BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC NÔNG NGHIỆP HÀ NỘI

VŨ KIM THÀNH

TOÁN RỜI RẠC

(Giáo trình dành cho sinh viên ngành công nghệ thông tin)

Hà nội 2008

MỤC LỤC

LỜI NÓI ĐẦU

Toán Rời rạc (Discrete mathematics) là môn toán học nghiên cứu các đối tượng rời rạc. Nó được ứng dụng trong nhiều ngành khoa học khác nhau, đặc biệt là trong tin học bởi quá trình xử lý thông tin trên máy tính thực chất là một quá trình rời rạc.

Phạm vi nghiên cứu của Toán Rời rạc rất rộng, có thể chia thành các môn học khác nhau. Theo quy định của chương trình môn học, giáo trình này đề cập đến các lĩnh vực: Thuật toán và bài toán đếm; Lý thuyết đồ thị; Đại số Logic và được chia thành 8 chương:

- Chương 1 đề cập đến một trong các vấn đề cơ bản nhất của *Thuật toán* đó là độ phức tạp về thời gian của thuật toán.
- Chương 2 nói về các nguyên lý cơ bản của *Bài toán đếm.*
- Các chương 3, 4, 5 và 6 trình bày về *Lý thuyết đồ thị và các ứng dụng*. Đây là phần chiếm tỷ trọng nhiều nhất của giáo trình. Trong đó có các chương về các khái niệm cơ bản của đồ thị, các đồ thị đặc biệt như đồ thị Euler, đồ thị Hamilton, đồ thị phẳng, Cây cùng các ứng dụng của các đồ thi đặc biệt này. Riêng chương 6 dành cho một vấn đề trọng là một số bài toán tối ưu trên đồ thị hoặc bài toán tối ưu được giải bằng cách ứng dụng lý thuyết đồ thị.
- Chương 7 là các kiến thức cơ bản về *Đại số Boole,* một công cụ hữu hiệu trong việc thiết kế các mạch điện, điện tử.

Cuối giáo trình là phụ chương: *Những khái niệm cơ bản về toán Logic* để người học có thể tự nghiên cứu thêm về Toán Logic.

Trong mỗi chương chúng tôi cố gắng trình bày các kiến thức cơ bản nhất của chương đó cùng các thí dụ minh họa cụ thể. Vì khuôn khổ số tiết học nên chúng tôi lược bỏ một số chứng minh phức tạp. Cuối mỗi chương đều có các bài tập để người học ứng dụng, kiểm chứng các lý thuyết đã học, đồng thời cũng cung cấp một số đáp số của các bài tập đã cho.

Cũng cần nói thêm rằng toán Rời rạc không chỉ được ứng dụng trong tin học mà còn được ứng dụng trong nhiều ngành khoa học khác. Bởi vậy giáo trình cũng có ích cho những ai cần quan tâm đến các ứng dụng khác của môn học này.

Tác giả xin chân thành cảm ơn các bạn đồng nghiệp đã động viên và góp ý cho việc biên soạn giáo trình này. Đặc biệt chúng tôi xin cảm ơn Nhà giáo ưu tú Nguyễn Đình Hiền đã hiệu đính và cho nhiều ý kiến đóng góp bổ ích và thiết thực.

Vì trình độ có hạn và giáo trình được biên soạn lần đầu nên không tránh khỏi các thiếu sót. Tác giả rất mong nhận được các ý kiến đóng góp của các đồng nghiệp và bạn đọc về các khiếm khuyết của cuốn sách.

TÁC GIẢ

CHƯƠNG1.

THUẬT TOÁN

1. Định nghĩa

Thuật toán (algorithm) là một dãy các quy tắc nhằm xác định một dãy các thao tác trên các đối tượng sao cho **sau một số hữu hạn** bước thực hiện sẽ **đạt được mục tiêu đặt ra.**

Từ định nghĩa của thuật toán cho thấy các đặc trưng (tính chất) cơ bản của thuật toán là:

a. Yếu tố vào, ra:

- Đầu vào (Input): Mỗi thuật toán có một giá trị hoặc một bộ giá trị đầu vào từ một tập xác định đã được chỉ rõ.
- Đầu ra (Output): Từ các giá trị đầu vào, thuật toán cho ra các giá trị cần tìm gọi là kết quả của bài toán.

b. Tính dừng:

Sau một số hữu hạn bước thao tác thuật toán phải kết thúc và cho kết quả .

c. Tính xác định:

Các thao tác phải rõ ràng, cho cùng một kết quả dù được xử lý trên các bộ xử lý khác nhau (hai bộ xử lý trong cùng một điều kiện không thể cho hai kết quả khác nhau).

d. Tính hiệu quả

Sau khi đưa dữ liệu vào cho thuật toán hoạt động phải đưa ra kết quả như ý muốn.

e. Tính tổng quát

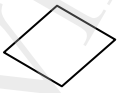
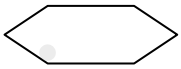
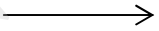
Thuật toán phải được áp dụng cho mọi bài toán cùng dạng chứ không phải chỉ cho một tập đặc biệt các giá trị đầu vào.

Có nhiều cách mô tả thuật toán như: Dùng ngôn ngữ tự nhiên; dùng lưu đồ (sơ đồ khối); dùng một ngôn ngữ lập trình nào đó (trong giáo trình này dùng loại ngôn ngữ mô tả gần như ngôn ngữ lập trình Pascal và gọi là phỏng Pascal); …

2. Mô tả thuật toán bằng lưu đồ

Sau khi có thuật toán để giải bài toán, trước khi chuyển sang ngôn ngữ lập trình, người ta thường phải thể hiện thuật toán dưới dạng sơ đồ. Sơ đồ đó gọi là lưu đồ của thuật toán. Các ký hiệu quy ước dùng trong lưu đồ được trình bày trong bảng 1.

Bảng 1. Các ký hiệu quy ước dung trong lưu đồ thuật toán

Tên ký hiệu	Ký hiệu	Vai trò của ký hiệu
Khối mở đầu hoặc kết thúc		Dùng để mở đầu hoặc kết thúc thuật toán
Khối vào ra		Đưa dữ liệu vào và in kết quả
Khối tính toán		Biểu diễn các công thức tính toán và thay đổi giá trị các đối tượng
Khối điều kiện		Kiểm tra các điều kiện phân nhánh
Chương trình con		Gọi các chương trình con
Hướng đi của thuật toán		Hướng chuyển thông tin, liên hệ giữa các khối

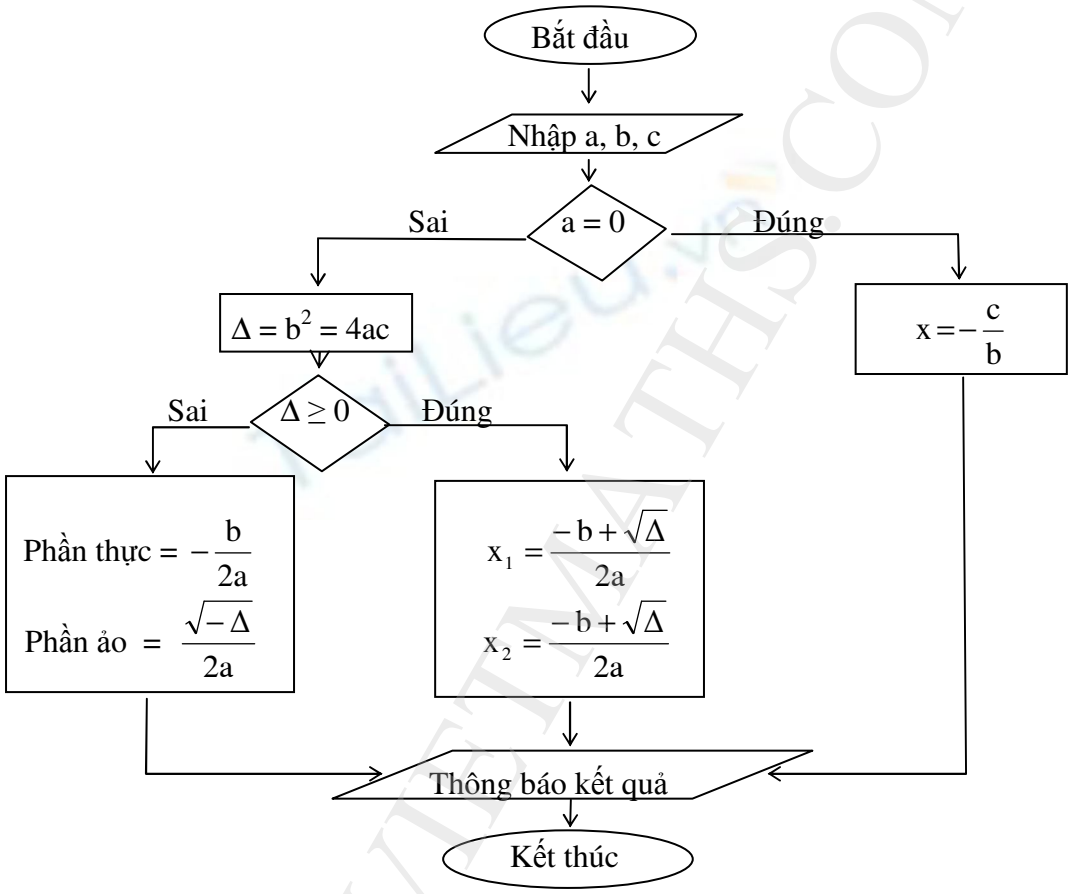
Thí dụ: Thuật toán giải phương trình bậc hai $ax^2 + bx + c = 0$ gồm các bước sau:

1) Xác định các hệ số a, b, c (thông tin đầu vào)
2) Kiểm tra hệ số a:

- Nếu $a = 0$: Phương trình đã cho là phương trình bậc nhất, nghiệm là: $x = -\frac{c}{b}$.
- Nếu $a \neq 0$: Chuyển sang bước 3.

3) Tính biệt thức $\Delta = b^2 - 4ac$.
4) Kiểm tra dấu của biệt thức Δ
   - Nếu $\Delta \geq 0$: Phương trình có nghiệm thực
   - Nếu $\Delta < 0$: Phương trình có nghiệm phức
5) In kết quả

Lưu đồ của thuật toán được trình bày trong hình 1



Hình 1. Lưu đồ giải phương trình bậc hai

3. Mô tả thuật toán bằng ngôn ngữ phỏng Pascal

Để giải bài toán trên máy tính điện tử phải viết chương trình theo một ngôn ngữ lập trình nào đó (Pascal, C, Basic, ...). Mỗi ngôn ngữ lập trình có một quy tắc cấu trúc riêng. Để thay việc mô tả thuật toán bằng lời, có thể mô tả thuật toán bằng các cấu trúc lệnh tương tự như ngôn ngữ lập trình Pascal và gọi là ngôn ngữ phỏng Pascal.

Các câu lệnh chính dùng để mô tả thuật toán gồm có: Procedure hoặc Function; câu lệnh gán; các câu lệnh điều kiện; các vòng lặp. Ngoài ra khi cần giải thích các câu lệnh bằng lời, có thể để các lời giải thích trong dấu (* ... *) hoặc {…}.

Nghĩa là ngôn ngữ phỏng Pascal hoàn toàn tương tự ngôn ngữ lập trình Pascal, nhưng không có phần khai báo. Tuy nhiên, phải nêu rõ đầu vào (Input) và đầu ra (output) của thuật toán.

3.1. Câu lệnh Procedure (thủ tục) hoặc Function (hàm)

Đứng ngay sau câu lệnh này là tên của thủ tục hoặc tên hàm. Các bước thực hiện của thuật toán được mô tả trong thủ tục (hàm) được bắt đầu bởi từ khóa begin và kết thúc bởi từ khóa end.

Thí dụ 1

```
Function Max(a, b, c)  (* Hàm tìm số lớn nhất trong 3 số a, b, c *)
Begin
 (* thân hàm*)
End;
```

Thí dụ 2

```
Procedure Giai_phuong_trình_bac_hai (* Thủ tục giải phương trình bậc hai *)
 Begin
 (* thân thủ tục *)
 End;
```

Chú ý rằng, khi mô tả thuật toán bằng function, trước khi kết thúc (end) thuật toán phải trả về (ghi nhận) giá trị của function đó.

3.2. Câu lệnh gán

Dùng để gán giá trị cho các biến. Cách viết:

Tên biến := giá trị gán

Thí dụ:

```
x := a;       (*biến x được gán giá trị a*)
max := b;   (*biến max được gán giá trị b*)
```

3.3. Khối câu lệnh tuần tự

Được mở đầu bằng từ khóa **begin** và kết thúc bằng **end** như sau:

```
begin
   Câu lệnh 1;
   Câu lệnh 2;
   ... .....
   Câu lệnh n;
end;
```

Các lệnh được thực hiện tuần tự từ câu lệnh thứ nhất đến câu lệnh cuối cùng.

3.4. Câu lệnh điều kiện

Có hai dạng: dạng đơn giản và dạng lựa chọn.

a. Dạng đơn giản: Cách viết:

if <điều kiện> **then** câu lệnh hoặc khối câu lệnh;

Khi thực hiện, điều kiện được kiểm tra, nếu điều kiện thỏa mãn thì câu lệnh (khối câu lệnh) được thực hiện, nếu điều kiện không thỏa mãn thì lệnh bị bỏ qua.

b. Dạng lựa chọn: Cách viết:

if <điều kiện> **then** câu lệnh hoặc khối câu lệnh 1 **else** câu lệnh hoặc khối câu lệnh 2;

Khi thực hiện, điều kiện được kiểm tra, nếu điều kiện thỏa mãn thì câu lệnh (khối câu lệnh) 1 được thực hiện, nếu điều kiện không được thỏa mãn thì câu lệnh (khối câu lệnh) 2 được thực hiện.

Thí dụ 1. Thuật toán tìm số lớn nhất trong 3 số thực a, b, c.

- Đầu tiên cho max = a;
- So sánh max với b, nếu b > max thì max = b;
- So sánh max với c, nếu c > max thì max = c.

```
Function  max(a,b,c)
 Input:    3 số thực a,b,c;
 Output:  Số lớn nhất trong 3 số đã nhập;
   Begin
      x := a;
      if  b > x  then  x:= b;
      if  c > x  then  x:= c;
      max := x;
    End;
```

Thí dụ 2. Thuật toán giải phương trình bậc hai $ax^2 + bx + c = 0$

```
Procedure  Giai_phuong_trinh_bac2;
 Input:    Các hệ số a, b, c;
 Output:  Nghiệm của phương trình;
          begin
             if  a = 0  then  x := -c/b;
             if  a ≠ 0 then
                begin
                   Δ := b^2 – 4ac;
                   if  Δ ≥ 0  then
                               begin
                                  x1 = (– b – √Δ )/2a ;
                                  x2 = (– b + √Δ )/2a;
                               end
                           else
                               begin
                                  Phần thực := -b/2a;
                                  Phần ảo := ( √(– Δ) )/2a;
                                end;
                end;
          end;
```

3.5. Các câu lệnh lặp

Có hai loại: Loại có bước lặp xác định và loại có bước lặp không xác định.

a. Loại có bước lặp xác định: Cách viết như sau:

for biến điều khiển := giá trị đầu **to** giá trị cuối **do** câu lệnh hoặc khối câu lệnh;

Khi thực hiện, biến điều khiển được kiểm tra, nếu biến điều khiển nhỏ hơn hoặc bằng giá trị cuối thì câu lệnh (khối câu lệnh) được thực hiện. Tiếp đó biến điều khiển sẽ tăng thêm 1 đơn vị và quá trình được lặp lại cho đến khi biến điều khiển lớn hơn giá trị cuối thì vòng lặp dừng và cho kết quả. Như vậy hết vòng lặp for số bước lặp là giá trị cuối (của biến điều khiển) trừ giá trị đầu cộng một.

Thí dụ: Tìm giá trị lớn nhất của dãy số $a_1, a_2, \ldots, a_n$.

Thuật toán: Đầu tiên cho giá trị lớn nhất (max) bằng a_1, sau đó lần lượt so sánh max với các số a_i (i = 2, 3, …, n), nếu max < a_i thì max bằng a_i, nếu max > a_i thì max không đổi.

```
Function  max_day_so;
   Input:    Dãy số a1, a2, … ,an;
   Output:  Giá trị lớn nhất (max) của dãy số đã nhập;
   begin
      max := a1 ;
      for  i:= 2  to  n  do
           if ai > max  then  max := ai ;
        max_day_so := max;
   end;
```

Chú thích: Vòng lặp for còn cách viết lùi biến điều khiển như sau:

for biến điều khiển := giá trị cuối **downto** giá trị đầu **do** câu lệnh hoặc khối câu lệnh;

Việc thực hiện câu lệnh này tương tự như khi viết biến điều khiển tăng dần.

b. Loại có bước lặp không xác định: Có hai cách viết

Cách thứ nhất: ***while*** điều kiện ***do*** câu lệnh hoặc khối câu lệnh;

Khi thực hiện, điều kiện được kiểm tra, nếu điều kiện được thoả mãn thì câu lệnh (khối câu lệnh) được thực hiện. Nếu điều kiện không thoả mãn thì vòng lặp dừng và cho kết quả.

Thí dụ: Kiểm tra xem số nguyên dương m đã cho có phải là số nguyên tố không?

Thuật toán như sau: Số m là số nguyên tố nếu nó không chia hết cho bất cứ số nguyên dương khác 1 nào nhỏ hơn hoặc bằng $\sqrt{m}$.

Thật vậy, nếu m là một hợp số (không phải là số nguyên tố), nghĩa là tồn tại các số nguyên dương a, b sao cho:

$$m = a.b \quad \Rightarrow \quad a \le \sqrt{m} \text{ hoặc } b \le \sqrt{m}$$

Vậy, nếu m là số nguyên tố thì m không chia hết cho mọi số nguyên dương i, $2 \le i \le \sqrt{m}$

```
Procedure nguyento(m);
   Input:   Số nguyên dương m;
   Output:  True, nếu m là số nguyên tố;   False, nếu m không phải là số nguyên tố;
      begin
            i := 2;
            while  i ≤ √m   do
             begin
                if  m mod i  = 0  then  nguyento := false
                                  else  nguyento := true;
              i := i+1;
             end;
       end;
```

Cách thứ hai: ***repeat*** câu lệnh hoặc khối câu lệnh ***until*** điều kiện;

Khi thực hiện, câu lệnh (khối câu lệnh) được thực hiện, sau đó điều kiện được kiểm tra, nếu điều kiện sai thì vòng lặp được thực hiện, nếu điều kiện đúng thì vòng lặp dừng và cho kết quả.

Thí dụ: Thuật toán Ơ-clit tìm ước số chung lớn nhất của hai số nguyên dương a, b như sau: Giả sử a > b và a chia cho b được thương là q và số dư là r, trong đó a, b, q, r là các số nguyên dương:

$$a = bq + r$$

suy ra: ƯCLN(a, b) = ƯCLN(b, r)

và số dư cuối cùng khác không là ước số chung lớn nhất của a và b.

Thật vậy: Giả sử d là ước số chung của hai số nguyên dương a và b, khi đó: r = a – bq chia hết cho d. Vậy d là ước chung của b và r.

Ngược lại, nếu d là ước số chung của b và r, khi đó do bq + r = a cũng chia hết cho d. Vậy d là ước số chung của a và b.

Chẳng hạn, muốn tìm ước số chung lớn nhất của 111 và 201 ta làm như sau:

$$201 = 1.\ 111 + 90$$
$$111 = 1.\ 90 + 21$$
$$90 = 4.\ 21 + 6$$
$$21 = 3.\ 6 + 3$$
$$6 = 2.\ 3$$

Vậy ƯCLN(111, 201) = 3 (3 là số dư cuối cùng khác 0).

```
Function   UCLN(a, b)
     Input:   a, b là 2 số nguyên dương;
     Output:  UCLN(a, b);
     begin
```

```
x := a;
  y : = b;
 repeat
  begin
    r := x mod y;  (* r là phần dư khi chia x cho y *)
    x : = y;    y := r ;
        if  y ≠ 0 then uc := y;
  end;
 until  y =  0;
UCLN := uc;
end  ;
```

Chú ý: Khi giải các bài toán phức tạp thì thường phải phân chia bài toán đó thành các bài toán nhỏ hơn gọi là các bài toán con. Khi đó phải xây dựng các thủ tục hoặc hàm để giải các bài toán con đó, sau đó tập hợp các bài toán con này để giải bài toán ban đầu đã đặt ra. Thuật ngữ tin học gọi các chương trình giải bài toán con đó là các chương trình con.

Thí dụ: Tìm số nguyên tố nhỏ nhất lớn hơn số nguyên dương m đã cho.

```
Procedure  So_nguyen_to_lon_hon(m);
Input:     Số nguyên dương m;
Output:  n là số nguyên tố nhỏ nhất lớn hơn m;
begin
    n := m + 1;
    while  nguyento(n) = false  do  n := n + 1;
end;
```

4. Độ phức tạp của thuật toán

Có hai lý do làm cho một thuật toán đúng có thể không thực hiện được trên máy tính. Đó là do máy tính không đủ bộ nhớ để thực hiện hoặc thời gian tính toán quá dài. Tương ứng với hai lý do trên người ta đưa ra hai khái niệm:

- **Độ phức tạp không gian của thuật toán,** độ phức tạp này gắn liền với các cấu trúc dữ liệu được sử dụng. Vấn đề này không thuộc phạm vi của môn học này.
- **Độ phức tạp thời gian của thuật toán,** độ phức tạp này được thể hiện qua số lượng các câu lệnh về các phép gán, các phép tính số học, phép so sánh, … được sử dụng trong thuật toán khi các giá trị đầu vào có kích thước đã cho.

4.1. Khái niệm độ tăng của hàm

Trước hết xét thí dụ: Giả sử thời gian tính toán của một thuật toán phụ thuộc vào kích thước n của đầu vào theo công thức:

$$t(n) = 60n^2 + 9n + 9$$

Bảng sau cho thấy khi n lớn, t(n) xấp xỉ số hạng $60n^2$:

n	$t(n) = 60n^2 + 9n + 9$	$60n^2$
10	9099	6000
100	600 909	600 000
1 000	60 009 009	60 000 000
10 000	6 000 090 009	6 000 000 000

Định nghĩa: Cho f(x) và g(x) là hai hàm từ tập số nguyên hoặc tập số thực vào tập các số thực. Người ta nói f(x) là O(g(x)) hay f(x) có quan hệ big-O với g(x), ký hiệu f(x) = O(g(x)), nếu tồn tại hai hằng số C và k sao cho:

$$|f(x)| \le C.|g(x)|, \quad \forall x \ge k.$$

Thí dụ 1. $t(n) = 60n^2 + 9n + 9 = O(n^2)$

Thật vậy: $\forall n \ge 1$, ta có: $|60n^2 + 9n + 9| = 60n^2 + 9n + 9$

$$= n^2\left(60 + \frac{9}{n} + \frac{9}{n^2}\right)$$

$$\le n^2(60 + 9 + 9) = 78n^2.$$

Vậy $C = 78$; $k = 1$.

Tương tự ta có thể chứng minh:

$$P_n(x) = a_0x^n + a_1x^{n-1} + ... + a_n = O(x^n), \ x \in R.$$

Thí dụ 2. $f(n) = 1 + 2 + 3 + ... + n < n + n + n + ... + n = n.n = n^2$.

Vậy $f(n) = O(n^2)$.

Thí dụ 3. Ta có: $n! < n^n$, suy ra: $n! = O(n^n)$; $(C = k = 1)$

Thí dụ 4. $\forall n$: $n! < n^n$, $\lg(n!) < n \lg n$,

suy ra $\lg(n!) = O(n\lg n)$; $(C = k = 1)$

Thí dụ 5. Ta có: $\lg n < n$, suy ra $\lg n = O(n)$; $(C = k = 1)$

Có thể hiểu đơn giản quan hệ f(x) = O(g(x)) là f(x) và g(x) là "cùng cấp", tuy nhiên g(x) là hàm đơn giản nhất có thể đại diện cho f(x) về độ lớn cũng như tốc độ biến thiên.

4.2. Độ tăng của tổ hợp các hàm

Định lý: Nếu $f_1(x) = O(g_1(x))$ và $f_2(x) = O(g_2(x))$

Thì: 1) $(f_1 + f_2)(x) = O(\max\{g_1(x), g_2(x)\})$.

2) $(f_1.f_2)(x) = O(g_1(x).g_2(x))$

Chứng minh: Theo giả thiết, ta có: $|f_1(x)| \le C_1|g_1(x)|$, $\forall x > k_1$

$|f_2(x)| \le C_2|g_2(x)|$, $\forall x > k_2$

Chọn $k = \max(k_1; k_2)$ thì cả hai bất đẳng thức đều thoả mãn. Do đó:

1) $|(f_1 + f_2)(x)| = |f_1(x) + f_2(x)| \le |f_1(x)| + |f_2(x)| \le$

$$\le C_1|g_1(x)| + C_2|g_2(x)| \le (C_1 + C_2)g(x)$$

ở đây $g(x) = \max\{|g_1(x)|, |g_2(x)|\}$.

2) $|(f_1.f_2)(x)| = |f_1(x)|.|f_2(x)| \le C_1C_2\,|g_1(x)|.|g_2(x)| = C_1C_2|g_1(x)g_2(x)|$

Hệ quả: Nếu $f_1(x) = O(g(x))$, $f_2(x) = O(g(x))$ thì $(f_1+f_2)(x) = O(g(x))$

Thí dụ. Cho đánh giá O của các hàm:

1/ $f(n) = 2n\lg(n!) + (n^3 + 3)\lg n$, $n \in N$.

2/ $f(x) = (x + 1)\lg(x^2 + 1) + 3x^2$, $x \in R$.

Giải: 1) Ta có: $\lg(n!) = O(n\lg n) \Rightarrow 2n\lg(n!) = O(n^2\lg n)$

$(n^3 + 3)\lg n = O(n^3\lg n)$

Vậy $f(n) = O(n^3\lg n)$

2) Ta có: $\lg(x^2 + 1) \le \lg 2x^2 = \lg 2 + 2\lg x \le 3\lg x$, $\forall x > 1$

$\Rightarrow \lg(x^2 + 1) = O(\lg x) \Rightarrow (x + 1)\lg(x^2 + 1) = O(x\lg x)$

Mặt khác: $3x^2 = O(x^2)$ và $\max\{x\lg x; x^2\} = x^2$.

Vậy $f(x) = O(x^2)$.

4.3. Độ phức tạp của thuật toán

Như đã nói ở phần đầu của mục 4, chúng ta chỉ đề cập đến độ phức tạp về thời gian của thuật toán. Độ phức tạp về thời gian của thuật toán được đánh giá qua số lượng các phép toán mà thuật toán sử dụng. Vì vậy phải đếm các phép toán có trong thuật toán. Các phép toán phải đếm là:

- Các phép so sánh các số nguyên hoặc số thực;
- Các phép tính số học: cộng, trừ, nhân, chia;
- Các phép gán;
- Và bất kỳ một phép tính sơ cấp nào khác xuất hiện trong quá trình tính toán.

Giả sử số các phép toán của thuật toán là f(n), trong đó n là kích thước đầu vào, khi đó người ta thường quy độ phức tạp về thời gian của thuật toán về các mức:

- Độ phức tạp O(1), gọi là độ phức tạp hằng số, nếu f(n) = O(1).
- Độ phức tạp $O(\log_a n)$, gọi là độ phức tạp logarit, nếu $f(n) = O(\log_a n)$. (Điều kiện $a > 1$)
- Độ phức tạp O(n), gọi là độ phức tạp tuyến tính, nếu f(n) = O(n).
- Độ phức tạp $O(n\log_a n)$, gọi là độ phức tạp nlogarit nếu $f(n) = O(\log_a n)$. (Điều kiện $a > 1$)
- Độ phức tạp $O(n^k)$, gọi là độ phức tạp đa thức, nếu $f(n) = O(n^k)$.
- Độ phức tạp $O(a^n)$, gọi là độ phức tạp mũ, nếu $f(n) = O(a^n)$. (Điều kiện $a > 1$)
- Độ phức tạp O(n!), gọi là độ phức tạp giai thừa, nếu f(n) = O(n!).

Thí dụ 1. Tìm độ phức tạp của thuật toán để giải bài toán: Tìm số lớn nhất trong dãy n số nguyên $a_1, a_2, \ldots, a_n$ đã cho:

Procedure max($a_1, a_2, \ldots, a_n$);
Input: Dãy số $a_1, a_2, \ldots, a_n$;
Output: Số lớn nhất (max) của dãy số đã cho;
begin
max := a_1;
for i := 2 to n do

```
        if  a_i > max   then  max := a_i;
    end;
```

Mỗi bước của vòng lặp for phải thực hiện nhiều nhất 3 phép toán: phép gán biến điều khiển i, phép so sánh a_i với max và có thể là phép gán a_i vào max; vòng lặp có (n – 1) bước (i = 2, 3, …, n) do đó nhiều nhất có cả thảy 3(n - 1) phép toán phải thực hiện. Ngoài ra thuật toán còn phải thực hiện phép gán đầu tiên max := a_1.

Vậy số phép toán nhiều nhất mà thuật toán phải thực hiện là:

$$3(n – 1) + 1 = 3n – 2 = O(n)$$

Độ phức tạp về thời gian của thuật toán là độ phức tạp tuyến tính.

Thí dụ 2. Độ phức tạp của thuật toán nhân ma trận.

```
Procedure   nhân_matran;
   Input:  2 ma trận  A = (a_ij)_m x p  và  B = (b_ij)_p x n;
   Output:   ma trận tích  AB = (c_ij)_m x n ;
   Begin
   for  i:=1 to  m  do
     for  j:=1 to  n  do
        begin
        c_ij := 0;
        for  k:=1  to  p  do
           c_ij := c_ij + a_ik b_kj ;
        end;
   End.
```

Số phép cộng và số phép nhân trong thuật toán trên là: Với mỗi phần tử c_{ij} phải thực hiện p phép nhân và p phép cộng. Có tất cả m.n phần tử c_{ij}, vậy phải thực hiện 2mnp phép cộng và phép nhân.

Để xác định độ phức tạp của thuật toán, ta giả sử A, B là hai ma trận vuông cấp n, nghĩa là m = n = p. như vậy phải cần $2n^3$ phép cộng và phép nhân. Vậy độ phức tạp của thuật toán là $O(n^3)$ – độ phức tạp đa thức.

Một điều thú vị là, khi nhân từ 3 ma trận trở lên thì số phép tính cộng và nhân phụ thuộc vào thứ tự nhân các ma trận ấy. Chẳng hạn A, B, C là các ma trận có kích thước tương ứng là 30×20, 20×40, 40×10. Khi đó:

Nếu thực hiện theo thứ tự ABC =A(BC) thì tích BC là ma trận kích thước 20×10 và cần thực hiện 20.40.10 = 8000 phép tính cộng và nhân. Ma trận A(BC) có kích thước 30×10 và cần thực hiện 30.20.10 = 6000 phép cộng và nhân. Từ đó suy ra cần thực hiện 8000+6000 = 14000 phép tính cộng và nhân để hoàn thành tích ABC.

Tương tự, nếu thực hiện theo thứ tự ABC = (AB)C thì cần thực hiện 30.20.40 phép tính cộng và nhân để thực hiện tích AB và 30.40.10 phép cộng và nhân để thực hiên tích (AB)C. Do đó số các phép tính cộng và nhân phải thực hiện để hoàn thành tích ABC là 24000+12000 = 36000 phép tính.

Rõ ràng hai cách nhân cho kết quả về số lượng các phép tính phải thực hiện là khác nhau.

5. Thuật toán tìm kiếm

Bài toán tìm kiếm được phát biểu như sau: Tìm trong dãy số $a_1, a_2, \ldots, a_n$ một phần tử có giá trị bằng số a cho trước và ghi lại vị trí của phần tử tìm được.

Bài toán này có nhiều ứng dụng trong thực tế. Chẳng hạn việc tìm kiếm từ trong từ điển, việc kiểm tra lỗi chính tả của một đoạn văn bản, …

Có hai thuật toán cơ bản để giải bài toán này: Thuật toán tìm kiếm tuyến tính và thuật toán tìm kiếm nhị phân. Chúng ta lần lượt xét các thuật toán này.

5.1. Thuật toán tìm kiếm tuyến tính (còn gọi là thuật toán tìm kiếm tuần tự).

Đem so sánh a lần lượt với a_i (i = 1, 2, …, n) nếu gặp một giá trị $a_i = a$ thì ghi lại vị trí của a_i, nếu không gặp giá trị $a_i = a$ nào ($a_i \neq a. \forall i$) thì trong dãy không có số nào bằng a.

```
Procedure  Tim_tuyen_tinh_phan_tu_bang_a;
Input:     a và dãy số a1, a2, ... , an;
Output:  Vị trí phần tử của dãy có giá trị bằng a, hoặc là số 0 nếu không tìm
                thấy a trong dãy;
begin
   i := 1;
   while  (i ≤ n  and ai ≠ a)  do    i := i + 1;
      if  i ≤ n  then   vitri := i    else   vitri := 0;
end;
```

Như vậy nếu a được tìm thấy ở vị trí thứ i của dãy ($a_i = a$) thì câu lệnh i := i + 1 trong vòng lặp while được thực hiện i lần (i = 1, 2, …, n). Nếu a không được tìm thấy, câu lệnh phải thực hiện n lần. Vậy số phép toán trung bình mà thuật toán phải thực hiện là:

$$\frac{1+2+\ldots+n}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2} = O(n)$$

Vậy, độ phức tạp của thuật toán tìm kiếm tuyến tính là độ phức tạp tuyến tính.

5.2. Thuật toán tìm kiếm nhị phân.

Giả thiết rằng các phần tử của dãy được xếp theo thứ tự tăng dần. Khi đó so sánh a với số ở giữa dãy, nếu $a < a_m$ với $m = \left[\frac{1+n}{2}\right]$ (cần nhắc lại rằng phần nguyên của x: [x] là số nguyên nhỏ nhất có trong x) thì tìm a trong dãy $a_1, \ldots, a_m$, nếu $a > a_m$ thì tìm a trong dãy $a_{m+1}, \ldots, a_n$. Đối với mỗi dãy con (một nửa của dãy đã cho) được làm tương tự để chỉ phải tìm phần tử có giá trị bằng a ở một nửa dãy con đó. Quá trình tìm kiếm kết thúc khi tìm thấy vị trí của phần tử có giá trị bằng a hoặc khi dãy con chỉ còn 1 phần tử.

Chẳng hạn việc tìm số 8 trong dãy số 5, 6, 8, 9, 11, 12, 13, 15, 16, 17, 18, 19, 20, 22 được tiến hành như sau:

Dãy đã cho gồm 14 số hạng, chia dãy thành 2 dãy con:

5, 6, 8, 9, 11, 12, 13 và 15, 16, 17, 18, 19, 20, 22

Vì 8 < 13 nên tiếp theo chỉ cần tìm ở dãy đầu tiên. Tiếp tục chia đôi thành 2 dãy: 5, 6, 8, 9 và 11, 12, 13; vì 8 < 9 nên lại chỉ phải tìm ở dãy 5, 6, 8, 9. Lại chia đôi dãy này thành 2 dãy con 5, 6 và 8, 9 thấy ngay 8 thuộc về dãy con 8, 9 và quá trình tìm kiếm kết thúc, vị trí của số 8 trong dãy đã cho là thứ ba

```
Procedure  Tim_nhi_phan_phan_tu_bang_a;
Input:     a và dãy số a1, a2, ... , an đã xếp theo thứ tự tăng;
Output:  Vị trí phần tử của dãy có giá trị bằng a, hoặc là số 0 nếu không tìm thấy
                trong dãy;
Begin
 i := 1;  (* i là điểm mút trái của khoảng tìm kiếm*)
 j := n;  (* j là điểm mút phải của khoảng tìm kiếm*)
 while i < j do
   begin
      m := ⌈(1 + j)/2⌉;
     if a > am then i := m+1
          else j := m;
   end;
         if  a = ai  then   vitri := i    else   vitri := 0;
end;
```

Độ phức tạp của thuật toán tìm kiếm nhị phân được đánh giá như sau: Không giảm tổng quát có thể giả sử độ dài của dãy $a_1, a_2, \ldots, a_n$ là $n = 2^k$ với k là số nguyên dương. (Nếu n không phải là lũy thừa của 2, luôn tìm được số k sao cho $2^{k-1} < n < 2^k$ do đó có thể xem dãy đã cho là một phần của dãy có 2^k phần tử). Như vậy phải thực hiện nhiều nhất k lần chia đôi các dãy số (mỗi nửa dãy của lần chia đôi thứ nhất có 2^{k-1} phần tử, của lần chia đôi thứ hai có 2^{k-2} phần tử, …, và của lần chia đôi thứ k là $2^{k-k} = 2^0 = 1$ phần tử). Nói cách khác là nhiều nhất có k vòng lặp while được thực hiện trong thuật toán tìm kiếm nhị phân. Trong mỗi vòng lặp while phải thực hiện hai phép so sánh, và vòng lặp cuối cùng khi chỉ còn 1 phần tử phải thực hiện 1 phép so sánh để biết không còn 1 phần tử nào thêm nữa và 1 phép so sánh để biết a có phải là phần tử đó hay không. Từ đó thấy rằng thuật toán phải thực hiện nhiều nhất $2k + 2 = 2[\log_2 n] + 2 = O(\log n)$ phép so sánh.

Vậy, độ phức tạp của thuật toán tìm kiếm nhị phân là độ phức tạp logarit.

6. Thuật toán đệ quy

6.1. Công thức truy hồi

Đôi khi rất khó định nghĩa một đối tượng nào đó một cách tường minh, nhưng có thể định nghĩa đối tượng đó qua chính nó với đầu vào nhỏ hơn. Cách định nghĩa như vậy gọi là cách định nghĩa bằng truy hồi hoặc định nghĩa bằng đệ quy và nó cho một công thức gọi là công thức truy hồi.

Định nghĩa: Định nghĩa bằng truy hồi bao gồm các quy tắc để xác định các đối tượng, trong đó có một số quy tắc dùng để xác định các đối tượng ban đầu gọi là các điều kiện ban đầu; còn các quy tắc khác dùng để xác định các đối tượng tiếp theo gọi là công thức truy hồi.

Thí dụ 1. Dãy số a_n được định nghĩa bằng đệ quy như sau:

$$a_0 = 3;\ a_n = a_{n-1} + 3.$$

Trong đó $a_0 = 3$ là điều kiện ban đầu, còn $a_n = a_{n-1} + 3$ là công thức truy hồi.

Thí dụ 2. Định nghĩa bằng đệ quy giai thừa của số tự nhiên n là:

$$GT(0) = 1; \quad GT(n) = n.GT(n-1).$$

Vì $GT(n) = n! = n(n-1)(n-2)\ldots1 = n.GT(n-1)$. Trong đó $GT(0) = 1$ là điều kiện ban đầu, còn $GT(n) = n.GT(n-1)$ là công thức truy hồi.

Thí dụ 3. Dãy số $F_0, F_1, F_2, \ldots, F_n$ được định nghĩa:

$$F_0 = 0;\ F_1 = 1;\ F_n = F_{n-1} + F_{n-2}.$$

Đó chính là định nghĩa bằng đệ quy của dãy số có tên là dãy Fibonacci. Trong đó $F_0 = 0$, $F_1 = 1$ là các điều kiện ban đầu, còn $F_n = F_{n-1} + F_{n-2}$ là công thức đệ quy.

Dễ thấy một số số hạng đầu tiên của dãy là: 0; 1; 1; 2; 3; 5; 8; 13; 21; …

6.2. Thuật toán đệ quy.

Nhiều khi việc giải bài toán với đầu vào xác định có thể đưa về việc giải bài toán đó với giá trị đầu vào nhỏ hơn. Chẳng hạn:

$$n! = n\ .\ (n-1)! \quad \text{hay} \quad UCLN(a, b) = UCLN(a \bmod b, b)\ ,\ a > b$$

Định nghĩa: Một thuật toán gọi là đệ quy nếu thuật toán đó giải bài toán bằng cách rút gọn liên tiếp bài toán ban đầu tới bài toán cũng như vậy nhưng với dữ liệu đầu vào nhỏ hơn.

Dễ thấy cơ sở của thuật toán là công thức truy hồi.

Thí dụ 1. Tính giai thừa của số tự nhiên n bằng đệ quy.

```
Function   GT(n);
   Input:    Số tự nhiên n;
Output:   Giá trị của n!;
Begin
 if  n = 0  then  GT(0) := 1
          else   GT(n) := n*GT(n – 1);
End;
```

Thí dụ 2. Tính số hạng của dãy Fibonacci bằng đệ quy.

```
Function   Fibonacci(n);
Input:    Vị trí thứ n của dãy Fibonacci;
Output:  Giá trị  Fn của dãy Fibonaci;
Begin
```

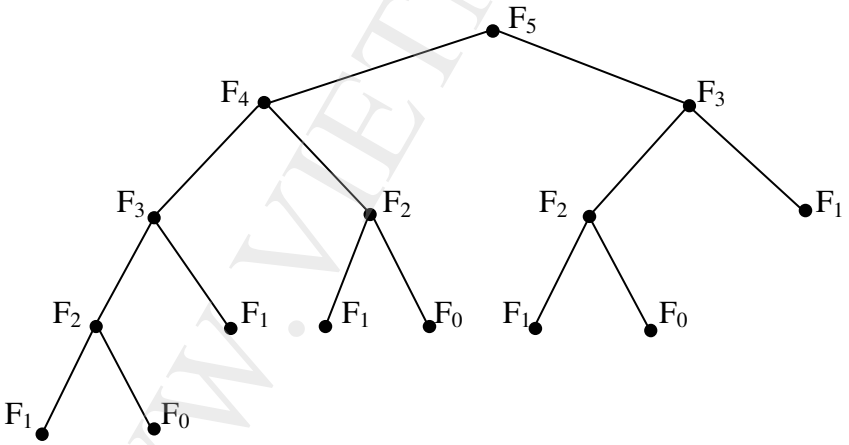
```
if  n = 0  then  Fibonacci(0) := 0
        else  if  n = 1  then  Fibonacci(1) := 1
            else   Fibonacci(n) := Fibonacci(n-1) + Fibonacci(n-2);
End;
```

Thí dụ 3. Thuật toán đệ quy tìm UCLN(a, b).

```
Function  UCLN(a, b);
Input:    Hai số nguyên dương a và b;
Output: Ước số chung lớn nhất của a và b;
Begin
  if  b = 0  then  UCLN(a, b) := a
        else  if a > b  then  UCLN(a, b) := UCLN(a mod b, b)
                              else  UCLN(a, b) := UCLN(b mod a, a);
End;
```

Bây giờ chúng ta thử tìm độ phức tạp về thời gian của một vài thuật toán viết bằng đệ quy. Chẳng hạn xét thuật toán đệ quy tính số hạng của dãy Fibonacci, để tính F_n ta biểu diễn $F_n = F_{n-1} + F_{n-2}$, sau đó thay thế cả hai số này bằng tổng của hai số Fibonacci bậc thấp hơn. Quá trình tiếp tục như vậy cho đến khi F_0 và F_1 xuất hiện thì được thay bằng các giá trị của chúng trong định nghĩa.

Mỗi bước đệ quy cho tới khi F_0 và F_1 xuất hiện, các số Fibonacci được tính hai lần. Chẳng hạn giản đồ cây ở hình 2 cho ta hình dung cách tính F_5 theo thuật toán đệ quy. Từ đó có thể thấy rằng để tính F_n cần thực hiện $F_{n+1} - 1$ phép cộng.



Hình 2. Lược đồ tính F_5 bằng đệ quy

Độ phức tạp của thuật toán đệ quy tìm ước số chung lớn nhất của hai số nguyên dương a, b (thí dụ 3): UCLN(a,b) = UCLN(a mod b,b), nếu $a \geq b$ (a mod b là phần dư khi chia a cho b) được đánh giá bằng cách ứng dụng dãy Fibonacci.

Trước hết bằng quy nạp toán học chúng ta chứng minh số hạng tổng quát của dãy Fibonacci thỏa mãn:

$F_n > \alpha^{n-2}$, $\forall n \geq 3$, trong đó $\alpha = \dfrac{1+\sqrt{5}}{2}$. (1)

Thật vậy:

Ta có: $\alpha < 2 = F_3$, nghĩa là (1) đúng với n = 3.

Giả sử $F_n > \alpha^{n-2}$ đúng với n, xét với n+1. Dễ thấy α là một nghiệm của phương trình $x^2 - x - 1 = 0$ nên suy ra $\alpha^2 = \alpha + 1$. Từ đó:

$$\alpha^{n-1} = \alpha^2\alpha^{n-3} = (\alpha + 1)\alpha^{n-3} = \alpha^{n-2} + \alpha^{n-3}.$$

Theo giả thiết quy nạp, nếu $n \geq 4$, ta có $F_{n-1} > \alpha^{n-3}$ và $F_n > \alpha^{n-2}$. Thay vào định nghĩa của dãy Fibonacci:

$$F_{n+1} = F_n + F_{n-1} > \alpha^{n-2} + \alpha^{n-3} = \alpha^{n-1}$$

Vậy $F_n > \alpha^{n-2}$, $\forall n \geq 3$.

Công thức (1) được chứng minh.

Trở lại thuật toán đệ quy tìm ước số chung lớn nhất của hai số nguyên dương a, b ($a \geq b$). Độ phức tạp của thuật toán được đánh giá qua số lượng các phép chia dùng trong thuật toán này.

Đặt $r_0 = a$, $r_1 = b$, ta có:

$r_0 = r_1q_1 + r_2$; $0 \leq r_2 < r_1$,

$r_1 = r_2q_2 + r_3$; $0 \leq r_3 < r_2$,

.....

$r_{n-2} = r_{n-1}q_{n-1} + r_n$; $0 \leq r_n < r_{n-1}$,

$r_{n-1} = r_nq_n$;

Như vậy phải dùng n phép chia để tìm r_n = UCLN(a,b). Các thương $q_1, q_2, \ldots, q_{n-1}$ luôn lớn hơn hoặc bằng 1, còn $q_n \geq 2$. Từ đó suy ra:

$r_n \geq 1 = F_2$,

$r_{n-1} \geq 2r_n = 2F_2 = F_3$,

$r_{n-2} \geq r_{n-1} + r_n \geq F_3 + F_2 = F_4$,

...

$r_2 \geq r_3 + r_4 \geq F_{n-1} + F_{n-2} = F_n$,

$b = r_1 \geq r_2 + r_3 \geq F_n + F_{n-1} = F_{n+1}$.

trong đó F_n là số hạng thứ n trong dãy Fibonacci.

Vậy nếu n là số các phép chia trong thuật toán Ơ-clit tìm ước số chung lớn nhất của hai số nguyên dương a, b thì $b \geq F_{n+1}$, trong đó F_n là số Fibonacci thứ n.

Do $F_{n+1} > \alpha^{n-1}$ với $n > 2$ và $\alpha = \dfrac{1+\sqrt{5}}{2}$ nên $b > \alpha^{n-1}$.

Từ đó: $\lg b > (n-1)\lg\alpha > \dfrac{n-1}{5}$ (vì $\lg\alpha \approx 0{,}208 > \dfrac{1}{5}$).

Vậy: $n - 1 < 5\lg b \Rightarrow n < 5\lg b + 1 = O(\lg b)$.