CHƯƠNG I

ĐẠI CƯƠNG VỀ THUẬT TOÁN

I. KHÁI NIỆM THUẬT TOÁN

a. Định nghĩa. *Thuật toán* là tập hợp hữu hạn các thao tác dẫn đến lời giải cho một vấn đề hay bài toán nào đó trong thời gian hữu hạn.

◊ *Ví dụ*

Giải phương trình bậc 2: $a.x^2+b.x+c = 0$, $a \neq 0$.

Bước 1. Tính $\Delta = b^2-4.a.c$

Bước 2. Nếu $\Delta < 0$, thì kết luận: phương trình vô nghiệm (nghiệm thực). Kết thúc.

Ngược lại, tức $\Delta \geq 0$, thì nghiệm $x_{1,2} = (-b \pm \sqrt{\Delta})/(2.a)$. Kết thúc.

◊ *Ví dụ*

Tìm ước số chung lớn nhất (UCLN) của 2 số nguyên a, b ($a \leq b$).

Bước 1. Nếu $a = 0$, thì UCLN$(0,b) = b$, Kết thúc.

Bước 2. Đặt $c=a$; $a = b \bmod a$; $b=c$;

Quay lại bước 1.

◊ *Ví dụ*. Thuật toán tìm kiếm nhị phân.

Tìm phần tử ***x*** trong danh sách có thứ tự tăng dần $a[1], a[2], ..., a[n]$.

Cách tìm:

Tìm phần tử ở giữa $a[m]$, $m = (1+n)/2$.

Nếu $x = a[m]$, thì $a[m]$ là phần tử cần tìm. Kết thúc.

Nếu $x < a[m]$, thì tìm x trong nửa danh sách từ $a[1]$ đến $a[m-1]$.

Nếu $x > a[m]$, thì tìm x trong nửa danh sách từ $a[m+1]$ đến $a[n]$.

◊ *Ví dụ*. *Bài toán Josephus*

Một nhóm n binh sĩ bị kẻ thù bao vây và họ phải chọn một người đi cầu cứu viện binh. Việc chọn người được thực hiện như sau.

Cho trước một số nguyên k gọi là *bước đếm*. Các binh sĩ xếp thành vòng tròn đánh số 1, 2, ... , n, bắt đầu từ người chỉ huy.

Đếm (theo chiều kim đồng hồ) từ người thứ nhất, khi đạt đến k thì binh sĩ tương ứng bị loại ra khỏi vòng.

Lại bắt đầu đếm từ người tiếp theo, khi đạt đến k thì binh sĩ tương ứng bị loại ra khỏi vòng.

Quá trình này tiếp tục cho đến khi chỉ còn lại một binh sĩ, và người này chính là người được điều đi cầu cứu viện binh.

b. Các tính chất cơ bản của thuật toán

- *Tính đúng đắn*: Giải quyết đúng vấn đề.
- *Tính hữu hạn* : Thuật toán phải kết thúc sau một số bước hữu hạn và sau thời gian hữu hạn (chấp nhận).
- *Tính tất định*: Tại mỗi bước, với cùng một đầu vào thuật toán phải cho cùng một đầu ra.
- *Tính phổ quát*: Thuật toán phải áp dụng được cho lớp nhiều bài toán.
- *Tính hiệu quả*: Về không gian (tiết kiệm bộ nhớ, tài nguyên khác), về thời gian (tính nhanh).

c. Cấu trúc điều khiển thuật toán

i) *Cấu trúc tuần tự:*

Thực hiện tuần tự các công việc.

ii) *Cấu trúc lặp:*

Cấu trúc lặp: *Với i* chạy từ 1 đến *n Thực hiện <công việc>*;

Cấu trúc lặp: *Lặp lại <công việc> Cho đến khi <điều kiện dừng> thỏa mãn*;

iii) *Cấu trúc chọn*:

Nếu <*điều kiện* 1> thỏa mãn, thì thực hiện <*công việc* 1>,

Ngược lại, thực hiện <*công việc* 2>.

II. PHÂN TÍCH ĐÁNH GIÁ THUẬT TOÁN

1. Các tiêu chuẩn đánh giá

a. Tính đúng đắn

Thuật toán phải giải đúng bài toán. Thông thường để kiểm tra tính đúng đắn của thuật toán người ta cài đặt chương trình thể hiện thuật toán và chạy thử nghiệm với dữ liệu mẫu và so sánh với kết quả đã biết.

b. Tính hữu hạn

Thuật toán phải kết thúc sau một số bước hữu hạn và sau thời gian hữu hạn (chấp nhận).

c. Tính tất định

Tại mỗi bước, với cùng một đầu vào thuật toán phải cho cùng một đầu ra.

d. Tính phổ quát: Thuật toán phải áp dụng được cho lớp nhiều bài toán.

e. Tính hiệu quả

- Thời gian thực hiện nhanh.
- Tài nguyên (Bộ nhớ, CPU, mạng, …) sử dụng tiết kiệm.

2. Phân tích thời gian thực hiện

a. Độ phức tạp tính toán

Với một bài toán, không phải chỉ có một thuật toán. Chọn một thuật toán đưa tới kết quả nhanh là một yêu cầu thực tế. Nhưng căn cứ vào đâu để có thể nói thuật toán này nhanh hơn hay chậm hơn thuật toán kia.

Có thể thấy ngay, thời gian thực hiện một thuật toán (chính xác hơn là chương trình thể hiện thuật toán đó) phụ thuộc vào rất nhiều yếu tố. Một yếu tố cần chú ý trước tiên là kích thước dữ liệu đầu vào. Chẳng hạn sắp xếp một dãy số phải chịu ảnh hưởng của số lượng các số thuộc dãy số đó. Nếu gọi n là số lượng dữ liệu (kích thước) đầu vào, thì thời gian thực hiện T của một thuật toán phải được biểu diễn như một hàm của n : $T(n)$.

Các kiểu lệnh và tốc độ xử lý của máy tính, ngôn ngữ lập trình và chương trình dịch ngôn ngữ đó đều ảnh hưởng tới thời gian thực hiện. Nhưng những yếu tố này không giống nhau trên các máy khác nhau, vì vậy không thể dựa vào chúng để xác lập $T(n)$. Điều đó cũng có nghĩa là $T(n)$ không biểu diễn được bằng giây, phút ...

Thời gian $T(n)$ ở đây phải hiểu là *cấp độ lớn* của số lượng phép tính, phụ thuộc vào kích thước đầu vào, và được gọi là *độ phức tạp* của thuật toán.

Tuy nhiên, thời gian $T(n)$ không chỉ phụ thuộc vào kích thước đầu vào mà còn phụ thuộc vào trạng thái dữ liệu đầu vào. Chẳng hạn nếu dãy cần sắp xếp đã

được sắp xếp từ trước với mức độ nào đó, thì thời gian sắp xếp sẽ nhanh hơn nhiều so với dãy bất kỳ.

Vì vậy chúng ta cần phân biệt các loại độ phức tạp thực hiện thuật toán sau:

- Độ phức tạp trung bình : $T_{tb}(n)$
- Độ phức tạp xấu nhất : $T_{max}(n)$
- Độ phức tạp tốt nhất : $T_{min}(n)$

Về lý thuyết, người ta thường đánh giá độ phức tạp $T(n)$ trong trường hợp xấu nhất.

Độ phức tạp thường được biểu diễn thông qua các hàm đa thức, luỹ thừa, hàm mũ. Để so sánh cấp độ lớn giữa các hàm người ta dùng ký hiệu O.

• *Ký pháp* O (O *lớn*)*:* Cho các hàm nguyên $f(n)$ và $g(n)$ phụ thuộc số nguyên n. Ta nói hàm $f(n)$ có *cấp* $g(n)$ và viết

$$f(n) = O(g(n))$$

nếu tồn tại các hằng số k, n_0 thoả mãn

$$f(n) \le k.g(n) \quad \forall n \ge n_0$$

◊ *Ghi chú*. Nếu tồn tại $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ hữu hạn, thì $f(n) = O(g(n))$.

◊ *Ví dụ*

$f(n) = 2.n^2 - 3.n + 100 \quad \Rightarrow \quad f(n) = O(n^2)$

$f(n) = K$ (hằng số) $\quad \Rightarrow \quad f(n) = O(1)$

$f(n) = 3.n \quad \Rightarrow \quad f(n) = O(2.n) = O(n)$

$f(n) = 2.n + 5.\ln(n) \quad \Rightarrow \quad f(n) = O(n)$

$f(n)$ là đa thức bậc $m \quad \Rightarrow \quad f(n) = O(n^m)$

• *Định nghĩa:* Nếu độ phức tạp $T(n)$ của một thuật toán thoả mãn $T(n)= O(g(n))$, thì ta nói thuật toán có độ phức tạp $g(n)$.

Thông thường các hàm thể hiện độ phức tạp tính toán của thuật toán có dạng $\log_2(\log_2 n)$, $\log_2(n)$, n, $n.\log_2(n)$, n^2, n^3, ..., n^m, 2^n, $n!$, n^n, ...

Các hàm như 2^n, $n!$, n^n gọi là hàm loại *mũ*. Một thuật toán có độ phức tạp hàm mũ thì tốc độ rất chậm. Các hàm $\log_2(\log_2 n)$, $\log_2(n)$, n, $n.\log_2(n)$, n^2, n^3, ..., n^m được gọi là *hàm đa thức*. Một thuật toán có độ phức tạp hàm đa thức thì tốc độ chấp nhận được.

Bảng dưới hiển thị giá trị các hàm trên đối với một số giá trị n.

Hàm / n	$\log_2(\log_2 n)$	$\log_2 n$	n	$n.\log_2 n$	n^2	n^3	2^n
2	0	1	2	2	4	8	4
4	1	2	4	8	16	64	16
16	2	4	16	64	256	4096	65536

32	$\log_2 5$	5	32	160	1024	32768	2147483684
1048576	4,32	20	1048576	$2{,}1*10^7$	$1{,}1*10^{12}$	$1{,}15*10^{18}$	$6{,}7*10^{315652}$

Giả sử mỗi lệnh thực hiện trong 1 micro giây, bảng dưới chỉ ra thời gian cần thiết để thực hiện $f(n)$ lệnh đối với các hàm thông dụng và với $n = 256$.

Hàm	***Thời gian***
$\log_2(\log_2 n)$	3 micro giây
$\log_2 n$	8 micro giây
n	0,25 mili giây
$n.\log_2 n$	2 mili giây
n^2	65 mili giây
n^3	17 giây
2^n	$3{,}7*10^{61}$ thế kỷ

b. Các quy tắc tính toán độ phức tạp

• *Quy tắc tổng*

Giả sử $T_1(n) = O(f(n))$ và $T_2(n) = O(g(n))$ là độ phức tạp của 2 đoạn thuật toán P_1 và P_2 kế tiếp nhau. Khi đó, độ phức tạp của P_1 và P_2 là

$$T(n) = T_1(n) + T_2(n) = O(f(n) + g(n)) = O(max(f(n), g(n)))$$

Tổng quát, độ phức tạp của nhiều đoạn thuật toán kế tiếp nhau bằng tổng độ phức tạp của mỗi đoạn thuật toán:

$T(n) = T_1(n) + T_2(n) + .. + T_k(n) = O(f_1(n) + f_2(n) + ... + f_k(n))$

$= O(max(f_1(n), f_2(n), ..., f_k(n)))$

trong đó $T_i(n) = O(f_i(n))$, $\forall i = 1, ..., k$.

◊ *Ví dụ:* Giả sử một thuật toán có 3 bước thực hiện mà độ phức tạp từng bước lần lượt là $O(n^2)$, $O(n^3)$ và $O(n.\log_2 n)$.

Khi đó, độ phức tạp thuật toán là $O(max(n^2, n^3, n.\log_2 n)) = O(n^3)$.

• *Quy tắc nhân*

Giả sử $T_1(n) = O(f(n))$ và $T_2(n) = O(g(n))$ là độ phức tạp của 2 đoạn thuật toán P_1 và P_2 lồng nhau. Khi đó, độ phức tạp của thuật toán thực hiện P_1 và P_2 lồng nhau là

$$T(n) = T_1(n) * T_2(n) = O(f(n)*g(n))$$

◊ *Ví dụ*. Thuật toán tính trị trung bình *Mean* của n số.

1. Đọc n;
2. $Sum := 0$;
3. $i := 1$;

4. Với ($i \le n$) thực hiện
   a. đọc *Number*;
   b. *Sum* := *Sum* + *Number*;
   c. i := i+1;
5. *Mean* := *Sum* / *n*;

Để tính độ phức tạp ta phân tích như sau:

Lệnh	*Số lần thực hiện*
1	1
2	1
3	1
4 (<=)	$n + 1$
a	n
b	n
c	n
5	1
Tổng cộng:	$4.n + 5$

Như vậy độ phức tạp $T(n) = 4n + 5 = O(n)$.

Áp dụng quy tắc tổng và quy tắc nhân ta có

$T(n) = O(max(T_1(n), T_2(n), T_3(n), T_4(n), T_5(n)))$

$= O(T_4(n)) = O(n*3) = O(n)$

• *Phép toán tích cực*

Dựa vào các nhận xét đã nêu ở trên về các quy tắc tính toán độ phức tạp, ta chỉ cần chú ý tới các bước tương ứng với một phép toán mà ta gọi là phép toán tích cực. *Phép toán tích cực* trong giải thuật là phép toán mà số lần thực hiện nó không ít hơn số lần thực hiện các phép toán khác. Chú ý rằng phép toán tích cực có thể không duy nhất.

Ví dụ trong thuật toán tính trị trung bình trên phép toán so sánh ($i \le n$) trên dòng 4 là tích cực và nó thực hiện n+1 lần. Vậy ta có độ phức tạp

$$T(n) = O(n).$$

c. Cách tính độ phức tạp (ĐPT) cho các cấu trúc thuật toán

• *Lệnh gán*: ĐPT bằng kích thước dữ liệu.

Gán 1 hằng: ĐPT = 1.

Gán 1 mảng n phần tử: ĐPT = n.

• Cấu trúc chọn: IF <*điều kiện*> THEN <*lệnh*>;

ĐPT = ĐPT(<*điều kiện*>) + ĐPT(<*lệnh*>)

• Cấu trúc rẽ nhánh: IF <*điều kiện*> THEN <*lệnh* 1> ELSE <*lệnh* 2>;

ĐPT = ĐPT(<*điều kiện*>) + max(ĐPT(<*lệnh* 1>), ĐPT(<*lệnh* 2>))

• Cấu trúc lặp: *Với i chạy từ* 1 *đến n Thực hiện* <*lệnh*>;

ĐPT = *n**ĐPT(<*lệnh*>)

• Cấu trúc lặp: *Lặp lại* <*lệnh*> *Cho đến* <*điều kiện dừng*>;

ĐPT = *n**(ĐPT(<*lệnh*>) + ĐPT(<*điều kiện dừng*>))

trong đó *n* là số lần tối đa thực hiện vòng lặp.

• Cấu trúc lặp: *Lặp lại* <*lệnh*> *Trong khi* <*điều kiện tiếp tục*>;

ĐPT = *n**(ĐPT(<*lệnh*>) + ĐPT(<*điều kiện tiếp tục*>))

trong đó *n* là số lần tối đa thực hiện vòng lặp.

• Cấu trúc lặp: *Trong khi* <*điều kiện lặp*> *Thực hiện* <*lệnh*>;

ĐPT = *n**ĐPT(<*lệnh*>) + (*n*+1)*ĐPT(<*điều kiện lặp*>)

trong đó *n* là số lần tối đa thực hiện vòng lặp.

◊ *Ví dụ*. Thuật toán tìm kiếm tuyến tính.

Tìm phần tử (biến) *Item* trong danh sách *a*[1], *a*[2], ..., *a*[*n*]. Biến *Found* sẽ có giá trị *true* và biến *Loc* có giá trị là vị trí của *Item* nếu tìm ra, ngược lại *Found* = *false* và *Loc* = *n* + 1.

1. *Found := false;*
2. *Loc* := 1;
3. while (*Loc* <= *n*) and not *Found* do
   a. if *Item* = *a*[*Loc*] then
   b. *Found* := *true*
      else
   c. *Loc* := *Loc* + 1

Phép toán (*Loc* <= *n*) là tích cực, thực hiện *n*+1 lần trong trường hợp xấu nhất. Vậy

ĐPT = O(*n*).

◊ *Ví dụ*. Thuật toán tìm kiếm nhị phân.

Tìm phần tử (biến) *Item* trong danh sách có thứ tự tăng dần *a*[1], *a*[2], ..., *a*[*n*]. Biến *Found* sẽ có giá trị *true* và biến *Mid* có giá trị là vị trí của *Item* nếu tìm ra, ngược lại *Found* = *false*.

1. *Found* := *false;*
2. *First* := 1;

3. *Last* := *n;*
4. while (*First* <= *Last*) and not *Found* thực hiện
   begin
   a. *Mid* := (*First* + *Last*) div 2;
   b. if *Item* < *a*[*Mid*] then
   c. *Last* := *Mid* – 1
      else
   d. if *Item* > *a*[*Mid*] then
   e. *First* := *Mid* + 1
      else
   f. *Found* := *true*
   end;

Giả sử số bước lặp là *k*. Phép toán 4 (*First* <= *Last*) là tích cực và thực hiện nhiều nhất là *k* lần. Vậy ĐPT = O(*k*). Ta phải biểu diễn *k* qua *n*.

Mỗi lần qua vòng lặp độ lớn của danh sách giảm một nửa. Lần cuối cùng, tức lần thứ *k*, danh sách còn độ lớn ít nhất là 1. Vì thế độ lớn của danh sách sau (*k*–1) lần lặp là

$$n / 2^{k-1} \geq 2.$$

Suy ra

$$n \geq 2^k \Rightarrow k \leq \log_2 n.$$

Như vậy ĐPT = O(*k*) = O($\log_2 n$).

III. ĐỆ QUY VÀ GIẢI THUẬT ĐỆ QUY

1. Khái niệm đệ quy

Đệ quy là công cụ hữu hiệu trong khoa học máy tính và toán học để giải quyết nhiều bài toán thực tế. Các ngôn ngữ bậc cao như Pascal, C, ... đều hỗ trợ cài đặt thuật toán đệ quy.

Một đối tượng như thế nào gọi là đối tượng *đệ quy* ? Xét ví dụ sau

◊ *Ví dụ*. Tính $n!$

Nếu $n=0$, thì $0! = 1$.

Nếu $n > 0$, thì $n! = n.(n-1)!$

Như vậy, đối tượng là *đệ quy* nếu nó được định nghĩa bằng chính nó hoặc được xác định với điều kiện nào đó.

◊ *Ví dụ*. Tính ước số chung lớn nhất của hai số nguyên không âm a, b ($a \le b$).

Kí hiệu $UCLN(a,b)$ là ước số chung lớn nhất của a, b. Cách tính như sau

Nếu $a = 0$, thì $UCLN(0,b) = b$.

Nếu $a > 0$, thì $UCLN(a,b) = UCLN(b \bmod a, a)$.

Ví dụ, áp dụng thuật toán đệ quy tìm UCLN của 50 và 70:

= UCLN(50, 70) = UCLN(70 *mod* 50, 50)
= UCLN(20, 50) = UCLN(50 *mod* 20, 20)
= UCLN(10, 20) = UCLN(20 *mod* 10, 10)
= UCLN(0, 10) = 10

• ***Định nghĩa***. Một thuật toán được gọi là *đệ quy* nếu nó giải bài toán bằng cách rút gọn liên tiếp bài toán ban đầu tới bài toán tương tự nhưng có đầu vào nhỏ hơn.

2. Cài đặt thuật toán đệ quy

Trong các ngôn ngữ bậc cao (Pascal, C, ...) thuật toán đệ quy được cài đặt bằng thủ tục hoặc hàm, mà trong thân chương trình thủ tục và hàm lại được gọi lại với tham số đầu vào nhỏ hơn.

◊ *Ví dụ*. Tính $n!$

```
long  giaithua(int n)
{
   if  (n == 0)  return  1;
   else  return (n*giaithua(n-1));
}
```

◊ *Ví dụ*. Tính ước số chung lớn nhất của hai số nguyên không âm a, b ($a \le b$).

```
int  ucln(int a, int b)
{
      if  (a == 0)  return b;
      else  return  ucln(b % a, a);
```

```
}
```

◊ *Ví dụ*. Tính số *Fibonacci*.

Dãy số Fibonacci $f(0), f(1), ..., f(n), ...$ được định nghĩa như sau

Nếu $n = 0$, thì $f(0) = 0$.

Nếu $n = 1$, thì $f(1) = 1$.

Nếu $n > 1$, thì $f(n) = f(n–1) + f(n–2)$.

```
long  fib(int n)
{
   if  (n <= 1)  return n;
   else  return (fib(n-1)+fib(n-2));
}
```

◊ *Ví dụ*. *Bài toán tháp Hà Nội*

Bài toán này do *Edouard Lucas* đưa ra ở cuối thế kỷ 19 (Ông cũng là người đưa ra dãy Fibonacci). Bài toán phát biểu như sau. Có 3 cọc, cọc thứ nhất có n đĩa kích thước khác nhau xếp chồng nhau, đĩa nhỏ nằm trên đĩa lớn. Hãy chuyển các đĩa từ cọc thứ nhất sang cọc thứ ba, sử dụng cọc trung gian thứ hai, sao cho luôn đảm bảo đĩa nhỏ nằm trên đĩa lớn và mỗi lần chỉ được chuyển 1 đĩa. Tìm phương án di chuyển đĩa tối ưu.

Giả sử các cọc lần lượt kí hiệu là A, B, C và ta phải chuyển n đĩa từ cọc A sang cọc C với cọc trung gian B. Thuật toán di chuyển đệ quy như sau:

Nếu $n = 1$, thì chuyển 1 đĩa từ cọc A sang cọc C.

Nếu $n > 1$, thì

Di chuyển $(n–1)$ đĩa từ cọc A sang cọc B với cọc trung gian C;

Chuyển 1 đĩa từ cọc A sang cọc C;

Di chuyển $(n–1)$ đĩa từ cọc B sang cọc C với cọc trung gian A;

```
void ThapHaNoi(int n, char A, char B, char C)
{
   if (n==1)
   {
      printf("\n%s%c%s%c", "chuyển
       đĩa từ cọc ", A, " sang cọc", C);
   }
   else
      {
         ThapHaNoi(n-1, A, C, B);
         ThapHaNoi(1, A, B, C);
         ThapHaNoi(n-1, B, A, C);
      }
}
```

◊ *Ghi chú:* Số lần di chuyển đĩa là $2^n–1$. Khi $n = 64$, ta có số lần di chuyển đĩa là:

18 446 744 073 709 551 615

Nếu mỗi giây di chuyển 10 đĩa, thì cần gần 585 triệu thế kỷ mới chuyển xong 64 đĩa.

Nếu mỗi giây di chuyển 10 triệu đĩa, thì cần gần 585 thế kỷ mới chuyển xong 64 đĩa.

3. Khử đệ quy

Phương pháp:

Khử đệ quy thực chất là chúng ta phải làm công việc của một trình biên dịch đối với một thủ tục như sau:

Đặt tất cả các giá trị của các biến cục bộ và địa chỉ của chỉ thị kế tiếp vào ngăn xếp (*stack*), xác định các giá trị tham số cho thủ tục và chuyển tới (goto) vị trí bắt đầu thủ tục, thực hiện lần lượt từng câu lệnh.

Sau khi thủ tục hoàn tất thì nó phải lấy ra khỏi ngăn xếp địa chỉ trả về và các giá trị của các biến cục bộ, khôi phục các biến và chuyển tới địa chỉ trả về.

Cho thủ tục đệ quy P. Mỗi lần có lời gọi đệ quy đến P, giá trị hiện hành của các tham số, biến cục bộ được cất vào ngăn xếp.

Mỗi lần có sự quay về đệ quy về P, giá trị các tham số, biến cục bộ được khôi phục từ ngăn xếp.

Việc xử lý địa chỉ khứ hồi được thực hiện như sau: Giả sử thủ tục P chứa lệnh gọi đệ quy tại bước lệnh *k*, địa chỉ khứ hồi *k*+1 sẽ được lưu vào ngăn xếp và được dùng để quay về mức thực thi thủ tục P hiện hành.

Để dễ theo dõi chúng ta lấy ví dụ với bài toán cụ thể là bài toán duyệt cây. Giả sử có một cây nhị phân lưu trữ trong biến con trỏ ***root*** được định nghĩa:

```
typedef   <kiểu dữ liệu> InfoType;
struct  Node {
            InfoType info;
            struct Node *left;
            struct Node *right;
         };
typedef struct Node *NodePointer;
NodePointer root ;
```

Thủ tục duyệt cây đệ quy theo thứ tự NLR như sau:

```
void Traverse_NLR( NodePointer p)
{
   if  (p != NULL)
      {
         printf("%d", p->info);
```

```
         Traverse_NLR(p->left);
         Traverse_NLR(p->right);
      }
}
```

Thủ tục gọi là `Traverse_NLR(root);`

Trước hết có thể thấy rằng lệnh gọi đệ quy thứ hai có thể được khử dễ dàng bởi không có mã lệnh theo sau nó. Khi lệnh này thực hiện thì thủ tục `Traverse_NLR()` được gọi với tham số `p->right` và khi lệnh gọi này kết thúc thì thủ tục `Traverse_NLR` hiện hành cũng kết thúc. Thủ tục được viết lại (với lệnh *goto*) như sau:

```
void Traverse_NLR( NodePointer p)
{
   if  (p != NULL)
      {
         printf("%d", p->info);
         Traverse_NLR(p->left);
         p = p->right;
      }
}
```

Đây là kỹ thuật rất nổi tiếng được gọi là *khử đệ quy phần cuối* (*end-recursion removal*). Việc khử lần gọi đệ quy còn lại đòi hỏi phải làm nhiều việc hơn. Giống như một trình biên dịch chúng ta phải tổ chức một ngăn xếp (stack) để lưu trữ các biến cục bộ, các tham số, và sử dụng các thủ tục:

`push(p)`: Đẩy *p* vào stack;
Hàm `pop(&p)`: lấy phần tử từ stack đưa vào *p*.
Hàm `stackempty()`: Báo hiệu stack đã rỗng.

Ở đây không có giá trị trả về và chỉ có một biến cục bộ là *p* nên chúng ta sẽ nạp nó vào stack nếu chưa được xử lý và ở mỗi bước chúng ta lấy biến ở đỉnh stack ra để xử lý nó và các nút con tiếp theo của nó. Chương trình khử cả lời gọi đệ quy thứ hai sẽ như sau:

```
void Traverse_NLR( NodePointer p)
{
   if  (p != NULL)
      {
         0:;
         if p = NULL goto 1
         printf("%d", p->info);
         push(p)
```

```
         p = p->left;
         goto 0;// chỉ thị goto 0 thứ nhất
         2:;
         p = p->right;
         goto 0; // chỉ thị goto 0 thứ hai
         1:;
         if stackempty() exit;
         pop(&p);
         goto 2;
      }
}
```

Thủ tục trên chỉ là diễn giải thô của ý tưởng để các bạn dễ hiểu, vì thế nó chứa các chỉ thị `goto` còn rườm rà.

Chúng ta sẽ viết lại một cách có cấu trúc hơn. Trước tiên đoạn chương trình giữa nhãn *2* và chỉ thị `goto 0` thứ 2 được bao quanh bởi các chỉ thị `goto 0` nên có thể khử đi một cách đơn giản, khử đi nhãn *2* cùng chỉ thị `goto 0` thứ nhất. Kế đó, đoạn chương trình giữa nhãn *0* và chỉ thị `goto 0` thứ hai được thay bằng vòng lặp `while`. Vòng lặp được gán nhãn 0, lệnh `push(p)` và `p = p->right` thay bằng lệnh `push(p->right)` và chỉ thị `goto 2` thay bằng `goto 0`. Ta có thủ tục sau:

```
void Traverse_NLR( NodePointer p)
{
   if  (p != NULL)
      {
         0:;
         while (p != NULL)
         {
            printf("%d", p->info);
            push(p->right);
            p = p->left;
         };
         if stackempty() exit;
         pop(&p);
         goto 0;
      }
}
```

Đến đây ta có thể thay nhãn *0* và chỉ thị `goto 0` bằng vòng lặp `do … while` và nhận được thủ tục không có `goto` như sau:

```
void Traverse_NLR( NodePointer p)
{
   if  (p != NULL)
```

```
   {
      push(p)
      do
      {
         pop(&p);
         while (p != NULL)
         {
            printf("%d", p->info);
            push(p->right);
            p = p->left;
         };
      }
      while (!stackempty());
}
```

Đây là thủ tục duyệt cây không đệ quy chuẩn. Vòng lặp `while` có thể đơn giản hóa bằng cách dùng ngăn xếp như sau:

```
void Traverse_NLR( NodePointer p)
{
   if  (p != NULL)
   {
      push(p)
      do
      {
         pop(&p);
         if  (p != NULL)
         {
            printf("%d", p->info);
            push(p->right);
            push(p->left);
         };
      }
      while (!stackempty());
}
```

Thủ tục này ưu điểm hơn thủ tục đệ quy ở chỗ (1) nó có thể chạy trong môi trường lập trình bất kỳ mà không cần hỗ trợ đệ quy và (2) nó chạy hiệu quả hơn thủ tục đệ quy.

Cuối cùng, để tránh trường hợp nạp các nút rỗng vào stack ta có thủ tục duyệt cây không đệ quy chuẩn như sau.

```
void Traverse_NLR( NodePointer p)
{
   if  (p != NULL)
   {
```

```
      push(p)
      do
      {
         pop(&p);
         printf("%d", p->info);
         if  (p->right != NULL)  push(p->right);
         if  (p->left != NULL)  push(p->left);
      }
      while (!stackempty());
}
```

IV. PHÂN LOẠI THUẬT TOÁN

Độ phức tạp của thuật toán chính là yếu tố cơ sở để phân loại vấn đề-bài toán. Một cách tổng quát, mọi bài toán đều có thể chia làm 2 lớp lớn là : giải được và không giải được. Lớp giải được chia làm 2 lớp con. Lớp con đầu tiên là các *bài toán có độ phức tạp đa thức*: *nghĩa là bài toán có thể giải được bằng thuật toán có độ phức tạp đa thức* (*hay nói ngắn gọn* : *lớp đa thức P*) được xem là có lời giải thực tế. Lớp con thứ hai là những bài toán có độ phức tạp không phải là đa thức mà lời giải của nó được xem là thực tế chỉ cho những số liệu đầu vào có chọn lựa cẩn thận và tương đối nhỏ. Cuối cùng là những bài toán thuộc loại NP chưa thể phân loại một cách chính xác là thuộc lớp bài toán có độ phức tạp đa thức hay có độ phức tạp không đa thức.

1. Lớp bài toán có độ phức tạp đa thức P

Các bài toán thuộc lớp này có độ phức tạp là $O(n^k)$ hoặc nhỏ hơn $O(n^k)$. Chẳng hạn như các bài toán có độ phức tạp là $O(n\log_2 n)$ được xem là các bài toán thuộc lớp đa thức vì $n\log_2 n$ bị chặn bởi n^2 ($n\log_2 n \le n^2$ với mọi $n>0$). Như vậy các bài toán có độ phức tạp hằng $O(1)$, phức tạp tuyến tính $O(n)$ và logarit $O(n\log_a n)$ ($a>0$) đều là các bài toán thuộc lớp đa thức. Còn các bài toán có độ phức tạp lũy thừa $O(a^n)$ hoặc giai thừa $O(n!)$ là không thuộc lớp đa thức.

Tuy độ phức tạp chỉ là số đo về độ tăng của chi phí ứng với độ tăng của dữ liệu đầu vào nhưng nó cũng cho chúng ta có một đánh giá tương đối về thời gian thi hành thuật toán. Các thuật toán thuộc lớp đa thức được xem là các bài toán có lời giải thực tế. *Lời giải thực tế* được hiểu rằng là chi phí về mặt thời gian và không gian cho việc giải bài toán là chấp nhận được trong điều kiện hiện tại. Bất kỳ một bài toán nào không thuộc lớp này thì đều có chi phí rất lớn.

Có thể giải được hay không?

Người ta đã ước tính thời gian cần thiết để giải một mật mã được mã hóa bằng khóa 128-bit là trên 1 triệu năm với điều kiện làm việc trên các siêu máy tính mạnh nhất hiện nay!

Chính vì lý do này, một bài toán được xem là có thể giải được trên thực tế hay không phụ thuộc vào độ phức tạp của bài toán đó có phải là đa thức hay không.

2. Lớp bài toán có độ phức tạp không đa thức

Thật không may mắn, nhiều bài toán thực sự có lời giải lại không thuộc lớp của bài toán đa thức. Ví dụ : *cho một tập hợp có n phần tử, hãy liệt kê tất cả các tập con khác rỗng của tập hợp này.* Bằng toán học, người ta đã chứng minh được rằng số tập con của một tập hợp có n phần tử là 2^n. Lời giải tuy đã có nhưng khi thể hiện lời giải này bằng bất kỳ thuật toán nào thì phải tốn ít nhất 2^n

bước. Dễ thấy rằng độ phức tạp của bài toán này cũng cỡ $O(2^n)$. Như vậy bài toán này không thuộc lớp của bài toán đa thức. Với *n* vào khoảng 16, số bước cần thiết chỉ khoảng vài chục ngàn là hoàn toàn giải được trên các máy tính hiện nay. Nhưng khi số phần tử lên đến 32 thì ta đã tốn một số bước lên đến 4 tỷ, chỉ thêm một phần tử nữa thôi, chúng ta đã tốn 8 tỷ bước! Với số lượng bước như vậy, dù chạy trên một siêu máy tính cũng phải tốn một thời gian đáng kể! Các bài toán không thuộc lớp đa thức chỉ giải được với một độ lớn dữ liệu đầu vào nhất định.

3. Lớp bài toán NP

Chúng ta đều biết rằng *tính tất định* là một trong các đặc tính quan trọng của thuật toán. Nghĩa là mỗi bước của thuật toán phải được xác định duy nhất và có thể thực thi được. Nếu có sự phân chia trường hợp tại một bước thì thông tin tại bước đó phải đầy đủ để thuật toán có thể *tự quyết định* chọn lựa trường hợp nào. Trong mục này, ta tạm gọi các thuật toán thỏa mãn tính tất định là các *thuật toán tự quyết*.

Vậy thì điều gì sẽ xảy ra nếu ta đưa ra một "thuật toán" có tính không tự quyết. Nghĩa là tại một bước của "thuật toán", ta đưa ra một số trường hợp chọn lựa nhưng không cung cấp đầy đủ thông tin để "thuật toán" tự quyết định. Thật ra, trong cuộc sống, những "thuật toán" thuộc loại này rất hay được áp dụng. Chẳng hạn ta có một lời chỉ dẫn khi đi du lịch : *"Khi đi hết khu vườn này, bạn hãy chọn một con đường mà bạn cảm thấy thích. Tất cả đều dẫn đến vòng quay mặt trời."*. Nếu là khách du lịch, bạn sẽ cảm thấy bình thường. Nhưng máy tính thì không! Nó không thể thực thi những hướng dẫn không rõ ràng như vậy!

Đến đây, lập tức sẽ có một câu hỏi rằng "Tại sao lại đề cập đến những thuật toán có tính không tự quyết dù máy tính không thể thực hiện một thuật toán như vậy ?". Câu trả lời là, khi nghiên cứu về thuật toán không tự quyết, dù không dùng để giải bài toán nào đi nữa, chúng ta sẽ thu được những hiểu biết về hạn chế của những thuật toán tự quyết thông thường.

Đến đây, ta hãy xem sự khác biệt về độ phức tạp của một thuật toán tự quyết và không tự quyết để giải quyết cho cùng một vấn đề.

- **Bài toán người bán hàng – TSP (Travel Salesman Problem)**

Một nhân viên phân phối hàng cho một công ty được giao nhiệm vụ phải giao hàng cho các đại lý của công ty, sau đó trở về công ty. Vấn đề của người nhân viên là làm sao đi giao hàng cho tất cả đại lý mà không tiêu quá số tiền đổ xăng mà công ty cấp cho mỗi ngày. Nói một cách khác, làm sao đừng đi quá một số lượng cây số nào đó.

Một lời giải cổ điển cho bài toán này là liệt kê một cách có hệ thống từng con đường có thể đi, so sánh chiều dài mỗi con đường tìm được với chiều dài giới hạn cho đến lúc tìm được một con đường phù hợp hoặc đã xét hết tất cả các con đường có thể đi. Tuy nhiên, cách giải quyết này có độ phức tạp không phải đa thức. Bằng toán học, người ta đã chứng minh được rằng độ phức tạp của thuật toán này là O(n!). Như vậy, với số đại lý lớn thì thuật toán trên được xem là không thực tế. Bây giờ, chúng ta xem qua một thuật toán không tự quyết.

(1**)** Chọn một con đường có thể và tính chiều dài của nó.

(2) Nếu chiều dài này không lớn hơn giới hạn thì báo là thành công, ngược lại báo chọn lựa sai.

Quan điểm của ta trong cách giải quyết này là nếu chọn sai thì là do lỗi của người chọn chứ không phải lỗi của thuật toán.

Theo thuật toán này thì chi phí để tính chiều dài của con đường được chọn sẽ tỷ lệ với số đại lý; chi phí để so sánh chiều dài quãng đường với giới hạn cho phép thì không liên quan đến số thành phố. Như vậy, chi phí của thuật toán này là một hàm có dạng T(n) = $a.n+b$ với n là số đại lý và a,b là các hằng số. Ta kết luận rằng, độ phức tạp của thuật toán này là O(n) hay độ phức tạp thuộc lớp đa thức.

Như vậy, nếu dùng thuật toán tự quyết thì bài toán người bán hàng sẽ có độ phức tạp không thuộc lớp đa thức, còn nếu dùng thuật toán không tự quyết thì bài toán sẽ có độ phức tạp đa thức.

- **Định nghĩa**

Một bài toán khi được giải bằng một thuật toán không tự quyết mà có độ phức tạp thuộc lớp đa thức thì được gọi là một bài toán *đa thức không tự quyết* hay viết tắt là bài toán NP.

Theo định nghĩa trên thì bài toán người bán hàng là bài toán thuộc lớp NP. Cho đến nay người ta chưa chứng minh được rằng **tồn tại** hay **không** một thuật toán tự quyết có độ phức tạp đa thức cho bài toán người bán hàng. Vì vậy, bài toán này (là một bài toán NP) chưa thể xếp được vào lớp đa thức hay không đa thức. Do đó, lớp bài toán NP chưa thể phân loại là thuộc lớp đa thức hay không.

Dĩ nhiên, lớp bài toán NP cũng chứa những bài toán thuộc lớp đa thức thực sự, bởi vì nếu một bài toán được giải bằng thuật toán tự quyết có độ phức tạp đa thức thì chắc chắn khi dùng thuật toán không tự quyết thì cũng sẽ có độ phức tạp đa thức.

Một số bài toán NP:

• ***Bài toán thỏa mãn mạch lôgic* – CSP (*Circuit Satisfiability Problem*)**

Cho biểu thức lôgic dạng

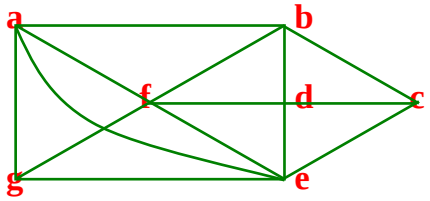
$$(x_1 \vee x_3 \vee x_5) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_3 \vee x_4 \vee x_5) \wedge (x_2 \vee \neg x_3 \vee x_5)$$

Bài toán CSP là xác định xem có tồn tại phép gán các giá trị lôgic cho các biến x_i, i = 1, …, 5, sao cho biểu thức luôn nhận giá trị đúng (true).

• ***Bài toán chu trình Hamilton* – HCP (*Hamilton Cycle Problem*)**

Cho đồ thị G=(V,E). *Chu trình Hamilton* là chu trình qua mọi đỉnh đồ thị, mỗi đỉnh đúng một lần.

◊ *Ví dụ*. Xét đồ thị



Đồ thị có chu trình Hamilton

$$a \to b \to c \to d \to e \to f \to g \to a$$

Phát biểu bài toán: Tìm chu trình Hamilton trong một đồ thị.

• **Khái niệm quy dẫn**

Bài toán P1 được coi là *quy dẫn* được về bài toán P2, ký hiệu P1⇒P2, nếu bất kỳ giải thuật nào giải được P2 thì cũng có thể được biến đổi sau thời gian đa thức thành giải thuật dùng để giải P1.

• **Bài toán NP-đầy đủ (NP-complete)**

Định nghĩa. Một bài toán A được gọi là **NP-đầy đủ** (**NP-Complete**), nếu như A là một bài toán trong **NP** và mọi bài toán trong **NP** đều có thể quy dẫn về A.

Có những bài toán thuộc lớp NP, nhưng không biết có thuộc lớp P hay không.

• Giả thiết: "***Lớp bài toán NP khác lớp bài toán P***".

Thực tế không tồn tại thuật toán tốt có độ phức tạp đa thức để giải một trong các bài toán trên (bài toán người bán hàng, bài toán thỏa mãn mạch loogic, bài toán tìm chu trình Hamilton, …) là minh chứng tin cậy cho giả thiết : NP ≠ P.

• **Bài toán NP-khó (NP-Hard)**

Định nghĩa. Một bài toán H được gọi là **NP-khó** (**NP-hard**) nếu mọi bài toán trong **NP** đều có thể quy dẫn về H.

Từ định nghĩa trên, ta nhận thấy

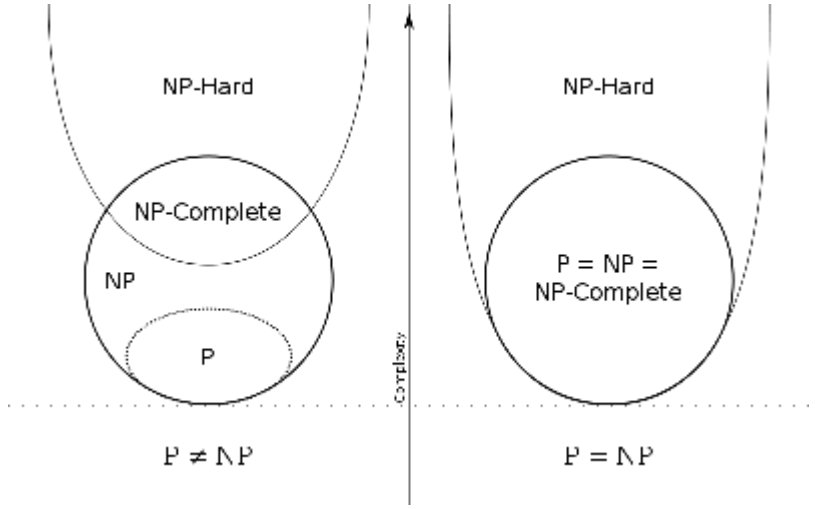
- Bài toán H ít nhất là khó bằng NP, nhưng H không nhất thiết là trong NP.

- Nếu P ≠ NP, thì các bài toán NP-khó không thể giải được trong thời gian đa thức, nhưng nếu P=NP, thì vẫn chưa thể biết một bài toán NP-khó có thể có thể giải được trong thời gian đa thức hay không.

Một cách không hình thức, có thể nói rằng nếu ta có thể giải được một cách hiệu quả một bài toán **NP-khó** cụ thể, thì ta cũng có thể giải hiệu quả bất kỳ bài toán trong **NP** bằng cách sử dụng thuật toán giải bài toán **NP-khó** như một chương trình con.

Từ định nghĩa bài toán **NP-khó** có thể suy ra rằng mỗi bài toán **NP-đầy đủ** đều là **NP-khó**. Tuy nhiên một bài toán **NP-khó** không nhất thiết phải là **NP-đầy đủ**, vì nó có thể không thuộc lớp NP.

Ta có thể minh họa mối quan hệ giữa các lớp bài toán bằng hình vẽ sau:



Năm 1971 Stephen Cook đã chứng minh bài toán CSP là bài toán NP-đầy đủ đầu tiên.

• ***Định lý Cook* (1971)**. Nếu tồn tại thuật toán thời gian đa thức để giải bài toán thỏa mãn mạch lôgic CSP, thì tất cả bài toán lớp NP giải được trong thời gian đa thức.

Các bài toán người bán hàng TSP, bài toán chu trình Hamilton HCP cũng là những bài toán NP-đầy đủ.

Các bài toán sau cũng là những bài toán NP-đầy đủ.

- *Bài toán ba lô*
- *Cây Steiner nhỏ nhất*
- *Bài toán tập đỉnh độc lập lớn nhất*
- *Bài toán đồ thị con đẳng cấu*
- *Bài toán tô màu đồ thị*
- *Bài toán phân hoạch số*. Cho tập số nguyên, có thể phân hoạch chúng thành hai tập có tổng bằng nhau hay không.

- *Bài toán quy hoạch tuyến tính nguyên*. Cho bài toán quy hoạch tuyến tính. Tồn tại hay không một phương án tối ưu nguyên ?

• Có những bài toán là NP-khó nhưng không phải NP-đầy đủ, chẳng hạn *bài toán dừng*. Bài toán này yêu cầu xác định xem với một chương trình và dữ liệu vào cho trước, liệu chương trình có chạy mãi mãi hay không.

Từ phần trình bày trên, ta thấy có rất nhiều bài toán ứng dụng quan trọng thuộc vào lớp **NP-khó**, và vì thế khó hy vọng xây dựng được thuật toán đúng hiệu quả để giải chúng. Do đó, một trong những hướng phát triển thuật toán giải các bài toán như vậy là xây dựng các thuật toán gần đúng, thuật toán ngẫu nhiên (phương pháp Monte Carlo), thuật toán heuristic, metaheuristic (thuật toán di truyền, mô phỏng luyện kim, thuật toán đàn kiến, …).

IV. TỔNG QUAN KỸ THUẬT THIẾT KẾ THUẬT TOÁN

Các kỹ thuật thiết kế thuật toán có thể phân thành các nhóm sau:

Kỹ thuật chia để trị,

Kỹ thuật quay lui,

Kỹ thuật nhánh cận,

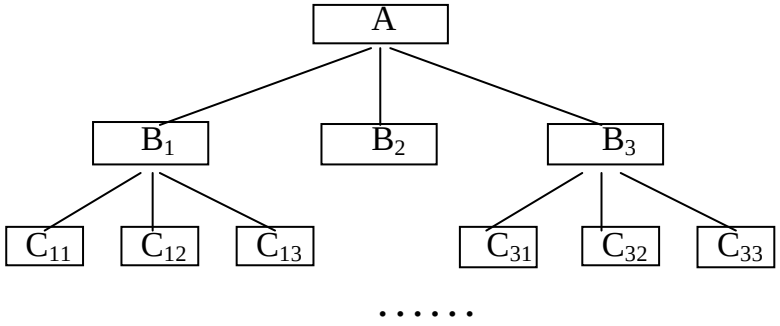
Kỹ thuật tham lam,

Kỹ thuật quy hoạch động.

1. Kỹ thuật chia để trị

Kỹ thuật chia để trị để giải quyết một vấn đề là cách phân tích tổng quát toàn bộ vấn đề. Chia bài toán lớn thành những bài toán nhỏ. Những bài toán nhỏ lại được chia thành những bài toán nhỏ hơn ... Cuối cùng mới đi vào giải quyết chi tiết từng bài toán cụ thể.

Như vậy lời giải của bài toán sẽ được tổ chức theo cấu trúc phân cấp sau: Bài toán lớn A được phân thành các bài toán nhỏ B_1, B_2, B_3, … . Các bài toán B_i lại được phân thành các bài toán nhỏ hơn C_{i1}, C_{i2}, C_{i3}, ... như hình sau



Cách phân tích tổ chức như vậy giúp ta có cái nhìn tổng quát toàn bộ vấn đề, đồng thời có thể phân chia công việc cụ thể một cách chính xác đầy đủ. Mỗi bài toán có thể coi là một mô-đun công việc độc lập.

• Kỹ thuật giảm (kích thước) để trị.

• Yêu cầu:

Cần phải giải bài toán có kích thước *n*.

• Phương pháp:

Ta chia bài toán ban đầu thành một số bài toán con đồng dạng với bài toán ban đầu có kích thước nhỏ hơn *n*.

Giải các bài toán con được các lời giải bộ phận.

Tổng hợp lời giải bộ phận suy ra lời giải của bai toán ban đầu.

•.Chú ý:

Đối với từng bài toán con, ta lại chia chúng thành các bài toán con nhỏ hơn nữa.

Quá trình phân chia này sẽ dừng lại khi kích thước bài toán đủ nhỏ, gọi là bài toán cơ sở, mà ta có thể giải dễ dàng.

◊ *Ví dụ*. Thuật toán tìm kiếm nhị phân.

Bài toán: Cho phần tử x và danh sách có thứ tự tăng dần $a[1], a[2], ..., a[n]$. Tìm phần tử x trong danh sách $a[1], a[2], ..., a[n]$.

Cách tìm:

Tìm phần tử ở giữa $a[m]$, $m = (1+n)/2$.

Nếu $x = a[m]$, thì $a[m]$ là phần tử cần tìm. Kết thúc.

Nếu $x < a[m]$, thì tìm x trong nửa danh sách từ $a[1]$ đến $a[m-1]$.

Nếu $x > a[m]$, thì tìm x trong nửa danh sách từ $a[m+1]$ đến $a[n]$.

2. Kỹ thuật quay lui

Giả sử một phương án (lời giải) của bài toán gồm n thành phần $x[1], x[2], \ldots, x[n]$.

Ý tưởng kỹ thuật quay lui là xây dựng lần lượt từng thành phần của lời giải qua n bước, $i=1..n$. Bước i chọn thành phần $x[i]$ bằng cách thử tất cả các khả năng có thể, $i=1..n$.

Ở mỗi bước i, có một số lựa chọn cho thành phần i.

Chọn một giá trị nào đó cho thành phần i.

Gọi đệ quy để tìm thành phần $(i+1)$.

Sau khi thực hiện đủ n bước ta được một lời giải.

Để tìm lời giải khác, ở mỗi bước $i=1..n$, quay lui lại 1 bước, chọn giá trị khác cho thành phần i.

Trong bài toán tối ưu, khi tìm được lời giải, so sánh với lời giải tốt nhất trước đó để chọn lời giải tối ưu.

Nếu phải duyệt qua tất cả các lời giải để tìm lời giải tối ưu, thì thuật toán được gọi là *thuật toán quay lui vét cạn*.

◊ *Ví dụ*. Bài toán liệt kê hoán vị n phần tử.

3. Kỹ thuật nhánh cận

Kỹ thuật nhánh cận cải tiến kỹ thuật quay lui vét cạn như sau:

Tại mỗi bước, ta sẽ xem xét xem có nên đi bước kế tiếp nữa hay không.

Việc xem xét dựa trên *khái niệm cận* của bước hiện hành.

4. Kỹ thuật tham lam

• Mục đích: Tìm một lời giải tốt trong thời gian chấp nhận được (độ phức tạp đa thức thay vì lũy thừa).

• Ý tưởng: Chia quá trình tìm lời giải thành nhiều bước như kỹ thuật quay lui.
• Với mỗi bước

Sắp xếp các lựa chọn cho bước đó theo thứ tự nào đó "có lợi" (tăng dần hoặc giảm dần tuy theo cách lập luận).

Chọn lựa chọn tốt nhất rồi đi tiếp bước kế (không quay lui).

5. Kỹ thuật quy hoạch động

• Mục đích: Cải tiến thuật toán chia để trị hoặc quay lui vét cạn để giảm thời gian thực hiện.
• Ý tưởng: Lưu trữ các kết quả của các bài toán con trong bảng.
• Thiết kế giải thuật bằng kỹ thuật quy hoạch động gồm các bước:

Phân tích bài toán dùng kỹ thuật chia để trị/quay lui.

Chia bài toán thành các bài toán con.

Tìm quan hệ, dạng công thức truy hồi, giữa lời giải của bài toán lớn và lời giải của các bài toán con.

Lập bảng phương án lưu kết quả bài toán con.

Truy vết tìm lời giải.