó bài toán đường đi người bán hàng với đồ thị đầy đủ (không định hướng) K_n với giá trọng lượng c là trường hợp đặc biệt của bài toán đường đi người bán hàng cho đồ thị định hướng $\hat{K}_n$: Với mỗi cạnh e của K_n ta gán đồng thời hai chiều e^- và e^+ của $\hat{K}_n$ tương ứng với e trọng lượng $c(e)$.

Ta xét chu trình Hamilton H của $\hat{K}_n$, không mất tính tổng quát ta giả sử đỉnh 1 khởi đầu và kết thúc của H. Kí hiệu i là đỉnh đầu tiên mà H tới sau khi dời đỉnh 1 và kí hiệu H_i là đường con của H từ đỉnh i tới đỉnh 1 mà nó đi qua mỗi đỉnh của $\hat{K}_n$ một lần (đường như vậy gọi là *đường Hamilton*). Nếu H là chu trình Hamilton giá nhỏ nhất, thì H_i phải là đường ngắn nhất từ i tới 1 mà những đỉnh trong của nó bao gồm tập hợp $V - \{1,i\}$ (một đường ngắn nhất P nối hai đỉnh i và j là đường có cực tiểu $cost(P)$ (ở đây $cost(P)$ là tổng tất cả giá chi phí trên mọi cạnh của P). Bây giờ ta xét tập con bất kì U của V. Với i là đỉnh không thuộc U, ta kí hiệu $mincost(i,U)$ là giá chi phí của đường ngắn nhất (chi phí tối thiểu) P từ i đến 1 mà những đỉnh trong của nó nằm trong tập U. Chú ý rằng vì P là đường ngắn nhất, P phải đi qua mỗi đỉnh của U một lần. Cũng chú ý rằng $mincost(1, V - \{1\})$ là chi phí nhỏ nhất của chu trình Hamilton mà nó chính là đường đi tối ưu của người bán hàng.

Một lời giải của bài toán là tìm đường ngắn nhất $P = iv_1 \ldots v_p 1$ từ đỉnh i đến đỉnh 1, ở đây $v_1, \ldots, v_p \in U$. Nó có thể biểu diễn như dãy những quyết định, ở đây quyết định thứ k là chọn đỉnh v_k như đỉnh thứ $k+1$ trong đường, $k = 1, 2, \ldots, p$. Bài toán nhận được từ kết quả trong quyết

Thuật toán 5.12 Thuật toán đường đi của người bán hàng

```
procedure travelingsaleman(C[1..n, 1..n], TP[1..n, 1..n], cost)
  Đầu vào: Mảng C[1..n, 1..n] ma trận chi phí.
  Đầu ra: TP[1..n, 1..n]
  for i := 2 to n do
     mincost[i, ∅] := C[i, 1];
  end for
  for setsize = 1 to n − 2 do
     for i := 2 to n do
        for mọi U ⊆ V − {i, 1} sao cho |U| = setsize do
           min := ∞;
           for j := 2 to n do
              if j ≠ i then
                 sum := C[i, j] + mincost[j, U − {j}];
                 if sum<min then
                    min := sum; minvertex := j;
                 end if
              end if
           end for
           mincost[i, U] := min;    Path[i, U] := minvertex;
        end for
     end for
  end for
  min := C[1, 2] + mincost[2, V − {1, 2}];
  for j := 3 to n do
     sum := c[1, j] + mincost[j, V − {1, j}];
     if sum<min then
        min := sum; minvertex := j;
     end if
  end for
  Path[1, V − {1}] := minvertex;    cost := min;
  TP[1] := 1;    U := V − {1};
  for i := 1 to n − 1 do
     j := Path(TP[i], U);    TP[i + 1] := j;    U := U − {j};
  end for
end travelingsaleman
```

định đầu tiên (chọn đỉnh v_1) là tìm đường ngắn nhất từ v_1 tới 1, mà

những đỉnh trong của nó gồm tập $V - \{v_1\}$. Dễ kiểm tra thấy rằng nếu P là đường ngắn nhất từ đỉnh i tới j mà những đỉnh trong của nó nằm trong U, thì đường $Q = v_1 v_2 ... v_p 1$ là đường ngắn nhất từ đỉnh v_1 đến đỉnh 1 mà đỉnh trong của nó nằm trong tập $U - \{v_1\}$. Do đó nguyên lí tối ưu của bài toán là đúng. Vậy ta có $cost(P) = c(i, v_1) + cost(Q)$, từ đây ta nhận được quan hệ hồi quy cho $mincost(i, U)$:

$$mincost(i, U) = \min_{j \in U}\{c(i, j) + mincost(j, U - \{j\})\}, \qquad (5.2)$$

điều kiện ban đầu là $mincost(i, \emptyset) = c(i, 1), \quad 2 \le i \le n$.

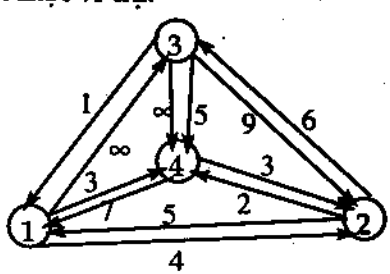
Quan hệ hồi quy (5.2) chính là thuật toán để giải bài toán người bán hàng. Dùng (5.2) để tính $mincost(i, U)$ với tất cả các cặp i, U mà $|U| = 1$. Sau đó lại dùng (5.2) để tính $mincost(i, U)$ với tất cả cặp i, U mà $|U| = 2$. Một cách tổng quát, ta tính $mincost(i, U)$ cho mọi cặp i, U mà $|U| = k - 1$, bằng cách dùng (5.2) để tính $mincost(i, U)$ cho tất cả các cặp mà $|U| = k$, với $k = 1, 2, ..., k - 2$. Khi đó dùng (5.2) để tính $mincost(1, V - \{1\})$ cho ta chi phí nhỏ nhất của chu trình Hamilton. Giả mã của thuật toán sau đây dựa trên cơ sở chiến thuật trên (thuật toán 5.12).

Ta đi phân tích thuật toán này với độ phức tạp xấu nhất $T(n)$. Ta chọn phép toán cơ sở là việc gán vào *mincost*. Với *setsize* $= k$ và i cố định thì số những tập hợp $U \in V - \{1, i\}$ có cỡ k là C_{n-2}^k. Như vậy tổng phép gán vào $mincost[i, U]$ sẽ là

$$\sum_{k=0}^{n-2}(n-1)C_{n-2}^k = (n-1)2^{n-2}.$$

Như vậy, $T(n) = (n-1)2^{n-2} \in \Theta(n2^n)$. Do đó thuật toán *travelingsalesman* có độ phức tạp hàm mũ. Một nhược điểm của thuật toán này là nó cũng có độ phức tạp không gian là $\Theta(n2^n)$.

Ta xét một ví dụ:



$$C = \begin{bmatrix} 0 & 4 & \infty & 3 \\ 5 & 0 & 6 & 2 \\ 1 & 9 & 0 & \infty \\ 7 & 3 & 5 & 0 \end{bmatrix}$$

$|U| = 0$

$mincost[2,\emptyset] = C[2,1] = 5,$

$mincost[3,\emptyset] = C[3,1] = 1,$

$mincost[4,\emptyset] = C[4,1] = 7.$

$|U| = 1$

$mincost[2,\{3\}] = C[2,3] + mincost[3,\emptyset] = 6+1 = 7, Path[2,\{3\}] = 3,$

$mincost[3,\{2\}] = C[3,2] + mincost[2,\emptyset] = 9+5 = 14, Path[3,\{2\}] = 2,$

$mincost[4,\{2\}] = C[4,2] + mincost[2,\emptyset] = 3+5 = 8, Path[4,\{2\}] = 2,$

$mincost[2,\{4\}] = C[2,4] + mincost[4,\emptyset] = 2+7 = 9, Path[2,\{3\}] = 4,$

$mincost[3,\{4\}] = C[3,4] + mincost[4,\emptyset] = \infty+7 = \infty, Path[3,\{2\}] = 4,$

$mincost[4,\{3\}] = C[4,3] + mincost[3,\emptyset] = 5+1 = 6, Path[4,\{2\}] = 3.$

$|U| = 2$

$$mincost[2,\{3,4\}] = \min\{C[2,3] + mincost[3,\{4\}], C[2,4] + mincost[4,\{3\}]\}$$
$$= \min\{6+\infty, 2+6\} = \min\{\infty, 8\} = 8; \quad Path[2,\{3,4\}] = 4,$$
$$mincost[3,\{2,4\}] = \min\{C[3,2] + mincost[2,\{4\}], C[3,4] + mincost[4,\{2\}]\}$$
$$= \min\{9+9, \infty+8\} = \min\{18, \infty\} = 18; \quad Path[3,\{2,4\}] = 2,$$
$$mincost[4,\{2,3\}] = \min\{C[4,2] + mincost[2,\{3\}], C[4,3] + mincost[3,\{2\}]\}$$
$$= \min\{3+7, 5+14\} = \min\{10, 19\} = 10; \quad Path[4,\{2,3\}] = 2.$$

$U = V\{1\}$

$$mincost[1,\{2,3,4\}] = \min\{C[1,2] + mincost[2,\{3,4\}],$$
$$C[1,3] + mincost[3,\{2,4\}], C[1,4] + mincost[4,\{2,3\}]\}$$
$$= \min\{4+8, \infty+18, 3+10\} = \min\{12, \infty, 13\} = 12;$$
$$Path[1,\{2,3,4\}] = 2,$$

$cost = mincost[1,\{2,3,4\}] = 12.$

$TP[1] = 1;$

$TP[2] = Path[1,\{2,3,4\}] = 2;$

$TP[3] = Path[2,\{3,4\}] = 4;$

$TP[4] = Path[4,\{3\}] = 3.$

5.7. BÀI TẬP

▷ **5.1.** Cho n loại đồng xu có giá trị tương ứng $C(1), C(2), ..., C(n)$ và tổng S. Tìm cách biểu diễn tổng S bằng số lượng những đồng ít nhất có thể được. Giá trị của $C(1), C(2), ..., C(n)$ và tổng của chúng S là những số nguyên.

▷ **5.2.** Hãy điền vào bảng giá m trong thuật toán quy hoạch động bài toán nhân các ma trận với những đầu vào sau:

$$n = 5, r_0 = 8, r_1 = 3, r_2 = 2, r_3 = 19, r_4 = 18, r_5 = 7.$$

▷ **5.3.** Hãy tìm một phản ví dụ cho thuật toán sau đây về bài toán nhân ma trận là không tối ưu. Nghĩa là, tìm $n, r_0, r_1, ..., r_n$ sao cho khi nhân các ma trận với các cỡ này được thực hiện như mô tả thuật toán thì hàm giá trị *cost* lớn hơn tối ưu:

Giả sử $n > 2$ và r_i là giá trị nhỏ nhất trong $r_1, r_2, ..., r_{n-1}$. Khi đó tách tích các ma trận sau ma trận M_i và áp dụng thủ tục hồi quy cho hai tích $M_1 \times M_2 \times \cdots \times M_i$ và $M_{i+1} \times M_{i+2} \times \cdots \times M_n$.

▷ **5.4.** Hãy chỉ ra các bước thực hiện của thủ tục *TravelngalesMan* cho ma trận giá trị

$$C = \begin{bmatrix} 0 & \infty & 9 & 3 \\ 5 & 0 & 6 & 2 \\ 9 & 6 & 0 & 7 \\ 7 & 3 & 5 & 0 \end{bmatrix}.$$

CHƯƠNG 6
PHƯƠNG PHÁP THAM

Phần lớn những bài toán trong chương trước là ta tìm nghiệm tối ưu trong tất cả các khả năng nghiệm. Ta nhớ lại với phương pháp quay lui như trong thiết kế các thuật toán để tìm nghiệm tối ưu cần thiết phải tìm tất cả những nghiệm tối ưu của tất cả những bài toán con (nhưng không may một số bài toán con lại phải tính đi, tính lại nhiều lần). Sự tính đi tính lại nhiều lần một bài toán con có thể khắc phục được bằng quy hoạch động, nhưng thật không may quy hoạch động lại liên quan đến việc cần một lượng bộ nhớ đủ lớn cho việc giữ những kết quả tính được. Tư tưởng để khắc phục điều này là: thuật toán tiến đến một trong những trường hợp con của bài toán và giải duy nhất với nó với *hi vọng* rằng nó sẽ cho lời giải đúng. Sự lựa chọn những trường hợp con này được thực hiện hoàn toàn theo tiêu chuẩn chủ quan (cục bộ) cho sự tối ưu. Ví dụ như những *thuật toán tham (lam)* luôn luôn chỉ đến cách chọn tốt nhất tại thời điểm đó như một cách tự nhiên, cho đến những giai đoạn sau mới phát hiện ra rằng sự lựa chọn như vậy là không phù hợp.

Những thuật toán tham được tạo ra rất dễ dàng, tương ứng với việc thể hiện nó trong ngôn ngữ lập trình không phức tạp, nhưng có nhược điểm là đôi khi chúng không đảm bảo nghiệm đúng bài toán ban đầu. Nhược điểm này không làm giảm sự có ích của loại thuật toán này, vì những thuật toán loại này tính toán rất nhanh và tìm được lời giải bài toán, những lời giải này cũng gần tối ưu. Ví dụ trong rất nhiều những bài toán thực tế ta không nghiên cứu được tất cả các trường hợp, đành phải chấp nhận phương án chủ quan do ta đặt ra mà thôi. Thiết lập những thuật toán như vậy đôi khi chỉ có 5% là nghiệm tồi trong những nghiệm tối ưu, so với vô hạn cách tìm nghiệm tối ưu.

Ta trở lại với thuật toán tham và quan sát sự áp dụng nó trong trường hợp cụ thể. Bài toán ta xét là:

Hãy tìm cách nhận được tổng số tiền đã cho S bằng cách dùng số lượng đồng tiền ít nhất với tập hợp những đồng tiền có giá trị $C = \{a_1, a_2, ..., a_n\}$.

Ví dụ đồng tiền Việt Nam có các mệnh giá: 1000 đồng, 2000 đồng, 5000 đồng, 10000 đồng, 20000 đồng, 50000 đồng (thực tế tiền Việt Nam còn nhiều loại nữa, nhưng trong ngân hàng của ta chỉ có sẵn những loại như vậy). Ta xét thuật toán giải bài toán như sau:

1. Khởi đầu cho $s = 0$.

2. Tìm loại đồng tiền với giá trị lớn nhất x ($x \in C$) sao cho $s + x \leq S$.

   (a) Nếu không có loại nào mà $s + x \leq S$ suy ra bài toán không có nghiệm. Thuật toán kết thúc.

   (b) Ngược lại, ta lấy loại tiền x và tăng s lên x (nghĩa là $s = s + x$).

      i. Nếu $s = S$ suy ra bài toán đã được giải. Thuật toán kết thúc.

      ii. Nếu $s < S$, thì ta chưa nhận đủ số tiền bằng tổng S và quay lại bước 2.

Ví dụ: Với tổng số tiền 298000 đồng, thuật toán sẽ lựa chọn như sau:

Năm tờ 50000 đồng, hai tờ 20000 đồng, một tờ 5000 đồng và ba tờ 1000 đồng (tổng cộng $250000 + 40000 + 5000 + 3000 = 298000$ đồng).

Hiển nhiên thuật toán đã viết ở trên phản ánh một tiêu chuẩn thuật toán tham: Tại mỗi bước nó chọn tờ tiền có giá trị lớn nhất, bằng cách này thì khả năng đạt đến tổng số tiền đã cho với khả năng nhanh nhất. Trong trường hợp cụ thể này nó là một giải pháp hiệu quả nhất của bài toán đã ra. Nhưng ta không thể khẳng định thuật toán này làm việc đúng đắn, nếu ta lại chọn các tờ tiền có giá trị khác. Một trong những trường hợp mà trong nó thuật toán trên luôn luôn đúng là dãy những giá trị đồng tiền thỏa mãn điều kiện

$$a_i \geq 2a_{i-1}, \text{ với } i = 2, ..., n, \text{ và } a_1 = 1.$$

Mặt khác, ta xét trường hợp ta chỉ có $C = \{2000, 5000, 20000, 30000\}$ và để nhận số tiền $S = 40000$ đồng. Thuật toán tham trên chọn lựa đầu tiên là 30000 đồng, sau đó sẽ chọn hai tờ 5000 đồng, nghĩa là tổng cộng ba tờ tiền. Nhưng hiển nhiên tồn tại một phương án tối ưu hơn: Hai tờ 20000 đồng. Ngoài chuyện thuật toán tham tìm ra nghiệm không phải tối ưu, mà nhiều khi nói chung nó không tìm ra nghiệm. Ví dụ nếu ta cần tổng số tiền là 6000 trong bốn loại đồng tiền trên: Thuật toán tham ở trên sau khi chọn tờ 5000 đồng rồi, thì thuật toán không còn khả năng lựa chọn nào khác (nhưng thực tế 6000 đồng có thể lấy từ ba tờ 2000 đồng).

Chương này ta xét những bài toán có thể giải bằng phương pháp tham dễ dàng và có hiệu quả.

6.1. LẬP CHƯƠNG TRÌNH CHO NHIỀU HOẠT ĐỘNG

Bài toán hay xảy ra là trên một nguồn cơ sở nào đó ta phải lập chương trình cho sự hoạt động tối ưu trên nguồn cơ sở đó. Ta có tập hợp $S = \{1, 2, ..., n\}$ gồm n những hoạt động, mỗi hoạt động i này phải bắt đầu từ thời gian s_i và kết thúc với thời gian đã cho f_i. Ví dụ, những hoạt động này có thể là những bài giảng được thực hiện trên một giảng đường, ở đây các bài giảng đã được đặt trước hoặc những đòi hỏi cần thiết cho việc thực hiện bài giảng.

6.1.1. Thuật toán tham lập lịch hoạt động

Vì chỉ có một nguồn lực (hội trường) và một số bài giảng có thời gian bắt đầu và kết thúc không chồng lên nhau (hai bài giảng không thể xảy ra trên một hội trường) và tất cả đòi hỏi có thể thỏa mãn được. Ta nói rằng hai hoạt động i và j *tương thích* nếu khoảng thời gian bắt đầu-kết thúc của chúng không chồng lên nhau, nghĩa là $[s_i, f_i) \cap [s_j, f_j) = \emptyset$. *Bài toán lập lịch các hoạt động* là lựa chọn tập hợp lớn nhất những hoạt động tương thích để dùng một nguồn lực đã cho. (Chú ý là có rất nhiều tiêu chuẩn khác ta có thể xem xét thay vào đây, ví dụ như ta có thể xét cực đại thời gian các hoạt động).

Vậy làm thế nào để lập lịch cho số lượng lớn nhất những hoạt động trên một nguồn lực? Bằng trực giác ta không chọn hoạt động có thời gian dài nhất, vì nó sẽ chiếm mất nguồn lực của ta và nó làm ta khó chọn những đòi hỏi khác. Điều này gợi ý cho ta chiến thuật tham sau đây: *Chọn lặp lại những hoạt động với thời gian kéo dài nhỏ nhất* $(f_i - s_i)$ *và lập lịch cho nó với điều kiện là nó tương thích với những hoạt động đã chọn trước đó.* Điều này thành ra chưa chắc đã tối ưu.

Thuật toán tham có thể thực hiện được. Trực giác có thể nói lên điều đó. Vì ta không muốn những hoạt động chiếm nhiều thời gian, ta chọn một hoạt động mà nó kết thúc sớm nhất và lên lịch cho nó. Sau đó giữa tất cả những hoạt động mà chúng tương thích với hoạt động đầu tiên này, ta chọn hoạt động kết thúc sớm nhất và cứ tiếp tục như vậy. Ta giả sử những hoạt động đã được sắp xếp theo thời gian kết thúc, như

$$f_1 \le f_2 \le \cdots \le f_n$$

(tất nhiên s_i cũng được sắp xếp theo, nhưng chưa chắc đã theo thứ tự trên).

Giả mã dưới đây thể hiện ý tưởng trên và giả thiết rằng các hoạt động đã được sắp xếp theo thời gian kết thúc. Đầu ra là danh sách những hoạt động đã được lập bảng. Biến *prev* lưu giữ chỉ số của những hoạt động vừa mới được cho vào lịch tại mọi thời điểm để xác định có tương thích với các hoạt động sau hay không.

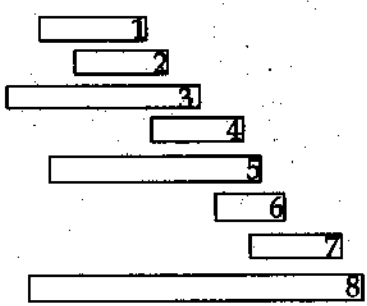
Thuật toán này đơn giản và rất hiệu quả. Thời gian tính toán để

Thuật toán 6.1 Lập lịch hoạt động (schedule)

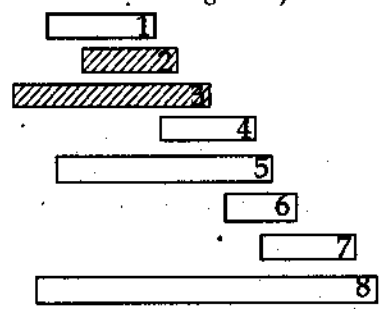
function *schedule*$(n, s[1..n], f[1..n])$
  Đầu vào: Số nguyên dương n, $s[1..n]$ và $f[1..n]$ đã sắp xếp.
  Đầu ra: Mảng $A[1..n]$ danh sách lập lịch.

```
A[1] := 1;    prev := 1;    j := 2;
for i := 2 to n do
    if s[i] >= f[prev] then
        A[j] := i;    j := j + 1;
    end if
end for
return(A)
end schedule
```

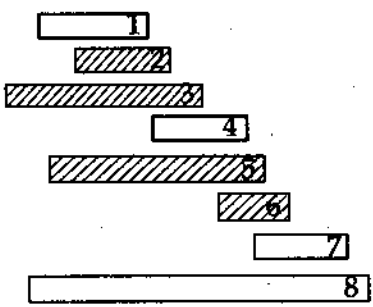
cho thuật toán này hoạt động là sắp xếp thời gian kết thúc của các hoạt động (có thể ta dùng sắp xếp tốt nhất như thủ tục *mergesort*).



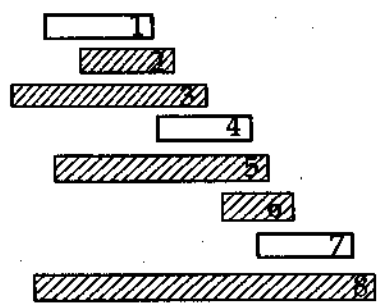
Hình 6.1 Đầu vào



Hình 6.2 Chọn 1



Hình 6.3 Chọn 4



Hình 6.4 Chọn 7

Như vậy tổng thời gian thực hiện thuật toán là $\Theta(n \log n)$.
Các hình (6.1, 6.2, 6.3, 6.4) minh họa hoạt động của thuật toán trên.

6.1.2. Tính đúng đắn của thuật toán

Chứng minh tính đúng đắn dựa trên cơ sở chỉ ra rằng lần chọn đầu tiên theo thuật toán là khả năng tốt nhất và sau đó dùng quy nạp toán học chỉ ra thuật toán là tối ưu toàn cục. Cấu trúc chứng minh sau là đáng ghi nhớ, vì rất nhiều những chứng minh tính đúng đắn của thuật toán tham dựa trên cùng một cơ sở: Chỉ ra rằng mọi nghiệm khác bất kì có thể chuyển thành nghiệm thuật toán tham mà không tăng chi phí của nó.

Mệnh đề 6.1. *Cho $S = \{1, 2, ..., n\}$ là tập những hoạt động để lập lịch đã được sắp xếp theo thời gian kết thúc tăng dần. Khi đó tồn tại một lịch tối ưu trong những hoạt động này mà hoạt động 1 được lên lịch đầu tiên.*

Chứng minh. Cho A là lịch tối ưu. Cho x là một hoạt động trong A với thời gian kết thúc nhỏ nhất. Nếu $x = 1$ thì mệnh đề đúng. Ngược lại, ta tạo ra một lịch mới A' bằng cách thay x bằng hoạt động 1. Ta sẽ chứng minh rằng điều đó có thể làm được (nghĩa là bảng mới này không có những hoạt động tương thích). Vì $A - \{x\}$ không thể có những hoạt động khác mà nó bắt đầu trước khi x kết thúc, và ngược lại những hoạt động này sẽ không tương thích với x. Vì 1 theo định nghĩa là hoạt động đầu tiên kết thúc, nó có thời gian kết thúc sớm hơn x và như vậy nó không thể không tương thích với bất kì hoạt động nào trong $A - \{x\}$. Như vậy A' là một lịch chấp nhận được. Rõ ràng A và A' chứa cùng số những hoạt động, suy ra A' cũng là tối ưu. ☺

Mệnh đề 6.2. *Thuật toán tham ở trên cho nghiệm tối ưu đối với bài toán lập lịch hoạt động.*

Chứng minh. Chứng minh bằng quy nạp theo số những hoạt động.

Bước cơ sở: Nếu không tồn tại hoạt động nào, thì thuật toán tham là hiển nhiên tối ưu.

Bước quy nạp: Giả sử thuật toán tham là tối ưu trên mọi tập những hoạt động có số lượng nhỏ hơn thực sự $|S|$, ta phải chứng minh rằng kết quả còn đúng cho tập những hoạt động có $|S|$. Đặt S' là tập hợp những

hoạt động mà nó tương thích với 1. Nghĩa là $S' = \{i \in S | s_i \geq f_1\}$. Ta thấy rằng một nghiệm bất kì của S' có thể tạo bởi nghiệm của S bằng cách đơn giản là cộng thêm 1 và ngược lại. Vì 1 là một lịch tối ưu (theo mệnh đề trên), suy ra để tạo ra nghiệm tối ưu của cả bài toán, ta phải lập lịch 1 và sau đó gắn thêm lịch tối ưu của S'. Nhưng theo quy hoạch toán học (vì $|S'| < |S|$), đó chính là thuật toán tham đã xây dựng ở trên được thực hiện. ☺

6.2. MÃ HUFFMAN

Những mã Huffman cung cấp cho ta phương pháp mã hóa dữ liệu một cách có hiệu quả. Bình thường khi những kí tự được mã hóa được dùng mã tiêu chuẩn như mã ASCII, mỗi kí tự được thể hiện bằng độ dài cố định *từ mã* của những bit (ví dụ như 8 bit cho một kí tự). Mã độ dài cố định là rất phổ biến, vì nó dễ ngắt một chuỗi thành những kí tự độc lập và xâm nhập tới kí tự đơn lẻ hay những chuỗi kí tự con trực tiếp bằng chỉ số. Tuy nhiên những mã độ dài cố định có thể rất không hiệu quả từ quan điểm tối thiểu hóa tổng số lượng của dữ liệu.

Ta xét ví dụ sau: Giả sử ta muốn mã hóa những chuỗi kí tự có 4 kí tự $C = \{a, b, c, d\}$. Ta có thể dùng mã độ dài cố định sau:

Kí tự	a	b	c	d
Từ mã độ dài cố định	00	01	10	11

Chuỗi kí tự như *abacdaacac* có thể được mã bằng cách thay mọi kí tự bằng tương ứng từ mã nhị phân.

a	b	a	c	d	a	a	c	a	c
00	01	00	10	11	00	00	10	00	10

Cuối cùng là 20 kí tự chuỗi số nhị phân: 00010010110000100010.

Bây giờ ta giả sử biết được những xác suất của những kí tự hay xuất hiện (điều này có thể thực hiện được bằng phân tích các văn bản ta cần mã hóa để tìm ra tần suất xuất hiện của các kí tự này). Bằng những hiểu biết xác suất ta có thể mã hóa những chuỗi kí tự một cách khác nhau. Người ta thường dùng là kí tự nào có tần xuất lớn thì dùng ít bit, còn

những kí tự ít xuất hiện thì dùng nhiều bit. Ví dụ giả sử những kí tự trên có những xác suất sau đây và ta mã hóa:

Kí tự	a	b	c	d
Xác suất	0,60	0,05	0,03	0,05
Từ mã độ dài biến đổi	0	110	10	111

Chú ý là không có đòi hỏi thứ tự chữ cái để áp dụng thứ tự từ mã. Bây giờ chuỗi kí tự trên trở thành:

a	b	a	c	d	a	a	c	a	c
0	110	0	10	111	0	0	10	0	10

Như vậy ta chỉ cần 17 chữ số nhị phân: 01100101110010010. Như vậy ta đã tiết kiệm được ba kí tự bằng cách dùng mã hóa khác. Vậy thì ta sẽ tiết kiệm bao nhiêu, khi một chuỗi kí tự có độ dài n? Với kiểu mã hóa theo mã từ độ dài cố định 2 bit, thì độ dài của chuỗi mã hóa nhị phân là đúng $2n$ bit. Với kiểu mã hóa độ dài mã từ biến đổi thì độ dài được tìm bằng tổng của tích độ dài mã từ với xác suất của chính kí tự này. Độ dài của chuỗi kí tự mã hóa mong muốn là đúng n nhân với độ dài kí tự mã mong muốn:

$$n(0,60.1+0,05.3+0,30.2+0,05.3) = 1,5n.$$

Như vậy ta đã tiết kiệm được 25% về độ dài.

6.2.1. Phân tích và thiết kế thuật toán

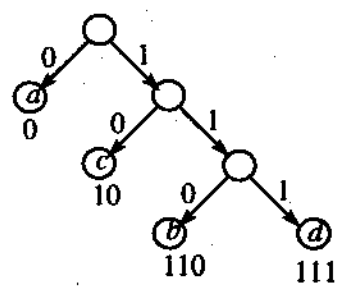
Người ta nhận thấy rằng không thể đưa ra mã hóa một cách tùy tiện. Thật vậy, những mã như trong ví dụ trên được chọn một cách cẩn thận. Giả sử ta chọn mã hoá a như là 0, nhưng ta cũng có thể mã hóa a như là 1, bấy giờ chuỗi mã hóa 111 là rất tối nghĩa, nó có thể là d và nó cũng có thể là aaa. Để tránh điều tối nghĩa này thì làm như thế nào? Ta có thể nhận được lời khuyên là thêm các dấu vào giữa các kí tự mã hóa, nhưng điều này dẫn đến kéo dài thêm mã hóa mà ta không muốn. Ta muốn mã hóa phải có tính duy nhất để giải mã.

Chú ý rằng, các mã từ có độ dài biến đổi trong ví dụ trên, nhưng mã từ này không có *tiền tố* của mã từ khác. Điều này đưa đến tính chất chìa

khóa. Ta thấy rằng nếu hai mã từ có chung một tiền tố, ví dụ $a \to 001$ và $b \to 00101$, thì khi đó ta xem 00101 kí tự đầu tiên được thông báo là a hoặc b. Ngược lại, nếu không có mã từ là tiền tố của bất kì kí tự nào, thì sớm muộn gì ta cũng thấy mã từ xuất hiện như một tiền tố trong văn bản mã hóa, khi đó ta biết rằng ta có thể giải mã được không lo ngại nó phải đối chiếu với mã từ dài hơn nào đó. Như vậy ta có định nghĩa sau:

Mã tiền tố: Một phép gán những mã từ bằng những kí tự sao cho không một mã từ nào là tiền tố của một từ mã bất kì khác.

Ta thấy rằng mã tiền tố nhị phân có thể mô tả bởi cây nhị phân, trong nó những mã từ là những lá của cây, và ở đây nhánh bên trái nghĩa là 0 và nhánh bên phải nghĩa là 1. Mã đã cho trong ví dụ sau cùng ở trên được mô tả trong hình 6.5. Độ dài của mã từ chính là độ sâu của cây này.



Hình 6.5 Những mã tiền tố

Giải mã tiền tố rất đơn giản. Ta đi dọc theo cây từ đỉnh đến lá sẽ biết kí tự đưa vào là kí tự nào. Khi ta tìm đến lá thì ta cho ra kí tự tương ứng và lại trở về gốc để tiếp tục tìm kí tự tiếp theo.

Biết xác suất xuất hiện của những kí tự khác nhau, ta có thể xác định độ dài tổng của văn bản mã hóa. Cho $p(x)$ là xác suất của kí tự x và kí hiệu $d_T(x)$ là độ dài từ khóa (nghĩa là chiều sâu của cây) quan hệ với cây tiền tố T nào đó. Số bit mong muốn để mã hóa văn bản với n kí tự được cho bằng công thức sau

$$B(T) = n \sum_{x \in C} p(x) d_T(x).$$

Điều này gợi ra vấn đề sau đây:

Sinh mã tối ưu: Cho tập chữ cái C và xác suất $p(x)$ xuất hiện của mọi kí tự $x \in C$, hãy tính mã tiền tố T sao cho đạt cực tiểu độ dài chuỗi bit được mã hóa $B(T)$.

Chú ý rằng mã tối ưu không phải duy nhất. Ví dụ, ta có thể làm đầy đủ tất cả những bit trong mã hóa ở các ví dụ trước mà không làm thay đổi độ dài chuỗi kí tự mã hóa mong muốn. Tồn tại một thuật toán đơn giản để tìm một mã hóa này. Nó được phát minh ra vào giữa những năm 1950 thế kỷ trước bởi David Huffman và người ta gọi là *mã Huffman*. Mã này được dùng cho công cụ của Unix, hàm *pack* cho việc nén tệp. (Ngày nay có nhiều phương pháp nén tốt hơn và đây cũng là một chủ đề lớn trong tin học).

Bây giờ ta đi xây dựng thuật toán Huffman. Ta đã biết xác suất của sự xuất hiện các kí tự. Ta phải đi xây dựng cây từ những bậc lá. Ta sẽ lấy hai kí tự x và y trong tập hợp chữ cái và ta trộn chúng vào thành một *siêu kí tự* gọi là z, mà nó thay x và y trong tập chữ cái C. Kí tự z sẽ có xác suất bằng tổng của những xác suất của x và y. Khi đó ta tiếp tục quá trình truy hồi xây dựng mã trên tập chữ cái mới, mà nó có ít hơn một kí tự. Khi quá trình hoàn thành, ta biết được mã cho z, ví dụ như là 010, thì khi đó ta nối vào 0 và 0 cho mã từ này, như 0100 cho x và 0101 cho y.

Ta có thể quan niệm một cách khác quá trình trên như sau: Ta trộn x và y như là con bên phải và con bên trái của nút gốc được gọi là z. Khi đó cây con cho z thay cho x và y trong danh sách các kí tự. Ta lặp lại quá trình này tới khi chỉ còn lại một siêu kí tự. Cây kết quả là cây tiền tố cuối cùng. Vì x và y sẽ xuất hiện tại phần dưới của cây, nên theo lôgic thì ta chọn hai kí tự có xác suất nhỏ nhất để thực hiện thao tác trên. Kết quả được thuật toán Huffman. Giả mã cho thuật toán Huffman được cho bởi thuật toán 6.2 (trang 184).

Cho C là tập hợp kí tự, mỗi kí tự $x \in C$ tương ứng với xác suất xuất hiện là $x.pob$.

1. Khởi đầu tất cả các kí tự được ghi vào một *hàng ưu tiên* Q. (Ta

Thuật toán 6.2 Mã hóa Huffman (Huffman)

```
function   Huffman(n, C[1..n])
  Đầu vào:   C[1..n] kí tự với xác suất.
  Đầu ra:    Phần tử cuối cùng trong Q và là gốc.
  Q := C;
  for i := 1 to n − 1 do
      z := new nút cây bên trong;
      x := Q.extractMin();
      y := Q.extractMin();
      z.left := x;      z.right := y;
      z.prob := x.prob + y.prob;
      Q.insert(z);
  end for
  return Phần tử trái cuối cùng trong Q như gốc.
end   Huffman
```

nhắc lại là cấu trúc dữ liệu này có thể xây dựng trong thời gian $O(n)$ và ta có thể lấy một phần tử với khóa nhỏ nhất trong thời gian $O(\log n)$ và đưa một phần tử mới vào nó trong thời gian $O(\log n)$, ví dụ như cây tìm kiếm nhị phân). Những đối tượng trong Q được sắp xếp theo xác suất của chúng.

2. Chú ý mỗi vòng lặp được thực hiện thì số phần tử trong hàng đợi bị giảm đi một. Như vậy sau $n-1$ vòng lặp, chỉ còn lại đúng một phần tử trong hàng đợi và đây là gốc của cây mã tiền tố cuối cùng.

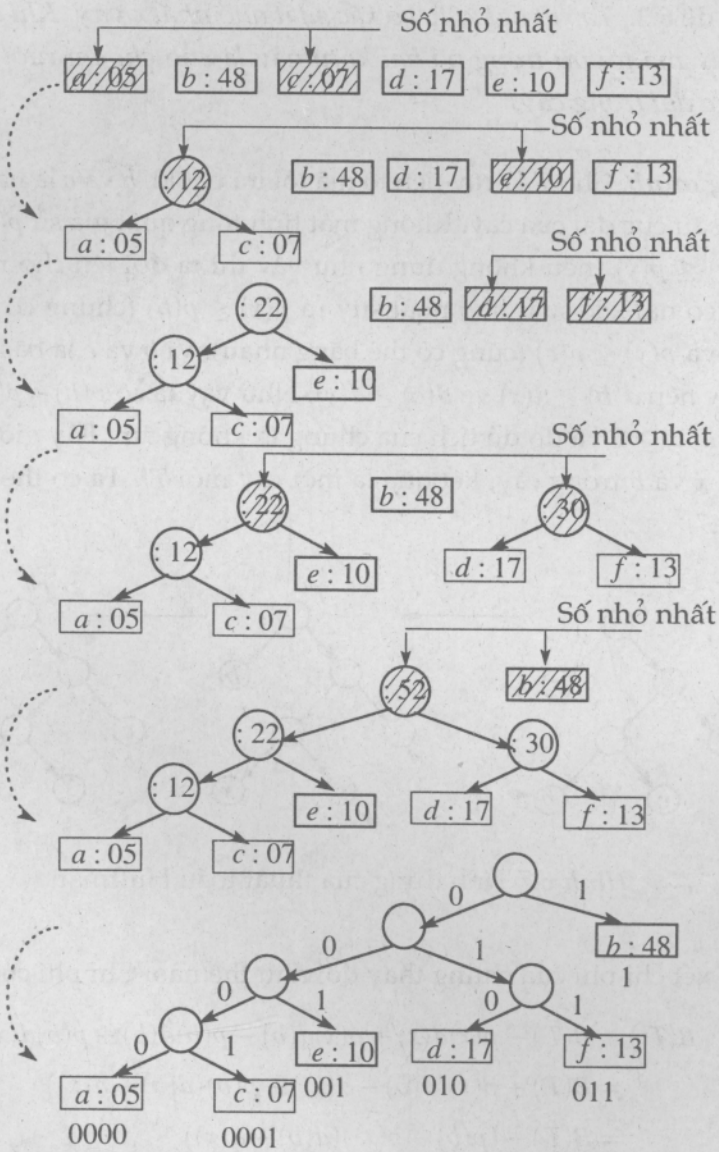
Ví dụ: Cho tập kí tự với xác suất kèm theo

$$C = \{(a:0,05),(b:0,48),(c:0,07),(d:0,17),(e:0,10),(f:0,13)\}$$

để chỉ dùng số nguyên ta nhân mỗi xác suất với 100 và thể hiện như hình 6.6 (trang 185).

6.2.2. Chứng minh tính đúng đắn của thuật toán

Câu hỏi đặt ra là tại sao thuật toán Huffman trên lại đúng? Ta chú ý là chi phí của cây mã hóa bất kì T là $B(T) = n\sum_{x\in C} p(x)d_T(x)$. Ta sẽ chứng minh rằng một cây bất kì khác với cây được xây dựng bằng thuật toán Huffman có thể chuyển đổi thành cây bằng với cây Huffman mà không

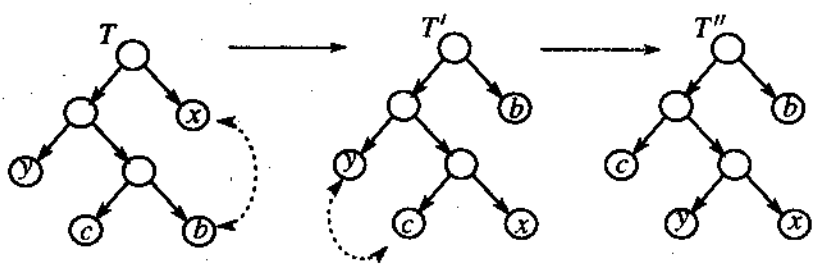


Hình 6.6 Thuật toán Huffman

tăng chi phí của nó. Trước tiên ta thấy rằng cây Huffman là một *cây nhị phân đầy đủ*, nghĩa là mọi nút trong của nó có đúng hai con. Ta không quan tâm tới những nút con bên trong mà chỉ có một con, như vậy ta giới hạn xem xét chỉ trên cây đầy đủ mà thôi.

Mệnh đề 6.3. *Ta xét hai kí tự có xác suất nhỏ nhất x và y. Khi đó tồn tại một cây mã tối ưu trong nó hai kí tự này là anh chị em ruột tại chiều sâu cực đại trong cây.*

Chứng minh. Cho T là cây tiền tố mã tối ưu bất kì, b và c là hai anh em tại độ sâu cực đại của cây. Không mất tính tổng quát giả sử $p(b) \leq p(c)$ và $P(x) \leq p(y)$ (nếu không đúng như vậy thì ta đổi tên cho nhau). Vì x và y có hai xác suất nhỏ nhất suy ra $p(x) \leq p(b)$ (chúng có thể bằng nhau) và $p(y) \leq p(c)$ (cũng có thể bằng nhau). Vì b và c là bậc sâu nhất của cây nên $d(b) \geq d(x)$ và $d(c) \geq d(y)$. Như vậy ta có $p(b) - p(x) \geq 0$ và $d(b) - d(x) \geq 0$ và do đó tích của chúng là không âm. Bây giờ ta đổi vị trí của x và b trong cây, kết quả là một cây mới T'. Ta có thể quan sát hình 6.7.



Hình 6.7 Tính đúng của thuật toán Huffman

Ta xét chi phí của chúng thay đổi như thế nào. Chi phí của T' là :

$$\begin{aligned} B(T') &= B(T) - p(x)d(x) + p(x)d(b) - p(b)d(b) + p(b)d(x) \\ &= B(T) + p(x)(d(b) - d(x)) - p(b)(d(b) - d(x)) \\ &= B(T) - (p(b) - p(x))(d(b) - d(x)) \\ &\leq B(T), \text{ vì } (p(b) - p(x))(d(b) - d(x)) \geq 0. \end{aligned}$$

Như vậy chi phí không tăng lên, suy ra T' là cây tối ưu. Bằng cách chuyển y với c ta nhận được cây T'', tương tự như trên ta suy ra T'' cũng là cây tối ưu. Cuối cùng T'' thỏa mãn mệnh đề đặt ra. ☺

Mệnh đề trên khẳng định rằng bước đầu tiên của thuật toán Huffman là thực hiện đúng. Để chứng minh đầy đủ tính đúng của thuật

toán ta dùng phương pháp quy nạp theo n (vì mỗi lần thực hiện loại đi đúng một kí tự).

Mệnh đề 6.4. *Thuật toán Huffman tạo ra một cây mã tiền tố tối ưu.*

Chứng minh. Chứng minh bằng quy nạp theo n số lượng kí tự.

Bước cơ sở : $n = 1$, cây gồm một nút lá, nó hiển nhiên là tối ưu.

Bước quy nạp : Giả sử với những giá trị thực sự nhỏ hơn n, thuật toán Huffman đảm bảo tạo ra cây tối ưu. Ta chứng minh rằng mệnh đề cũng đúng cho n kí tự. Giả sử ta có n kí tự. Mệnh đề trên phát biểu rằng ta có thể giả thiết trong một cây tối ưu, hai kí tự có xác suất thấp nhất x và y sẽ là anh em cùng lứa tại bậc thấp nhất của cây. Ta bỏ x và y, thay chúng bằng kí tự mới z mà nó có xác suất là $p(z) = p(x) + p(y)$. Như vậy còn lại $n-1$ kí tự. Ta xét cây mã tiền tố bất kì T được làm bằng tập hợp mới gồm $n-1$ kí tự. Ta có thể chuyển nó thành một cây mã tiền tố cho những tập ban đầu của các kí tự với việc thực hiện thao tác ngược lại là thay z bằng x và y (nghĩa là thêm 0 bit cho x và 1 bit cho y). Khi đó chi phí của cây mới sẽ là

$$\begin{aligned}
B(T') &= B(T) - p(z)d(z) + p(x)(d(z)+1) + p(y)(d(z)+1) \\
&= B(T) - (p(x)+p(y))d(z) + (p(x)+p(y))(d(z)+1) \\
&= B(T) + (p(x)+p(y))(d(z)+1-d(z)) \\
&= B(T) + p(x) + p(y).
\end{aligned}$$

Vì sự thay đổi chi phí phụ thuộc vào cấu trúc của cây T dẫn đến cực tiểu chi phí của cây cuối cùng T', ta cần phải xây dựng cây tối ưu T trên $n-1$ kí tự. Theo giả thiết quy nạp, đây chính là thuật toán Huffman thực hiện. Như vậy cây cuối cùng cũng là tối ưu. ☺

6.3. BÀI TOÁN BA LÔ

Bài toán ba lô bao gồm n vật $b_1, b_2, ..., b_n$ và một ba lô có dung lượng là C. Ta giả sử mỗi vật b_i có một trọng lượng dương (hoặc dung lượng dương) w_i và một giá trị v_i (như lợi nhuận, giá trị, ...), $i = 1, ..., n$. Ta đặt những đối tượng hoặc một phần những đối tượng đó vào ba lô càng nhiều càng tốt sao cho không vượt quá dung lượng của ba lô. Bài toán

ba lô được chia ra làm hai loại, một loại người ta gọi là *bài toán* 0/1 *ba lô*. Mỗi vật phải lấy (1) hoặc không lấy (0) nguyên thể vật đó để đưa vào ba lô chứ không chấp nhận chia lẻ thành từng phần. Bài toán tối ưu loại này phát sinh từ công nghiệp ứng dụng đóng gói. Ví dụ người ta muốn chở một tập hợp con những gói hàng hóa trên một toa tầu có dung tích giới hạn. Ngược lại, *bài toán ba lô phân số* cũng như vậy nhưng cho phép phân lẻ những vật, mà không đòi hỏi nguyên thể khi đặt vào ba lô.

Bài toán 0/1 ba lô là một bài toán khó giải và thuộc vào lớp bài toán NP-đầy đủ, nghĩa là có thể không có lời giải tối ưu. Trong khi đó bài toán ba lô phân số thực hiện một cách khá dễ bằng phương pháp tham. Để ngắn gọn trong mục này ta gọi *bài toán ba lô* là bài toán ba lô phân số. Như vậy nếu một phân số f_i của vật b_i được đặt vào ba lô ($0 \le f_i \le 1$), thì được phân bố $f_i v_i$ vào tổng giá trị trong ba lô đã cho. Bài toán ba lô là đặt những vật hoặc một phần của vật vào ba lô mà nó không vượt quá dung tích của ba lô sao cho tổng giá trị của các vật trong ba lô là lớn nhất. Một cách hình thức người ta viết bài toán này như sau:

$$\text{Tìm giá trị cực đại hàm } \sum_{i=1}^{n} f_i v_i, \tag{6.1}$$

với những ràng buộc $\sum_{i=1}^{n} f_i w_i \le C, 0 \le f_i \le 1, i = 1, \ldots, n$. Một số khả năng của phương pháp tham ta có thể lựa chọn để giải bài toán ba lô này. Ví dụ ta có thể đặt những vật có giá trị cao nhất vào ba lô, rồi tiếp đến lại chọn vật có giá trị cao nhất còn lại, ... cho đến khi ba lô chứa hết dung tích. Một chiến lược khác là cũng như trên nhưng ta chọn đặt vào ba lô theo trọng lượng không tăng. Ta có thể dễ tìm thấy ví dụ ở đó những chiến thuật chọn như vậy không tối ưu (không đạt giá trị cực đại). Ta sẽ xét chiến thuật đặt các vật vào ba lô theo sự không giảm của tỉ số $\frac{v_i}{w_i}$ cho tới khi ba lô đầy sẽ mang lại nghiệm tối ưu.

Sự giải thích sau đây về bài toán ba lô làm ta hiểu tại sao tỉ số $\frac{v_i}{w_i}$ lại được đưa ra. Giả sử có một ông chủ cửa hàng "tham" ở Hà Nội có số mặt hàng như bảng 6.1 với số lượng (tính theo tấn) và tổng số tiền tương ứng (tính theo triệu đồng). Ông chủ này có một chiếc ô tô trọng

tải 10 tấn, vậy ông chủ phải xếp lên xe những loại hàng nào để một chuyến đi vào thành phố Hồ Chí Minh, có tổng giá trị hàng lớn nhất?

Loại hàng hóa	**Số lượng** (tấn)	**Tổng tiền** (đồng)
Đậu xanh	8	32
Chè Thái	2	32
Măng khô	4	25
Mứt hoa quả	1	10
Bánh đậu xanh	2	9
Miến	6	18
Nhãn khô	10	55
Gạo nếp	50	100

Bảng 6.1 Kho hàng của ông chủ "tham"

Ta muốn chở được 10 tấn hàng với giá trị cao nhất, bằng trực giác thì ta chọn loại hàng có tỉ số giữa giá tiền và cân nặng lớn nhất với số lượng có thể, sau đó chọn loại hàng thứ hai có tỉ số giữa giá tiền và trọng lượng thứ hai và tiếp tục như vậy cho đến 10 tấn thì thôi (bảng 6.2).

Loại hàng hóa	**Số lượng** (tấn)	**Tổng tiền** (đồng)	**Chọn** (tấn)
Chè Thái	16	1	2
Mứt hoa quả	10	1	1
Măng khô	6.26	1	4
Nhãn khô	5.5	0.3	3
Bánh đậu xanh	4.5	0	0
Đậu xanh	4	0	0
Miến	3	0	0
Gạo nếp	2	0	0

Bảng 6.2 Kho hàng của ông chủ "tham" sau khi chọn

Tổng quát hóa chiến thuật tham, làm sao xếp vào xe số hàng hóa có giá trị lớn nhất. Thuật toán *knapsack* dưới đây giả sử các đối tượng hàng hóa đã được sắp xếp theo chiều không giảm của $\frac{v_i}{w_i}, i = 1,2,...,n$. Thuật toán 6.3 (trang 190) thể hiện ý tưởng trên.

Mệnh đề 6.5. *Thủ tục knapsack sinh ra nghiệm tối ưu khi* $\sum_{i=1}^{n} w_i \leq C$, *từ tất cả những đối tượng được đặt vào trong ba lô.*

Thuật toán 6.3 Ba lô (knapsack)

function *knapsack*$(V[1..n], W[1..n], C, F[1..n])$
Đầu vào: $V[1..n], W[1..n]$ mảng các số nguyên dương, $\frac{V[1]}{W[1]} \geq ... \geq \frac{V[n]}{W[n]}$ và C là số dương.
Đầu ra: $F[1..n]$ mảng số thập phân không âm.

```
for i := 1 to n do
    F[i] := 0;
end for
RemainCap := C;  i := 1;
if W[1] <= C then
    Fits := true;
else
    Fits := false;
end if
while Fits and i <= n do
    F[i] := 1;
    RemainCap := RemainCap - W[i];
    i := i + 1;
    if W[i] <= RemainCap then
        Fits := true;
    else
        Fits := false;
    end if
end while
if i <= n then
    F[i] := RemainCap/W[i];
end if
end knapsack
```

Chứng minh. Ta chứng minh bằng phản chứng. Giả sử *knapsack* sinh ra nghiệm tối ưu khi $\sum_{i=1}^{n} w_i > C$. Giả sử ngược lại có một nghiệm $F = (f_1, f_2, ..., f_n)$ sinh ra bởi thủ tục *knapsack* là nghiệm con tối ưu. Bây giờ ta xét một nghiệm tối ưu $Y = (y_1, y_2, ..., y_n)$, và cho j là chỉ số đầu tiên i sao cho $f_i \neq y_i$, như vậy $1 = f_i = y_i$, $1 \leq i \leq j-1$. Ta chọn Y sao cho j cực đại ở tất cả các nghiệm tối ưu. Chú ý rằng $\sum_{i=1}^{n} y_i w_i = C$ (ngược lại

ta có thể tăng một trong những y_i và hiển nhiên tăng tổng giá trị của nghiệm). Vì chiến thuật tham dùng trong *knapsack* nên ta có $f_j > y_j$.

Bằng cách chuyển đổi Y, ta xây dựng nghiệm tối ưu $Z = (z_1, z_2, ..., z_n)$ mà nó tương thích với F khác với lúc ban đầu. Ta đặt $z_i = f_i, 1 \leq i \leq j$, và với $i > j$ giá trị của z_i nhận được từ giảm một cách thích hợp y_i sao cho $\sum_{i=1}^{n} z_i w_i = C$. Suy ra

$$(z_j - y_j)w_j = \sum_{i=j+1}^{n} (y_i - y_i)w_i.$$

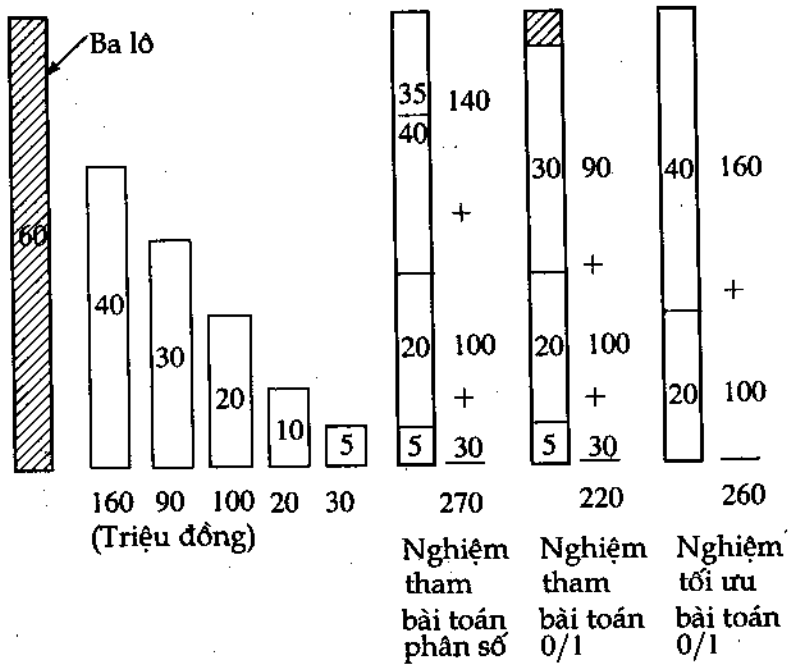
So sánh tổng giá trị của nghiệm Z và Y và dùng đánh giá $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq ... \geq \frac{v_n}{w_n}$ ta nhận được :

$$\begin{aligned}\sum_{i=1}^{n} z_i v_i - \sum_{i=1}^{n} y_i w_i &= (z_j - y_j)v_j - \sum_{i=j+1}^{n} (y_i - z_i)v_i \\ &= (z_j - y_j)w_j \frac{v_j}{w_j} - \sum_{i=j+1}^{n} (y_i - z_i)w_i \frac{v_i}{w_i} \\ &\geq \left[(z_j - y_j)w_j + \sum_{i=j+1}^{n} (y_i - z_i)w_i\right] \frac{v_j}{w_j} = 0\end{aligned}$$

Như vậy giá trị của Z không nhỏ hơn Y, vậy Z cũng là một nghiệm tối ưu. Nhưng Z tương thích với F tại vị trí đầu tiên j, điều này trái với giả thiết Y là nghiệm tối ưu mà nó tương thích với F hầu như các vị trí ban đầu.

Dễ thấy thuật toán *knapsack* có độ phức tạp tuyến tính trong trường hợp xấu nhất. Tuy nhiên do giả thiết thuật toán này phải sắp xếp đối tượng theo $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq ... \geq \frac{v_n}{w_n}$. Ta có thể áp dụng một loại sắp xếp nhanh nhất có độ phức tạp $O(n \log n)$. Như vậy nói chung thuật toán này có độ phức tạp $O(n \log n)$ trong trường hợp xấu nhất. ☺

Ta có thể lấy ví dụ thuật toán tham không cho nghiệm tối ưu với bài toán 0/1 ba lô . Ví dụ được thể hiện trong hình 6.8 (trang 192): Ba lô ở đây là xe tải chở được 60 (tấn), số lượng hàng tính theo (tấn) và tổng giá trị được tính theo (triệu) đồng.



Hình 6.8 Bài toán ba lô phân số và bài toán 0/1 ba lô.

6.4. BÀI TOÁN CÂY BAO TRÙM NHỎ NHẤT VÀ THUẬT TOÁN

6.4.1. Cây bao trùm nhỏ nhất

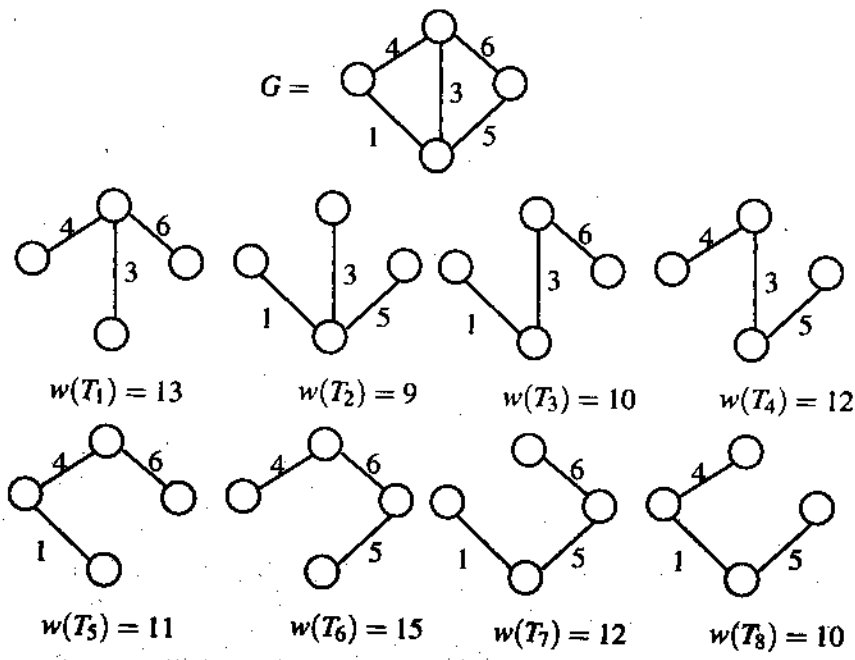
Bài toán chung cho các mạng truyền thông và thiết kế vi mạch điện tử là nối các điểm nốt với nhau (mạng truyền thông là các trạm truyền dẫn hoặc các phần tử vi mạch) bởi mạng có tổng độ dài nhỏ nhất (tổng độ dài các dây dẫn). Ta giả sử mạng không định hướng. Để tối thiểu hóa độ dài của mạng thì nó không bao giờ có chu trình khép kín (vì nếu có chu trình khép kín thì ta ngắt nó ra và khi đó độ dài sẽ ngắn hơn). Vì thế, mạng ta coi là đồ thị liên thông, không định hướng và không chu trình khép kín, đồ thị như vậy gọi là *cây tự do*.

Một cách hình thức hóa ta xét đồ thị liên thông, không định hướng $G = (V,E)$, V là tập đỉnh, E là tập các cạnh. *Cây bao trùm* là một tập con không chu trình khép kín các cạnh $T \subset E$ mà chúng nối tất cả các đỉnh của đồ thị với nhau.

Giả sử mỗi cạnh (u,v) của G có trọng lượng $w(u,v)$ (có thể là số 0 hoặc một số âm). Ta định nghĩa *trọng lượng* của cây bao trùm T là tổng trọng lượng của các cạnh của cây bao trùm này:

$$w(T) = \sum_{(u,v)\in T} w(u,v).$$

Một cây bao trùm nhỏ nhất (viết tắt là MST) là cây bao trùm có trọng lượng nhỏ nhất.



Hình 6.9 Số cây bao trùm của G và cây bao trùm nhỏ nhất là T_2

Chú ý rằng cây bao trùm nhỏ nhất không phải duy nhất, nhưng đúng là nếu tất cả các cạnh có trọng lượng khác nhau thì MST sẽ khác nhau. Ví dụ cây T_2 có trọng lượng nhỏ nhất (hình 6.9).

6.4.2. Tiếp cận thuật toán tìm cây bao trùm nhỏ nhất

Ta sẽ trình bày thuật toán tham cho việc tính cây bao trùm nhỏ nhất. Thuật toán tham được xây dựng trên cơ sở lặp lại lựa chọn trọng lượng nhỏ nhất giữa tất cả khả năng lựa chọn tại mỗi bước (có tính chất địa

phương hoặc chủ quan). Đặc trưng quan trọng của thuật toán tham là luôn luôn chọn được, không bao giờ không thể chọn. Để lập được thuật toán ta nhắc lại một số tính chất của cây tự do. Bổ đề sau đây dễ dàng được chứng minh.

Mệnh đề 6.6. *Đối với một cây tự do thì*

- *Cây tự do có n đỉnh thì có đúng $n-1$ cạnh.*
- *Tồn tại duy nhất một đường đi giữa hai đỉnh bất kì của cây tự do.*
- *Lập thêm một cạnh bất kì của cây tự do tạo ra một vòng tròn duy nhất. Ngắt cạnh bất kì của đường tròn vừa lập, trả đồ thị về cây tự do.*

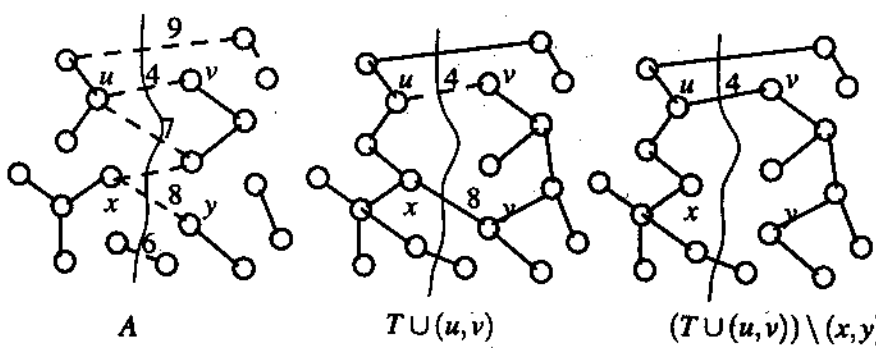
Cho $G = (V, E)$ là đồ thị liên thông, không định hướng và các cạnh của nó có trọng lượng. Trực giác đằng sau thuật toán tham tìm cây bao trùm nhỏ nhất là thực hiện trên tập con A các cạnh, mà khởi đầu nó bằng trống và sau đó ta cho thêm vào mỗi cạnh một lần cho đến khi A bằng MST. Ta nói rằng một tập con $A \subset E$ là *có khả năng* nếu A tập con các cạnh của cây bao trùm nhỏ nhất nào đó. Ta nói rằng một cạnh $(u, v) \in E \setminus A$ là *chắc chắn* nếu $A \cup \{(u, v)\}$ có khả năng. Nói cách khác (u, v) chắc chắn chọn thêm vào sao cho A có thể mở rộng đến cây bao trùm nhỏ nhất. Chú ý là nếu A có khả năng thì không thể có chu trình khép kín. Thuật toán tham là thực hiện lặp lại thêm các cạnh chắc chắn vào cây bao trùm hiện thời.

Khi nào một cạnh là chắc chắn? Ta xét về mặt lí thuyết, khi nào một cạnh là chắc chắn hoặc không. Cho S là tập hợp con các đỉnh $S \subseteq V$. Kí hiệu $cut(S, V \setminus S)$ là *phần cắt* phân chia các đỉnh của đồ thị thành hai tập không giao nhau. Một cạnh (u, v) băng qua phần cắt nếu một đỉnh thuộc S và đỉnh kia thuộc phần còn lại $V \setminus S$. Cho tập hợp con A các cạnh, ta nói rằng phần cắt tương ứng A nếu không có cạnh trong A đi qua phần cắt. Không khó thấy rằng phần cắt tương ứng là quan trọng với bài toán này. Nếu ta phải tính bộ phận cây bao trùm nhỏ nhất và ta muốn biết cạnh nào có thể được thêm vào mà không tạo ra vòng tròn trong cây hiện thời MST, bất cứ một cạnh nào đi qua phần cắt tương ứng đều có khả năng lựa chọn.

Một cạnh của E là *cạnh nhẹ* đi qua phần cắt, nếu giữa tất cả các cạnh qua phần cắt thì nó có trọng lượng nhỏ nhất (đường nhẹ nhất này có thể không là duy nhất). Trực giác thấy rằng từ tất cả các cạnh mà nó đi qua phần cắt không tạo với các cạnh trong phần cắt tương ứng tạo ra một vòng tròn khép kín, đều có thể là đường nhẹ. Thuật toán sẽ thiết kế đều dùng ý tưởng thêm vào cạnh có trọng lượng nhỏ nhất đi qua phần cắt.

Mệnh đề 6.7. *Cho $G = (V,E)$ liên thông, không định hướng với các trọng lượng thực trên các cạnh. Cho A là tập con có thể được của E (nghĩa là tập con của một cây bao trùm nhỏ nhất nào đó), cho $(S, V \setminus S)$ là phần cắt mà tương ứng với A và (u,v) là cạnh nhẹ nhất đi qua phần cắt này. Khi đó cạnh (u,v) là một cạnh chắc chắn cho A.*

Chứng minh. Để đơn giản chứng minh ta giả thiết trọng lượng trên mỗi cạnh là khác nhau. Cho T là cây bao trùm nhỏ nhất cho G. Nếu T chứa (u,v) thì mệnh đề đã được chứng minh. Giả sử nó không có một MST nào chứa (x,y). Ta sẽ chứng minh bằng phản chứng.



Hình 6.10 Cây bao trùm nhỏ nhất theo bổ đề

Cộng thêm cạnh (u,v) vào T, như vậy tạo ra vòng tròn. Vì u và v là các đỉnh đối diện của phần cắt, mà vòng tròn bất kì phải đi qua phần cắt số chẵn lần, do đó tồn tại ít nhất một cạnh (x,y) trong T đi qua phần cắt này.

Cạnh (x,y) không nằm trong A (vì phần cắt tương ứng A). Bằng cách

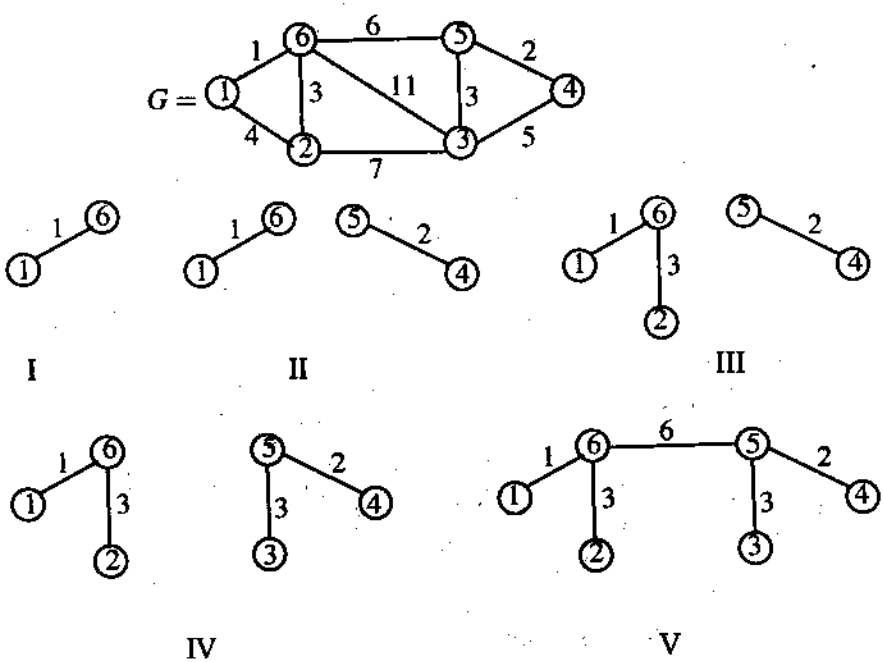
bỏ đi (x,y) đưa T trở về cây bao trùm T'. Ta có

$$w(T') = w(T) - w(x,y) + w(u,v).$$

Vì (u,v) là cạnh nhẹ nhất đi qua phần cắt, ta có $w(u,v) < w(x,y)$. Do đó $w(T') < w(T)$. Điều này vô lí với T là cây bao trùm nhỏ nhất. ☺

6.5. THUẬT TOÁN KRUSKAL

Thuật toán Kruskal thực hiện bằng cách cố gắng thêm các cạnh vào A với trọng lượng tăng lên theo thứ tự. Nếu cạnh tiếp theo không tạo ra chu trình khép kín giữa tập các cạnh hiện thời, thì nó được thêm vào A. Nếu nó tạo ra chu trình khép kín thì cạnh này được bỏ qua và ta xét cạnh tiếp theo với thứ tự trọng lượng.



Hình 6.11 Thuật toán Kruskal

Chú ý rằng khi thuật toán này chạy, các cạnh của A sẽ tạo ra một rừng các đỉnh. Bằng cách tiếp tục thuật toán, các cây của rừng này được hợp với nhau cho tới khi ta có một cây duy nhất chứa tất cả các đỉnh. Ví dụ thuật toán Kruskal cho cây như hình 6.11.

Ta có thể thấy thuật toán trên là đúng đắn. Ta xét cạnh (u,v) mà thuật toán Kruskal tiến tới thêm vào và giả sử cạnh này không tạo ra chu trình khép kín trong A. Kí hiệu A' cây của rừng A chứa đỉnh u. Ta xét phần cắt $(A', V \setminus A')$. Mỗi cạnh đi qua phần cắt không thuộc vào A và do đó nó thuộc phần tương ứng A và (u,v) là cạnh nhẹ nhất đi qua phần cắt. Như vậy theo mệnh đề trên (u,v) là có thể chọn.

Thuật toán được thể hiện với các hàm theo cấu trúc dữ liệu:

$Create-set(u)$: Tạo ra một tập hợp một phần tử u.

$Find-set(u)$: Tìm tập mà nó chứa phần tử đã cho u.

$Union(u,v)$: Trộn tập chứa u và tệp chứa v thành một tập chung.

Trong thuật toán Kruskal các đỉnh của đồ thị sẽ là các phần tử được lưu vào trong các tập hợp và các tập hợp này sẽ là các đỉnh trong mỗi cây của A. Tập hợp A được lưu lại như một danh sách duy nhất các cạnh. Thuật toán thể hiện giả mã như sau:

Thuật toán 6.4 Thuật toán tìm cây bao trùm Kruskal

function *kruskal*(G,w)
  Đầu vào: G (đồ thị có tập n đỉnh V và tập cạnh $E = \{e_i = (u_i, v_i) | i = 1,2,...,m\}$); w hàm trọng lượng của E.
  Đầu ra: A cây bao trùm nhỏ nhất.
  $A = \emptyset$.
  for (Mỗi u thuộc V) **do**
    $Create-set(u)$;
  end for
  Sắp xếp E theo thứ tự tăng của trọng lượng w;
  for Mỗi (u,v) trong danh sách đã sắp xếp **do**
    if $(Find-set(u) not = Find-set(v))$ **then**
      Thêm (u,v) vào A;
      $Union(u,v)$;
    end if
  end for
  return$A$
end *kruskal*

Thời gian tính toán của thuật toán Kruskal là bao nhiêu? Kí hiệu $|V|$ là số các đỉnh và $|E|$ là số các cạnh. Do đồ thị là liên thông, ta giả thiết

rằng $|E| \geq |V| - 1$. Ta biết rằng thuật toán sắp xếp các cạnh có hàm thời gian là $\Theta(|E| \log |E|)$. Vòng lặp **for** lặp $|E|$ lần và mỗi lần lặp lại số hợp các phần tử *Union* và so sánh *Find – set*. Mỗi lần thao tác trên tập hợp là $\Theta(V)$, tổng thể là $\Theta(|E| \log |V|)$. Như vậy tổng thời gian thực hiện là tổng các số như vậy nên đó là $\Theta((|V| + |E|) \log |V|)$. Vì $|V|$ là tiệm cận không lớn hơn $|E|$, kết quả trên viết đơn giản là $\Theta(|E| \log |V|)$.

CHƯƠNG 7
THUẬT TOÁN QUAY LUI

Thường thường khi ta giải bài toán, nhiều khi không có cách nào tốt hơn là thử tất cả các khả năng của nghiệm bài toán có thể xảy ra. Cách tiếp cận như vậy người ta gọi là *tìm kiếm toàn diện*, như vậy việc tìm lời giải rất chậm, nhưng nhiều khi có còn hơn không có cách giải nào! Đặc biệt khi cỡ bài toán đủ nhỏ để ta có thể kiên trì giải được. Kỹ thuật thuật toán để giải theo kiểu này là sử dụng thuật toán chia để trị thích hợp nhất.

Rất nhiều bài toán tìm kiếm toàn diện có thể giảm đáng kể độ phức tạp đến bằng bài toán sinh ra những đối tượng tổ hợp đơn giản, thường thường nhất là sinh ra chuỗi, sinh hoán vị và sinh ra tổ hợp. Vì vậy ta nghiên cứu trong chương này những thuật toán công cụ cơ bản: Những thuật toán cho việc sinh ra những đối tượng cơ bản như :

- Sinh ra 2^n chuỗi nhị phân có độ dài n bit.

- Sinh ra k^n chuỗi k bit từ chuỗi có độ dài n.
- Sinh ra $n!$ hoán vị từ n đối tượng.
- Sinh ra $\frac{n!}{r!(n-r)!}$ tổ hợp từ n vật chọn lấy r vật một lần.

Kĩ thuật tỉa nhánh trong tìm kiếm toàn diện cho ta một phương pháp quay lui nhanh chóng và có hiệu quả.

7.1. THUẬT TOÁN QUAY LUI VỚI CHUỖI NHỊ PHÂN

Trong phần này ta nghiên cứu nguyên tắc chung về thuật toán quay lui với chuỗi nhị phân, với k-chuỗi, kĩ thuật tỉa đi những nhánh không cần thiết khi tìm lời giải và học cách viết một thuật toán quay lui như thế nào. Những cơ sở trên được áp dụng cho một dạng bài toán ba lô, bài toán chu trình Hamilton, bài toán đường đi của người bán hàng.

7.1.1. Những chuỗi bit

1. Bài toán

Lập thuật toán sinh ra tất cả các chuỗi nhị phân khác nhau gồm n bit.

Lời giải: Ta dùng cách chia để trị. Chuỗi nhị phân nên bit cuối cùng chỉ có 0 hoặc 1, vậy ta có thể xét tất cả các chuỗi chỉ có 0 sau cùng hoặc chỉ có 1. Vậy để tạo ra ta lần lượt tạo ra chuỗi 0 ở cuối trước sau đó là chuỗi có 1 ở sau bằng cách hồi quy. Ta chứa chuỗi nhị phân trong mảng $A[1..n]$. Mục đích là ta gọi một thủ tục sinh ra A một lần (gọi đó là *process*(A)) với $A[1..n]$ có chứa mỗi chuỗi nhị phân. Thuật toán 7.1 (trang 201) thể hiện ý tưởng lập ra các dãy này.

2. Chứng minh tính đúng đắn của thuật toán 7.1

Mệnh đề 7.1. *Với mọi $m \geq 1$, thủ tục binary(m) gọi process(A) một lần với $A[1..m]$ chứa mọi chuỗi m bit.*

Chứng minh. Ta chứng minh quy nạp theo m. Mệnh đề đúng với $m = 0$.

Bây giờ ta giả sử thủ tục *binary*($m-1$) gọi *process*(A) một lần với $A[1..m]$ chứa mỗi chuỗi $m-1$ bit

Khi đó, thủ tục *binary*(m) thực hiện

Thuật toán 7.1 Xử lí tất cả chuỗi nhị phân có độ dài m

```
procedure  binary(m)
  Đầu vào:  Giá trị m.
  Đầu ra:   Sinh ra các chuỗi nhị phân có bit khác nhau.
  if m = 0 then
     process(A);
  else
     A[m] := 0; binary(m − 1);
     A[m] := 1; binary(m − 1);
  end if
end  binary
```

- Gán $A[m]$ bằng 0, và
- Gọi $binary(m-1)$.

Theo giả thiết quy nạp, lời gọi thủ tục $binary(m-1)$ là gọi $process(A)$ một lần với $A[1..m]$ chứa mỗi chuỗi m bit, có bit cuối cùng bằng 0.

Sau đó, thủ tục $binary(m)$ thực hiện

- Gán $A[m]$ bằng 1;
- Gọi $binary(m-1)$.

Theo giả thiết quy nạp, lời gọi thủ tục $binary(m-1)$ là gọi $process(A)$ một lần với $A[1..m]$ chứa mỗi chuỗi m bit, có bit cuối cùng bằng 1.

Do đó, $binary(m)$ là gọi $process(A)$ một lần với $A[1..m]$ chứa mỗi chuỗi m bit. ☺

3. Phân tích thuật toán

Kí hiệu $T(n)$ là thời gian chạy của thuật toán $binary(m)$. Giả sử thủ tục xử lí chỉ tốn thời gian $O(1)$. Khi đó

$$T(n) = \begin{cases} c & \text{nếu } n = 1 \\ 2T(n-1)+d & \text{ngược lại}. \end{cases}$$

Do đó bằng cách thay liên tiếp vào đẳng thức trên ta có

$$T(n) = (c+d)2^{n-1} - d.$$

Suy ra $T(n) = O(2^n)$, nghĩa là thuật toán sinh chuỗi bit là tối ưu.

4. k-chuỗi

Bài toán : *Lập thuật toán sinh ra tất cả các chuỗi của n số mà chỉ điền các số từ* 0 *đến* $k-1$.

Lời giải : Ta chứa k-chuỗi trong mảng $A[1..n]$. Mục đích của ta là gọi một thủ tục sinh ra *process*(A) một lần với $A[1..n]$ chứa k-chuỗi. Gọi thủ tục *string*(n) (thuật toán 7.2):

Thuật toán 7.2 Sinh tất cả k-chuỗi có độ dài m

```
procedure   string(m)
  Đầu vào:  Giá trị m.
  Đầu ra:   Sinh ra các chuỗi nhị phân tất cả k-chuỗi có độ
            dài m.
  if m = 0 then
     process(A);
  else
     for j := 0 to k − 1 do
        A[m] := j;   string(m − 1);
     end for
  end if
end   string
```

Chứng minh tính đúng : Tương tự trường hợp nhị phân.

Phân tích : Tương tự như trong trường hợp nhị phân.

5. Những nguyên lý quay lui

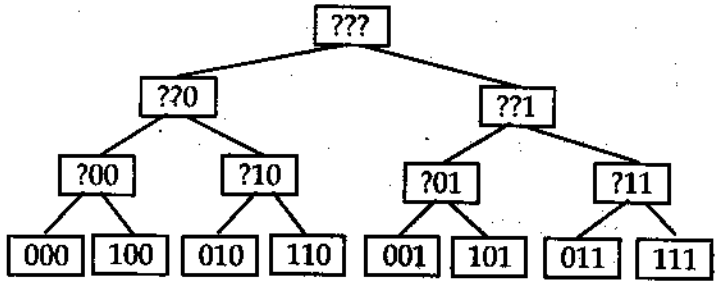
Phép quay lui thấy rõ hơn bằng cấu trúc cây trên không gian lời giải. Ví dụ như chuỗi nhị phân độ dài 3 hình 7.1 (trang 203).

Phép quay lui thực hiện bỏ qua thứ tự trước trên cây này, đồng thời xử lý lá.

Tiết kiệm thời gian bằng cách *tỉa bớt*, nghĩa là bỏ qua những nút bên trong mà nó cho là không có ích.

Ví dụ: k-chuỗi với sự tỉa bớt

Thuật toán 7.3 *string* điền vào mảng từ phải qua trái, thuật toán *string*(m,k) có thể tỉa bớt cách chọn cho $A[m]$ mà nó không tương thích với $A[m+1..n]$.



Hình 7.1

Thuật toán 7.3 Sinh tất cả chuỗi nhị phân có độ dài m

```
procedure   string(m)
  Đầu vào:   Giá trị m.
  Đầu ra:    Sinh ra các chuỗi nhị phân tất cả k-chuỗi có độ dài m.
  if m = 0 then
     process(A);
  else
     for j := 0 to k − 1 do
        if j là phép chọn có hiệu lực cho A[m] then
           A[m] := j;   string(m − 1);
        end if
     end for
  end if
end   string
```

Những bước thiết lập thuật toán quay lui

Những bước thực hiện cho việc quay lui như sau:

1. Chọn đối tượng cơ sở, ví dụ như các chuỗi, các phép hoán vị, các phép tổ hợp (trong bài này ta chỉ chọn các chuỗi).
2. Bắt đầu với mã nguồn chia để trị cho việc sinh ra những đối tượng cơ sở.
3. Đặt mã nguồn cho việc kiểm tra những tính chất thiết kế tại bước cơ sở của hồi quy.
4. Đặt mã nguồn cho sự tỉa bớt trước mỗi lệnh gọi hồi quy.

Như phần trên thì ta phải có thuật toán sinh các đối tượng cơ sở, phần đầu chương này đã nói đến hai thủ tục cụ thể để sinh chuỗi. Một

cách tổng quát ta có thể thiết lập mã nguồn của thuật toán sinh ra như sau:

Thuật toán 7.4 Sinh tất cả chuỗi nhị phân có độ dài m

```
procedure   generate(m)
  Đầu vào:   Giá trị m.
  Đầu ra:    Sinh ra các chuỗi nhị phân tất cả k-chuỗi có độ dài m.
  if m = 0 then
     process(A);
  else
     for mỗi nhánh của số j do
        A[m] := j;   generate(m − 1);
     end for
  end if
end   generate
```

Thuật toán quay lui là tỉa bớt các nhánh không cần thiết, một cách tổng quát ta có thuật toán 7.5 như sau :

Thuật toán 7.5 Sinh tất cả chuỗi nhị phân có độ dài m

```
procedure   generate(m)
  Đầu vào:   Giá trị m.
  Đầu ra:    Sinh ra các chuỗi nhị phân tất cả k-chuỗi có độ dài m.
  if m = 0 then
     process(A);
  else
     for mỗi nhánh của số j do
        if j phù hợp với A[m+1..n] then
           A[m] := j;   generate(m − 1);
        end if
     end for
  end if
end   generate
```

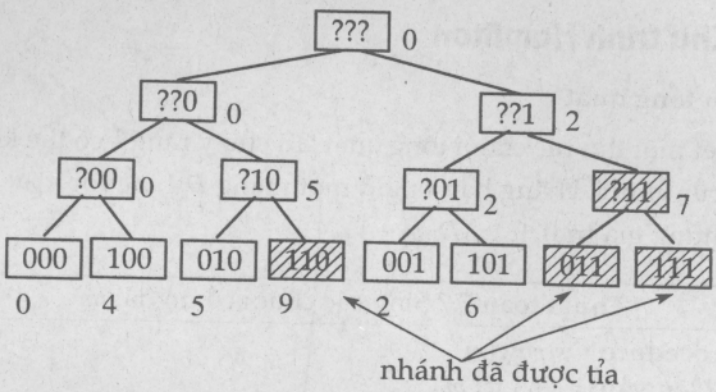
Để ứng dụng các nguyên tắc thiết lập thuật toán kiểu quay lui trên ta xét các bài toán cụ thể sau.

7.1.2. Bài toán ba lô

Ta đã xét bài toán ba lô, cho n ba lô, ta dùng mảng bit $A[1..n]$ và đặt $A[i]$ bằng 1 nếu ba lô i dùng được. Tìm kiếm mọi khả năng thông qua tất cả các chuỗi nhị phân $A[1..n]$ kiểm tra cho thích hợp. Ta cho số liệu cụ thể

$$n = 3, s_1 = 4, s_2 = 5, s_3 = 2, L = 6,$$

ở đây L là vật chứa cần cho các ba lô s_1, s_2, s_3.



Hình 7.2

Thuật toán 7.6 Bài toán m ba lô, cỡ chứa ℓ

```
procedure   knapsack(m, ℓ)
  Đầu vào:  Giá trị m, ℓ.
  Đầu ra:   Ba lô đã được sắp tối ưu.
    if m = 0 then
        if ℓ = 0 then
            print(A);
        end if
    else
        A[m] := 0; knapsack(m − 1, ℓ);
        if s_m <= ℓ then
            A[m] := 1; knapsack(m − 1, ℓ − s_m);
        end if
    end if
end   knapsack
```

Thiết lập thuật toán ta dùng thuật toán chuỗi nhị phân. Phương

cách tỉa của ta cho ℓ là độ dài còn lại của vật chứa ba lô (thuật toán 7.6):

- $A[m] = 0$ là luôn luôn hợp lệ.
- $A[m] = 1$ không hợp lệ nếu $s_m > \ell$; và tỉa nhánh ở đây.

Để in tất cả các lời giải cho bài toán ba lô với n ba lô có cỡ $s_1, s_2, ..., s_n$ và ba lô chứa có cỡ L, gọi thủ tục $knapsack(n, L)$.

Dễ thấy độ phức tạp của thuật toán là $O(2^n)$.

7.1.3. Chu trình Hamilton

1. Chuỗi tổng quát

Ta xét một thủ tục chuỗi tổng quát. Ta chú ý rằng k có thể khác mọi chữ số của chuỗi, chẳng hạn ta giữ một mảng $D[1..n]$ với $A[m]$ lấy một trong những giá trị $0, 1, ..., (D[m] - 1)$.

Thuật toán 7.7 Sinh các chuỗi có độ dài m

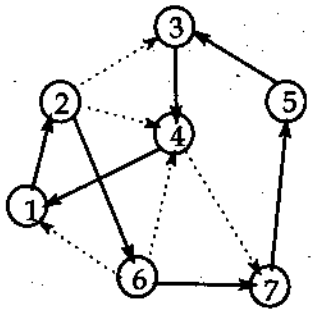
```
procedure   string(m)
  Đầu vào:  Giá trị m.
  Đầu ra:   Sinh ra chuỗi có độ dài m.
  if m = 0 then
      process(A)
  else
      for j := 0 to D[m] - 1 do
          A[m] := j; string(m - 1);
      end for
  end if
end   string
```

Sử dụng cho thuật toán trên với đồ thị có n nốt, khi đó $D[m]$ có thể là bậc của nút m.

2. Chu trình Hamilton

Định nghĩa 7.1. *Chu trình Hamilton* trong đồ thị định hướng là một chu trình đi qua mọi đỉnh của đồ thị đúng một lần trừ đỉnh xuất phát.

Hình 7.3 chỉ ra một chu trình trong đồ thị định hướng có 7 nút.



Hình 7.3

Bài toán đặt ra là cho một đồ thị định hướng $G = (V, E)$, hãy tìm một chu trình Hamilton?

Để giải bài toán này ta ghi nhận đồ thị như một danh sách liền kề: Với mỗi đỉnh $v \in \{1, 2, ..., n\}$ ta ghi vào một danh sách những đỉnh sao cho $(v, w) \in E$.

N	1	2	3
1	2		
2	3	4	6
3	4		
4	1	7	
5	3		
6	1	4	7
7	5		

D	
1	2
2	3
3	4
4	1
5	3
6	1
7	5

Chu trình Hamilton

	1	2	3	4	5	6	7
A	6	2	1	4	3	5	7

Hình 7.4

Ghi nhận chu trình Hamilton như một mảng $A[1..n]$, ở đây chu trình

$$A[n] \to A[n-1] \to \cdots A[2] \to A[1] \to A[n]$$

Với số liệu ở cây hình 7.3, thì danh sách đưa vào $D[1..n]$ như hình 7.4.

3. Thuật toán

Ta dùng thuật toán chuỗi tổng quát. Phương pháp tỉa nhánh: Lưu giữ mảng Boolean $U[1..n]$, ở đây $U[m]$ có giá trị *true* khi và chỉ khi đỉnh m không được sử dụng đến. Khi nhập m vào thì

- $U[m] = true$ là hợp lệ.
- $U[m] = false$ không hợp lệ ; và tỉa nhánh ở đây.

Để giải bài toán chu trình Hamilton, trước tiên ta đặt mảng N hai chiều (các nút lân cận) và mảng một chiều D (bậc) cho việc nhập đồ thị và chạy sau đó. Giả sử $n > 1$ thì chương trình gồm những dòng lệnh sau:

```
for i := 1 to n − 1 do
    U[i] := true;
end for
U[n] := false; A[n] := n;
hamilton(n − 1);
```

procedure *hamilton*(m)

Đầu vào: Giá trị m.

Đầu ra: Chu trình hamilton từ $A[m+1]$ với m những nút không được dùng.

```
if m = 0 then
    process(A)
else
    for j := 0 to D[A[m + 1]] do
        w := N[A[m + 1], j];
        if U[w] then
            U[w] := false;
            A[m] := w; hamilton(m − 1);
            U[w] := true;
        end if
    end for
end if
end hamilton
```

```
procedure   process(A)
  Đầu vào:   Giá trị m.
  Đầu ra:    Kiểm tra chu trình đóng.
18:   ok := false;
19:   for j := 1 to D[A[1]] do
20:       if N[A[1], j] = A[n] then
21:           ok := true;
22:       end if
23:   end for
24:   if ok then
25:       print(A);
26:   end if
end   process
```

Ta chú ý một số điểm trong thuật toán như sau:

- Từ dòng 4, $A[n] = n$.
- Thủ tục *process*(A) đóng chu trình lại. Tại thời điểm ở đó *process*(A) được gọi, $A[1..n]$ chứa đường đi Hamilton từ đỉnh $A[n]$ đến $A[1]$. Nó chỉ còn kiểm tra một cạnh từ $A[1]$ đến $A[n]$ và in ra A nếu việc kiểm tra thắng lợi.
- Dòng 11 thực hiện việc tỉa nhánh, dựa trên cơ sở không có nút mới nào được sử dụng (và được đánh dấu ngay dòng 12 sau đó).
- Dòng 14 đánh dấu nút không được dùng khi ta trở lại từ việc thăm viếng nó.

4. Phân tích thuật toán

Câu hỏi đặt ra là thời gian bao lâu thì thủ tục *hamilton* tìm được một chu trình Hamilton? Ta xét trường hợp xấu nhất là trong cây đã cho không có một chu trình Hamilton nào cả.

Ký hiệu $T(n)$ là thời gian thực hiện của thuật toán *hamilton*. Giả sử đồ thị có bậc cực đại là d. Khi đó, với $b, c \in \mathbb{N}$, ta có

$$T(n) = \begin{cases} bd & \text{nếu } n = 0 \\ dT(n-1) + c & \text{ngược lại} \end{cases}.$$

Bằng cách thế liên tiếp cho kết quả và đánh giá $T(n) = O(d^n)$.

Do đó, tổng thời gian chạy trên n đỉnh là $O(d^n)$ nếu không tồn tại chu trình Hamilton nào. Thời gian chạy thuật toán sẽ là $O(d^n + d) = O(d^n)$ nếu một chu trình được tìm thấy (d cuối cùng là kiểm tra tính đóng của chu trình).

7.1.4. Đường đi của người bán hàng

Phần trước ta tìm tất cả các khả năng của chu trình Hamilton cho một cây định hướng. Bây giờ ta cho cây định hướng có gán nhãn $G = (V,E)$, bài toán đặt ra là tìm chu trình Hamilton có chi phí nhỏ nhất? Bài toán này gọi là bài toán tối ưu và kí hiệu là BTU.

Ta mở rộng bài toán trên với việc đòi hỏi chu trình bị chặn thôi như sau: Cho đồ thị định hướng có nhãn $G = (V,E)$ và $B \in \mathbb{N}$, hãy tìm chu trình Hamilton có chi phí nhỏ hơn hoặc bằng B. Bài toán này gọi là bài toán tối ưu bị chặn, kí hiệu là $BTUC$.

Ta chỉ cần giải bài toán bị chặn là đủ. Khi đó bài toán tối ưu có thể giải được bằng cách dùng tìm kiếm nhị phân.

Giả sử ta có thủ tục $BTUC(x)$ mà nó trả về chu trình Hamilton có độ dài nhiều nhất là x nếu một chu trình tồn tại. Để giải bài toán tối ưu, gọi lệnh $BTU(1,B)$ với B là tổng tất cả chi phí của các cạnh (một chu trình Hamilton chi phí rẻ nhất nhỏ hơn tổng này, nếu nó tồn tại). Ta có thuật toán 7.8 (trang 211).

Chú ý là ta phải gọi $BTUC(B)$ để biết chắc chắn là chu trình Hamilton có tồn tại không đã.

Thuật toán BTU được gọi $O(\log B)$ lần (phân tích như là trong tìm kiếm nhị phân). Giả sử

- Đồ thị có n cạnh.
- Đồ thị có nhãn nhiều nhất là b (như vậy $B \leq bn$).
- Thủ tục $BTUC(B)$ chạy mất thời gian $O(T(n))$.

Khi đó, thủ tục BTU chạy trong thời gian $(\log n + \log b)T(n)$.

Ta thiết kế quay lui cho thủ tục $BTUC$: Kí hiệu $C[v,w]$ là chi phí của

Thuật toán 7.8 Đường đi của người bán hàng chi phí nhỏ nhất

function $BTU(\ell, r)$
Đầu vào: Giá trị ℓ và r.
Đầu ra: Chu trình Hamilton chi phí nhỏ nhất nằm trong khoảng $\geq \ell$ và $\leq r$.

```
if ℓ = r then
    return(BTUC(ℓ))
else
    m := ⌊(ℓ + r)/2⌋;
    if BTUC(m) tìm thấy chu trình then
        return(BTU(ℓ, m))
    else
        return(BTU(m + 1, r))
    end if
end if
```

end BTU

cạnh $(v,w) \in E$ và $C[v,w] = \infty$ nếu $(v,w) \notin E$. Khi đó thuật toán ở phần trước được cụ thể hóa:

procedure $BTUC(B)$
Đầu vào: Giá trị B.
Đầu ra: Trả về chu trình chi phí nhỏ hơn hoặc bằng B.

```
for i := 1 to n − 1 do
    U[i] := true
end for
U[n] := false; A[n] := n;
hamilton(n − 1, B);
```

end $BTUC$

procedure $hamilton(m,b)$
Đầu vào: Giá trị m, b.
Đầu ra: Chu trình Hamilton từ $A[m+1]$ với m những nút không được dùng, chi phí nhỏ hơn hoặc bằng b.

```
if m = 0 then
    process(A, b)
```

```
 8:   else
 9:      for j := 1 to D[A[m+1]] do
10:         w := N[A[m+1], j];
11:         if U[w] ∧ (C[A[m+1], w] <= b) then
12:            U[w] := false;
13:            A[m] := w; hamilton(m-1, b-C[v,w]);
14:            U[w] := true;
15:         end if
16:      end for
17:   end if
end   hamilton
```

```
procedure   process(A, b)
  Đầu vào:   Giá trị A, b.
  Đầu ra:    Kiểm tra chu trình có đóng không?
18:   if (C[A[1], j] <= b) then
19:      print(A);
20:   end if
end   process
```

Một số chú ý về thuật toán:

- Tỉa những nhánh có chi phí đắt được thực hiện tại dòng 11.
- Thủ tục *process* dòng 18-19 dễ dàng được thực hiện khi ta dùng ma trận gần kề.

Thuật toán được phân tích có thời gian $O(d^n)$ như là việc tìm chu trình Hamilton.

7.2. THUẬT TOÁN QUAY LUI VỚI NHỮNG PHÉP HOÁN VỊ

7.2.1. Sinh những hoán vị

Bài toán. *Lập thủ tục sinh ra tất cả những hoán vị của n số khác nhau.*

Lời giải. Trường hợp quay lui với tất cả chuỗi n có độ dài n.

- Lưu giữ mảng Boolean $U[1..n]$, ở đây $U[m]$ là *true* khi và chỉ khi m không được dùng.

- Lưu giữ hoán vị hiện thời vào mảng $A[1..n]$.
- Mục đích là gọi $process(A)$ một lần với $A[1..n]$ chứa mỗi hoán vị.

Gọi thủ tục $permute(n)$:

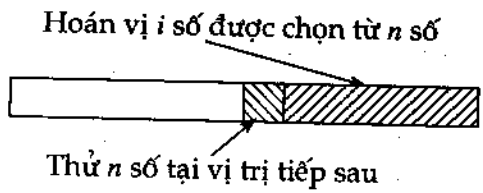
Thuật toán 7.9 Sinh ra các hoán vị

```
procedure  permute(m)
  Đầu vào:  Giá trị m.
  Đầu ra:   Sinh tất cả hoán vị độ dài n.
 1:  if m = 0 then
 2:     process(A, b)
 3:  else
 4:     for j := 1 to n do
 5:        if U[j] then
 6:           U[j] := false;
 7:           A[m] := j; permute(m − 1);
 8:           U[j] := true;
 9:        end if
10:     end for
11:  end if
end  permute
```

Phân tích. Dễ thấy thuật toán chạy với thời gian $O(n^n)$, nếu không phân tích chặt chẽ, nghĩa là không tính đến chuyện tỉa bớt. Phần tỉa bớt thể hiện trong thuật toán tại dòng thứ 9.

Như vậy trạng thái như thế nào khi dòng thứ 9 được thực hiện? Khi đó thuật toán có một số i nào đó mà $0 \leq i < n$:

- Đã điền đầy i chỗ tại phần cuối của A với một phần hoán vị (hình 7.5);



Hình 7.5

- Thử n ứng cử cho chỗ tiếp sau tại dòng 8.

Vậy i chỗ đầu tiên trong hoán vị thì nó như thế nào?

- i số được chọn từ n số không có số trùng nhau.
- Được hoán vị cùng một cách như bước tiếp ta đang phân tích.

Như vậy ta có tất cả các cách để điền vào một phần của hoán vị là

$$C_n^i i!.$$

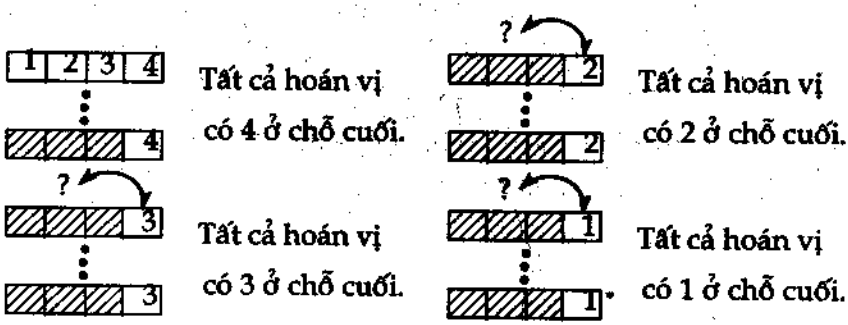
Do đó, số lần thực hiện ở dòng 9 là

$$n\sum_{i=0}^{n-1} C_n^i i! = n\sum_{i=0}^{n-1}\frac{n!}{(n-i)!} = n.n!\sum_{i=0}^{n}\frac{1}{i!}$$
$$\leq (n+1)!(e-1) \leq 1,718(n+1)!.$$

Do đó thủ tục *permute* chạy với thời gian $O((n+1)!)$. Điều này có khá hơn $O(n^n)$. Nhưng cũng không nhiều lắm vì ta có $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.

Những hoán vị nhanh hơn. Thuật toán sinh hoán vị trên không tối ưu vì nó vượt qua giai thừa của n. Ta biết rằng sinh ra $n!$ hoán vị, lại phải tốn thời gian là $O((n+1)!)$.

Thuật toán tốt hơn ta dùng phương pháp chia để trị để tạo ra thuật toán quay lui chỉ sinh ra các hoán vị.



Hình 7.6

- Khởi đầu ta đặt $A[i] = i$ với mọi $1 \leq i < n$.
- Thay vì đặt $A[n]$ bằng $1..n$ ta chuyển đổi những giá trị từ $A[1..n-1]$ vào đó,

Thuật toán 7.10 Sinh ra các hoán vị

```
procedure  permute(m)
  Đầu vào:  Giá trị m.
  Đầu ra:   Sinh tất cả hoán vị độ dài n.
  1:  if m := 1 then
  2:      process(A)
  3:  else
  4:      permute(m − 1);
  5:      for j := 1 to m − 1 do
  6:          swap(A[m],A[j]) với 1 ≤ j < m;
  7:          permute(m − 1);
  8:      end for
  9:  end if
end  permute
```

Làm thế nào để ta chắc chắn giá trị chuyển đổi vào $A[m]$ ở dòng 10 trên là khác nhau? Muốn như vậy A được đặt lại sau mỗi lần gọi hồi quy và lúc đó chuyển đổi $A[m]$ với $A[i]$.

Thuật toán 7.11 Sinh ra các hoán vị

```
procedure  permute(m)
  Đầu vào:  Giá trị m.
  Đầu ra:   Sinh tất cả hoán vị độ dài n.
  1:  if m := 1 then
  2:      process(A)
  3:  else
  4:      permute(m − 1);
  5:      for j := m − 1 downto 1 do
  6:          swap(A[m],A[i]);
  7:          permute(m − 1);
  8:          swap(A[m],A[i]);
  9:      end for
 10:  end if
end  permute
```

Ta gọi $T(n)$ là số lần làm chuyển đổi được thực hiện bởi *permute*(n). Khi đó $T(1) = 0$, và với $n > 2$, $T(n) = nT(n-1) + 2(n-1)$.

Mệnh đề 7.2. *Chứng minh rằng với mọi* $n \geq 1$, $T(n) \leq 2n! - 2$.

Chứng minh. Ta chứng minh bằng quy nạp theo n. Mệnh đề đúng với $n = 1$. Giả sử mệnh đề đúng với n. Khi đó

$$\begin{aligned} T(n+1) &= (n+1)T(n) + 2n \\ &\le (n+1)(2n! - 2) + 2n = 2(n+1)! - 2. \end{aligned}$$

Do đó mệnh đề đúng với mọi n. Từ đó suy ra thuật toán có thời gian thực hiện là $O(n!)$. Vậy nó là tối ưu. ☺

7.2.2. Chu trình Hamilton với đồ thị dày đặc

Ta trở lại bài toán Hamilton ở phần trước và áp dụng cách làm thuật toán tối ưu trên. Dùng ma trận liền kề: $M[i, j]$ là đúng (true) khi và chỉ khi $(i, j) \in E$. Từ dòng 1 đến dòng 4 là đoạn chính của chương trình:

```
1:  for i := 1 to n do
2:      A[i] := i;
3:  end for
4:  hamilton(n − 1);
```

```
procedure  hamilton(m)
  Đầu vào:  Giá trị m.
  Đầu ra:   Chu trình Hamilton từ A[m] với m những nút không được
            dùng.
5:  if m = 0 then
6:      process(A)
7:  else
8:      for j := m downto 1 do
9:          if M[A[m + 1],A[j]] then
10:             swap(A[m],A[j]);
11:             hamilton(m − 1);
12:             swap(A[m],A[j]);
13:         end if
14:     end for
15: end if
end  hamilton
```

procedure *process*(A)
  Đầu vào: Giá trị A.
  Đầu ra: Kiểm tra chu trình có đóng không?
  if $M[A[1], n]$ **then**
    print(A)
  end if
end *process*

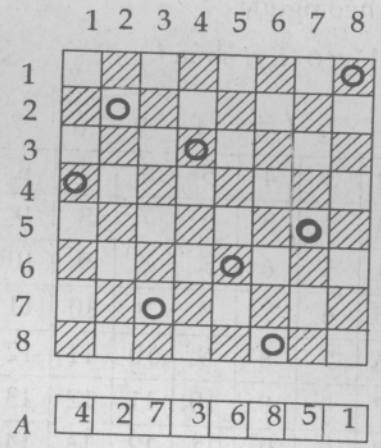
Phân tích thuật toán: Thuật toán này có thời gian thực hiện phụ thuộc vào việc thiết kế như ở phần trước và phần này thì:

- Thời gian chạy thuật toán là $O(d^n)$ khi ta dùng những chuỗi tổng quát.
- Thời gian chạy thuật toán là $O((n-1)!)$ khi ta dùng những hoán vị ở trên.

Hai giá trị bị chặn này cái nào nhỏ hơn cái nào? Nó phụ thuộc vào d. Thuật toán theo chuỗi tốt hơn nếu $d \leq \frac{n}{e}$ ($e \approx 2,7183$).

7.2.3. Bài toán quân hậu hòa bình

Bài toán. Có bao nhiêu cách đặt n quân hậu trên bàn cờ quốc tế cỡ $n \times n$ sao cho không có quân hậu nào cản đường của nhau.



Hình 7.7

Số hàng và số cột từ 1 đến n. Ta dùng mảng $A[1..n]$ với $A[i]$ chứa số hàng của quân hậu trong cột i. Đây là bài toán hoán vị các số trong mảng.

Thuật toán 7.12 Đường đi quân hậu hòa bình

```
procedure   queen(m)
  Đầu vào:  Giá trị m.
  Đầu ra:   In ra các bước đi.
  if m := 0 then
     process(A)
  else
     for j := m downto 1 do
        if OK đặt quân hậu tại(A[i],m) then
           swap(A[m],A[j]);
           queen(m − 1);
           swap(A[m],A[j]);
        end if
     end for
  end if
end   queen
```

Ta đã áp dụng thuật toán sinh hoán vị.

Ta xác định như thế nào nếu đó là *OK* (đúng) để đặt quân hậu tại vị trí (i, j)? Đó là *OK* nếu không có quân hậu nào cùng nằm trên đường chéo xuôi và đường chéo ngược.

Ta xét mảng sau đây: ô (i, j) chứa $i + j$:

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

Ta chú ý rằng :

- Giá trị mỗi ô trên đường chéo ngược là trùng nhau.
- Những ô có giá trị trong khoảng từ 2 đến $2n$.

Ta xét mảng sau đây: ô (i, j) chứa $i - j$:

	1	2	3	4	5	6	7	8
1	0	-1	-2	-3	-4	-5	-6	-7
2	1	0	-1	-2	-3	-4	-5	-6
3	2	1	0	-1	-2	-3	-4	-5
4	3	2	1	0	-1	-2	-3	-4
5	4	3	2	1	0	-1	-2	-3
6	5	4	3	2	1	0	-1	-2
7	6	5	4	3	2	1	0	-1
8	7	6	5	4	3	2	1	0

Ta chú ý rằng :

- Giá trị mỗi ô trên đường chéo xuôi là trùng nhau.
- Những ô có giá trị trong khoảng $-n+1..n-1$.

Ta lưu giữ mảng $b[2..2n]$ và $d[-n+1..n-1]$ khởi đầu là true, sao cho

- $b[i]$ là false nếu đường chéo ngược i bị chiếm bởi quân hậu, $2 \le i \le 2n$.
- $d[i]$ là false nếu đường chéo ngược i bị chiếm bởi quân hậu, $-n+1 \le i \le n-1$.

Khi đó thủ tục được cụ thể là thuật toán 7.13 (trang 220).

7.3. THUẬT TOÁN QUAY LUI VỚI NHỮNG TỔ HỢP

7.3.1. Sinh tổ hợp

Bài toán đặt ra là thiết lập thủ tục sinh ra tất cả tổ hợp của n đối tượng trong khi chọn r đối tượng một lần?

Tương tự như các thuật toán sinh phần trước. Ta lưu giữ tổ hợp hiện thời trong mảng $A[1..r]$. Gọi thủ tục $choose(n, r)$ (thuật toán 7.14):

Thuật toán 7.13 Đường đi quân hậu hòa bình

```
procedure   queen(m)
  Đầu vào:  Giá trị m.
  Đầu ra:   In ra các bước đi.
  if m = 0 then
     process(A)
  else
     for j := m downto 1 do
        if b[A[i] + m] ∧ d[A[i] − m] then
           swap(A[m],A[j]);
           b[A[i] + m] := false;
           d[A[i] − m] := false;
           queen(m − 1);
           b[A[i] + m] := true;
           d[A[i] − m] := true;
           swap(A[m],A[j]);
        end if
     end for
  end if
end   queen
```

Thuật toán 7.14 Sinh tổ hợp chập q trong m phần tử

```
procedure   choose(m,q)
  Đầu vào:  Giá trị m,q.
  Đầu ra:   Sinh ra các tổ hợp chọn q phần tử từ 1 đến m.
  if m = 0 then
     process(A)
  else
     for i := q to m do
        A[q] := i;
        choose(i − 1,q − 1);
     end for
  end if
end   choose
```

7.3.2. Chứng minh tính đúng đắn của thuật toán

Ta phải chứng minh theo ba phần:

1. Chỉ những tổ hợp mới được sinh ra.

2. Những số tổ hợp đúng đã được sinh.
3. Không một tổ hợp nào sinh ra hai lần.

Trước khi đi vào chứng minh cụ thể ta nhắc lại những đẳng thức tổ hợp sau đây (bạn đọc có thể chứng minh bằng quy nạp):

$$C_n^r + C_n^{r-1} = C_{n+1}^r. \tag{7.1}$$

Với mọi $n \geq 0$ và $1 \leq r \leq n$,

$$\sum_{i=r}^{n} C_i^r = C_{n+1}^r. \tag{7.2}$$

Mệnh đề 7.3. *Nếu* $0 \leq r \leq n$, *khi đó trong những tổ hợp sinh bởi* $choose(n,r)$, $A[1..r]$ *chứa những tổ hợp r phần tử từ* 1 *đến n.*

Chứng minh. Chứng minh bằng quy nạp theo r. Theo giả thiết rỗng là đúng cho $r = 0$.

Ta giả thiết $r > 0$ và với mọi $i \geq r-1$, trong những tổ hợp sinh ra bởi $choose(i, r-1)$, $A[1..r-1]$ chứa tổ hợp của $r-1$ phần tử chọn từ $1..i$.

Bây giờ, $choose(n,r)$ gọi $choose(i-1, r-1)$ với i chạy từ r đến n. Do đó theo giả thiết quy nạp và từ $A[r]$ được gán i vào, trong những tổ hợp được sinh ra bởi $choose(n,r)$, $A[1..r]$ chứa tổ hợp $r-1$ phần tử chọn từ $1..i-1$ sau đó là giá trị i.

Nghĩa là $A[1..r]$ chứa tổ hợp r phần tử chọn từ $1..n$. ☺

Mệnh đề 7.4. *Gọi lệnh* $choose(n,r)$ *sinh ra đúng* C_n^r *tổ hợp.*

Chứng minh. Ta lại chứng minh bằng quy nạp toán học theo r. Mệnh đề đúng với $r = 0$.

Giả sử $r > 0$ và gọi $choose(i, r-1)$ sinh ra đúng C_i^{r-1} tổ hợp với mọi $r-1 \leq i \leq n$.

Bây giờ, $choose(n,r)$ gọi $choose(i-1, r-1)$ với i chạy từ r đến n. Do đó, theo giả thiết quy nạp số những tổ hợp sinh ra là

$$\sum_{i=r}^{n} C_{i-1}^{r-1} = \sum_{i=r-1}^{n-1} C_i^{r-1} = C_n^r.$$

Đẳng thức sau cùng suy ra từ việc định chỉ số lại và dùng đẳng thức (7.2).

☺

Mệnh đề 7.5. *Gọi thủ tục choose(n,r) sinh ra những tổ hợp r phần tử chọn từ 1 đến n không có tổ hợp lặp lại.*

Chứng minh. Ta cũng chứng minh bằng quy nạp theo r và hoàn toàn tương tự như các mệnh đề trên. ☺

Bây giờ ta chứng minh tính đúng đắn của thuật toán: Theo mệnh đề 7.3, một lần gọi thủ tục *choose*(n,r) sinh ra tổ hợp r phần tử từ 1 đến n (vì r phần tử cùng một lúc được sinh ra từ mảng A). Theo mệnh đề 7.5, nó không có tổ hợp lặp lại. Theo mệnh đề 7.4 nó sinh ra đúng số lượng tổ hợp cần có. Do đó nó sinh ra đúng những tổ hợp của r phần tử chọn từ 1 đến n.

7.3.3. Phân tích thuật toán

Kí hiệu $T(n,r)$ là số những phép gán vào A được thực hiện trong *choose*(n,r).

Mệnh đề 7.6. *Với mọi $n \geq 1$ và $0 \leq r \leq n$, ta có*

$$T(n,r) \leq rC_n^r.$$

Chứng minh. Khi ta gọi *choose*(n,r) thì có lời gọi hồi quy sau đây:

```
for i := r to n do
    A[q] := i;
    choose(i-1, q-1);
end for
```

Ta có những phép toán được thực hiện lần lượt như sau:

$choose(r-1,r-1)$ $\quad T(r-1,r-1)+1$
$choose(r,r-1)$ $\quad T(r,r-1)+1$
$\vdots$
$choose(n-1,r-1)$ $\quad T(n-1,r-1)+1$

Do đó, cộng lại cho ta kết quả

$$T(n,r) = \sum_{i=r-1}^{n-1} T(i,r-1) + (n-r+1).$$

Ta sẽ chứng minh bằng quy nạp theo r cho bất đẳng thức của mệnh đề.

Mệnh đề đúng với $r = 0$.

Giả sử $r > 0$ và mọi $i < n$,

$$T(i, r-1) \leq (r-1)C_i^{r-1}.$$

Khi đó theo giả thiết quy nạp và công thức (7.2) ta có

$$\begin{aligned}
T(n,r) &= \sum_{i=r-1}^{n-1} T(i, r-1) + (n-r+1) \\
&\leq \sum_{i=r-1}^{n-1} ((r-1)C_i^{r-1}) + (n-r+1) \\
&= (r-1) \sum_{i=r-1}^{n-1} C_i^{r-1} + (n-r+1) \\
&= (r-1)C_n^r + (n-r+1) \\
&\leq rC_n^r.
\end{aligned}$$

Bất đẳng thức cuối cùng đúng do với $1 \leq r \leq n$,

$$C_n^r \geq n - r + 1.$$

Bất đẳng thức này cũng chứng minh bằng quy nạp theo r. Dễ thấy với $r = 1$ bất đẳng thức đúng. Giả sử đúng với $r-1$, ta phải chứng minh nó đúng cho r. Thật vậy,

$$\begin{aligned}
C_n^r &= \frac{n!}{r!(n-r)!} \\
&= \frac{n(n-1)!}{r(r-1)!((n-1)-(r-1))!} \\
&= \frac{n}{r}\frac{(n-1)!}{(r-1)!((n-1)-(r-1))!} \\
&= \frac{n}{r}C_{n-1}^{r-1} \\
&\geq \frac{n}{r}((n-1)-(r-1)) \text{ (theo giả thiết quy nạp)} \\
&\geq n - r + 1 \text{ (vì } r \leq n\text{)}.
\end{aligned}$$

☺

Vì thế, thuật toán sinh tổ hợp có thời gian chạy là $O(rC_n^r)$. Như vậy thuật toán này không tối ưu. Vì số tổ hợp chỉ có C_n^r, mà thời gian thực hiện lại phải nhân với một hệ số. Người ta đã chứng minh được rằng

với $r \leq \frac{n}{2}$ thì thuật toán mới tối ưu, nghĩa là chỉ đúng cho những tổ hợp với số ít phần tử.

7.3.4. Tính tối ưu của thuật toán

Ta kí hiệu A là số phép gán vào dãy A, C là số tổ hợp, và *Ratio* là tỉ số $\frac{A}{C}$. Qua thực nghiệm ta có bảng sau:

n	r	A	C	*Ratio*
20	1	20	20	1,00
20	2	209	190	1,10
20	3	1329	1140	1,17
20	4	5984	4845	1,24
20	5	20348	15504	1,31
20	6	54263	38760	1,40
20	7	116279	77520	1,50
20	8	203489	125970	1,62
20	9	293929	167960	1,75
20	10	352715	184756	1,91
20	11	352715	167960	2,10
20	12	293929	125970	2,33
20	13	203489	77520	2,62
20	14	116279	38760	3,00
20	15	54263	15504	3,50
20	16	20348	4845	4,20
20	17	5984	1140	5,25
20	18	1329	190	6,99
20	19	209	20	10,45
20	20	20	1	20,00

Từ định nghĩa $T(n,r)$ và bảng trên ta dự đoán rằng $T(n,r) < 2C_n^r$ với $r \leq \frac{n}{2}$. Điều dự đoán này hoàn toàn đúng thông qua mệnh đề sau :

Mệnh đề 7.7. *Nếu* $1 \leq r \leq \frac{n}{2}$ *thì*

$$T(n,r) \leq 2C_n^r - r.$$

Chứng minh. Ta chứng minh mệnh đề với $r = 1$ và $r = 2$. Sau đó chứng minh mệnh đề với $3 \leq r \leq \frac{n}{2}$ bằng phương pháp quy nạp theo n.

Trường hợp $r = 1$: Theo định nghĩa $T(n,r) = n$ và $2C_n^1 - 1 = 2n - 1 \geq n$. Do đó $T(n,1) \leq 2C_n^1 - 1$, đó là điều cần chứng minh.

Trường hợp $r = 2$: Từ định nghĩa ta có

$$T(n,2) = \sum_{i=1}^{n-1} T(i,1) + (n-1) = \sum_{i=1}^{n-1} i + (n-1)$$
$$= \frac{n(n-1)}{2} + (n-1) = \frac{(n+2)(n-1)}{2},$$

và

$$2C_n^2 - 2 = n(n-1) - 2.$$

Ta cần chứng minh

$$\frac{(n+2)(n-1)}{2} \leq n(n-1) - 2 \Leftrightarrow 2n(n-1) - (n+2)(n-1) \geq 4$$
$$\Leftrightarrow (n-2)(n-1) \geq 4 \Leftrightarrow n^2 - 3n - 2 \geq 0.$$

Điều này đúng vì $n \geq 4$ (do giả thiết $r \leq \frac{n}{2}$) và vì nghiệm lớn nhất của $n^2 - 3n - 2 \geq 0$ là $\frac{3+\sqrt{17}}{2} < 3,562$.

Trường hợp $3 \leq r \leq \frac{n}{2}$: Ta chứng minh bằng quy nạp theo n bất đẳng thức trong bổ đề.

Bước cơ sở khi $n = 6$. Khi đó chỉ chọn $r = 3$. Bằng cách tính cụ thể thuật toán dùng 34 phép gán tạo ra 20 tổ hợp. Từ $2.20 - 2 = 38 > 34$, nên bất đẳng thức đúng trong trường hợp này.

Bước quy nạp, giả sử $n \geq 6$, điều này kéo theo $r \geq 3$. Giả thiết bất đẳng thức đúng đến $n-1$, khi đó kết hợp với trường hợp $r = 1,2$, ta có

$$T(n,r) = \sum_{i=r-1}^{n-1} T(i,r-1) + (n-r+1)$$
$$= \sum_{i=r-1}^{n-1} \left(2C_i^{r-1} - (r-1)\right) + (n-r+1)$$
$$= 2\sum_{i=r-1}^{n-1} C_i^{r-1} - (n-r+1)(r-2)$$
$$= 2C_n^r - (n-r+1)(r-2) \leq 2C_n^r - (n - \frac{n}{2} + 1)(3-2) < 2C_n^r - r$$

Đó là điều cần chứng minh. ☺

7.4. BÀI TẬP

▷ **7.1.** Giả sử $A[1..n]$ chứa n giá trị khác nhau. Thuật toán sau đây sinh ra tất cả hoán vị của $A[1..n]$ theo nghĩa sau đây: Lời gọi thủ tục *permute*(n) sẽ là kết quả trong thủ tục *process*(A), gọi một lần với $A[1..n]$ mà nó chứa mỗi hoán vị phần tử trong nó.

```
procedure  permute(n)
  if n = 1 then
     process(A)
  else
     B := A;
     permute(n − 1);
     for i := 1 to n − 1 do
        chuyển đổi A[n] với A[1];
        permute(n − 1);
     end for
     A := B chuyển vòng tròn một chỗ về phía phải.
  end if
end  permute
```

Tìm hàm thời gian của thuật toán trên.

TÀI LIỆU THAM KHẢO

[1] G. Brassard, P. Bratley, *Fundamentals of algorithmics*, Prentice-Hall, 1996.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of computer algorithms*, Addison-Wesley Publishing Company, Reading, MA, 1974.

[3] K. A. Berman and J. L. Paul, *Fundamentals of sequential and parallel algorithms*, PWS Publishing Company, 1997.

[4] S. Baase, *Computer algorithms: Introduction to Design and Analysis*, Addison-Wesley, 1978.

[5] L. Banachowski, A. Kreczmar, W. Rytter *Analysis of Algorithms and Data structures*, Addison-Wesley, 1990.

[6] R. j. McEliece, R. B. Ash, C. Ash, *Introduction to Discrete Mathematics*, McGraw-Hill Book Co., 1989.

[7] Th. Cormen, Ch. Leiserson, R. Rivest, *Introduction to algorithm*, The MIT Press, 1997.

[8] D. E. Knuth, *The Art of Computer Programming. Volume I. Fundamental Algorithms*, Massachusetts, 1968.

[9] D. H. Greene and D. E. Knuth, *Mathematics for the Analysis of Algorithms*, Birkhauser, 1982.

[10] J. D. Smith, *Design and Analysis of Algorithms*, PWS-KENT, 1989.

[11] I. Parberry and W. Gasarch, *Problems on Algorithms*, Prentice-Hall, 2002.

[12] G. H. Hardy, J.E. Littlewood, G. Polya *Bất đẳng thức*, NXB ĐHQG Hà Nội, 2002.

[13] N. H. Điển, *Phương pháp quy nạp toán học*, NXB GD, 2000.

[14] N. H. Điển, *LaTeX với gói lệnh và phần mềm công cụ*, NXBĐHQGHN, 2004.

MỤC LỤC

Chịu trách nhiệm xuất bản :
Chủ tịch HĐQT kiêm Tổng Giám đốc NGÔ TRẦN ÁI
Phó Tổng Giám đốc kiêm Tổng biên tập NGUYỄN QUÝ THAO

Biên tập nội dung:
NGUYỄN TRỌNG THIỆP
Trình bày bìa:
BÙI QUANG TUẤN
Chế bản:
NGUYỄN HỮU ĐIỂN
Sửa bản in:
NGUYỄN TRỌNG THIỆP

MỘT SỐ VẤN ĐỀ VỀ THUẬT TOÁN
Mã số: 8I107M6-DAI
In 1.000 bản, khổ 17 x 24 cm, tại Công ty In và Văn hoá phẩm.
Số in 70. Số xuất bản: 26-2006/CXB/2-2097/GD.
In xong và nộp lưu chiểu Quý I năm 2006.

CÔNG TY CỔ PHẦN SÁCH ĐẠI HỌC - DẠY NGHỀ
HEVOBCO
Địa chỉ : 25 HànThuyên, Hà Nội

TÌM ĐỌC SÁCH THAM KHẢO CHUYÊN TOÁN THPT

Dùng cho học sinh khá và giỏi Toán của cùng tác giả

NGUYỄN HỮU ĐIỂN

1. Phương pháp quy nạp Toán học
2. Những phương pháp điển hình trong giải Toán phổ thông
3. Sáng tạo trong giải Toán phổ thông
4. Đa thức và ứng dụng
5. Giải Toán bằng phương pháp đại lượng bất biến
6. Giải Toán bằng phương pháp đại lượng cực biên
7. Một số chuyên đề hình học tổ hợp
8. Một số vấn đề về thuật toán

Bạn đọc có thể mua tại các Công ti Sách - Thiết bị trường học ở địa phương hoặc các Cửa hàng Sách của Nhà xuất bản Giáo dục :

** 25 Hàn Thuyên, 187 Giảng Võ, 23 Tràng Tiền, 232 Tây Sơn - Hà Nội.*
** 15 Nguyễn Chí Thanh - TP Đà Nẵng*
** 240 Trần Bình Trọng - Quận 5 - TP. Hồ Chí Minh.*

8 934980 697757

Giá: 25.300đ