LỜI NÓI ĐẦU

Được sự động viên mạnh mẽ của các đồng nghiệp trong các Khoa Toán-Cơ-Tin học, Công nghệ Thông tin và Vật lý (Trường Đại học Khoa học-Đại học Huế), các Khoa Toán và Tin học (Trường Đại học Sư phạm-Đại học Huế) và đặc biệt do nhu cầu học tập của các sinh viên trong Đại học Huế ở các Khoa nói trên và các học viên cao học ngành Phương pháp giảng dạy Toán, chúng tôi mạnh dạn viết giáo trình *Toán rời rạc* trong khi trên thị trường sách có khá nhiều tài liệu liên quan đến Toán rời rạc. Điều mà chúng tôi mong muốn là các kiến thức của học phần này phải được đưa vào đầy đủ, cô đọng, chính xác, cập nhật, bám sát theo yêu cầu đào tạo sinh viên các ngành Công nghệ Thông tin, Toán-Tin, Vật lý-Tin và một số ngành kỹ thuật khác của các trường đại học và cao đẳng.

Với sự nổ lực hết mình của bản thân, chúng tôi thiết nghĩ đây sẽ là tài liệu tham khảo tốt cho các giáo viên giảng dạy học phần toán rời rạc, các học viên cao học ngành Phương pháp giảng dạy Toán, các thí sinh thi vào cao học ngành công nghệ thông tin, các sinh viên thuộc các ngành được đề cập ở trên và các học sinh thuộc khối chuyên Toán, chuyên Tin.

Nội dung của tài liệu này được bố trí trong 4 phần, không kể lời nói đầu, mục lục, tài liệu tham khảo và phần phụ lục:

-- Phần 1 được dành cho Chương I đề cập đến *Thuật toán*;

-- Phần 2 được dành cho Chương II nói đến *bài toán đếm*;

-- Phần 3, đây là phần chiếm nhiều trang nhất trong giáo trình, bàn về *Lý thuyết đồ thị và các ứng dụng* gồm 5 chương: *Đồ thị, Đồ thị Euler và đồ thị Hamilton, Một số bài toán tối ưu trên đồ thị, Cây, Đồ thị phẳng và tô màu đồ thị*;

-- Phần 4 được dành cho Chương 8, chương cuối cùng, đề cập đến *Đại số Boole*.

Trong mỗi chương, các chứng minh của các định lý, mệnh đề được trình bày chi tiết, ngoại trừ một số định lý có phần chứng minh quá phức tạp thì được chúng tôi bỏ qua. Trong các phần của mỗi chương có nhiều ví dụ cụ thể minh hoạ cho những khái niệm cũng như những kết quả của chúng. Cuối của mỗi chương là những bài tập được chọn lọc từ dễ đến khó, bám theo nội dung của chương đó.

Chúng tôi xin chân thành cám ơn các đồng nghiệp đã động viên và góp ý cho công việc viết giáo trình *Toán rời rạc* này và lời cám ơn đặc biệt xin dành cho Khoa Công nghệ Thông tin về sự giúp đỡ quý báu và tạo điều kiện thuận lợi cho việc xuất bản giáo trình này.

Tác giả mong nhận được sự chỉ giáo của các đồng nghiệp và độc giả về những thiếu sót khó tránh khỏi của cuốn sách.

Mùa Thu năm 2003

MỤC LỤC

CHƯƠNG I:
THUẬT TOÁN

1.1. KHÁI NIỆM THUẬT TOÁN.

1.1.1. Mở đầu:

Có nhiều lớp bài toán tổng quát xuất hiện trong toán học rời rạc. Chẳng hạn, cho một dãy các số nguyên, tìm số lớn nhất; cho một tập hợp, liệt kê các tập con của nó; cho tập hợp các số nguyên, xếp chúng theo thứ tự tăng dần; cho một mạng, tìm đường đi ngắn nhất giữa hai đỉnh của nó. Khi được giao cho một bài toán như vậy thì việc đầu tiên phải làm là xây dựng một mô hình dịch bài toán đó thành ngữ cảnh toán học. Các cấu trúc rời rạc được dùng trong các mô hình này là tập hợp, dãy, hàm, hoán vị, quan hệ, cùng với các cấu trúc khác như đồ thị, cây, mạng - những khái niệm sẽ được nghiên cứu ở các chương sau.

Lập được một mô hình toán học thích hợp chỉ là một phần của quá trình giải. Để hoàn tất quá trình giải, còn cần phải có một phương pháp dùng mô hình để giải bài toán tổng quát. Nói một cách lý tưởng, cái được đòi hỏi là một thủ tục, đó là dãy các bước dẫn tới đáp số mong muốn. Một dãy các bước như vậy, được gọi là một thuật toán.

Khi thiết kế và cài đặt một phần mềm tin học cho một vấn đề nào đó, ta cần phải đưa ra phương pháp giải quyết mà thực chất đó là thuật toán giải quyết vấn đề này. Rõ ràng rằng, nếu không tìm được một phương pháp giải quyết thì không thể lập trình được. Chính vì thế, thuật toán là khái niệm nền tảng của hầu hết các lĩnh vực của tin học.

1.1.2. Định nghĩa: Thuật toán là một bảng liệt kê các chỉ dẫn (hay quy tắc) cần thực hiện theo từng bước xác định nhằm giải một bài toán đã cho.

Thuật ngữ “Algorithm” (thuật toán) là xuất phát từ tên nhà toán học Ả Rập Al-Khowarizmi. Ban đầu, từ algorism được dùng để chỉ các quy tắc thực hiện các phép tính số học trên các số thập phân. Sau đó, algorism chuyển thành algorithm vào thế kỷ 19. Với sự quan tâm ngày càng tăng đối với các máy tính, khái niệm thuật toán đã được cho một ý nghĩa chung hơn, bao hàm cả các thủ tục xác định để giải các bài toán, chứ không phải chỉ là thủ tục để thực hiện các phép tính số học.

Có nhiều cách trình bày thuật toán: dùng ngôn ngữ tự nhiên, ngôn ngữ lưu đồ (sơ đồ khối), ngôn ngữ lập trình. Tuy nhiên, một khi dùng ngôn ngữ lập trình thì chỉ những lệnh được phép trong ngôn ngữ đó mới có thể dùng được và điều này thường làm cho sự mô tả các thuật toán trở nên rối rắm và khó hiểu. Hơn nữa, vì nhiều ngôn ngữ lập trình đều được dùng rộng rãi, nên chọn một ngôn ngữ đặc biệt nào đó là điều người ta không muốn. Vì vậy ở đây các thuật toán ngoài việc được trình bày bằng ngôn ngữ tự nhiên cùng với những ký hiệu toán học quen thuộc còn dùng một dạng giả mã để mô tả thuật

toán. Giả mã tạo ra bước trung gian giữa sự mô tả một thuật toán bằng ngôn ngữ thông thường và sự thực hiện thuật toán đó trong ngôn ngữ lập trình. Các bước của thuật toán được chỉ rõ bằng cách dùng các lệnh giống như trong các ngôn ngữ lập trình.

Thí dụ 1: Mô tả thuật toán tìm phần tử lớn nhất trong một dãy hữu hạn các số nguyên.

a) Dùng ngôn ngữ tự nhiên để mô tả các bước cần phải thực hiện:

1. Đặt giá trị cực đại tạm thời bằng số nguyên đầu tiên trong dãy. (Cực đại tạm thời sẽ là số nguyên lớn nhất đã được kiểm tra ở một giai đoạn nào đó của thủ tục.)

2. So sánh số nguyên tiếp sau với giá trị cực đại tạm thời, nếu nó lớn hơn giá trị cực đại tạm thời thì đặt cực đại tạm thời bằng số nguyên đó.

3. Lặp lại bước trước nếu còn các số nguyên trong dãy.

4. Dừng khi không còn số nguyên nào nữa trong dãy. Cực đại tạm thời ở điểm này chính là số nguyên lớn nhất của dãy.

b) Dùng đoạn giả mã:

---

procedure max (a_1, a_2, ..., a_n: integers)

    max:= a_1

    for i:= 2 **to** n

        if max <a_i **then** max:= a_i

{max là phần tử lớn nhất}

Thuật toán này trước hết gán số hạng đầu tiên a_1 của dãy cho biến max. Vòng lặp "for" được dùng để kiểm tra lần lượt các số hạng của dãy. Nếu một số hạng lớn hơn giá trị hiện thời của max thì nó được gán làm giá trị mới của max.

1.1.3. Các đặc trưng của thuật toán:

-- **Đầu vào** (Input)**:** Một thuật toán có các giá trị đầu vào từ một tập đã được chỉ rõ.

-- **Đầu ra** (Output)**:** Từ mỗi tập các giá trị đầu vào, thuật toán sẽ tạo ra các giá trị đầu ra. Các giá trị đầu ra chính là nghiệm của bài toán.

-- **Tính dừng:** Sau một số hữu hạn bước thuật toán phải dừng.

-- **Tính xác định:** Ở mỗi bước, các bước thao tác phải hết sức rõ ràng, không gây nên sự nhập nhằng. Nói rõ hơn, trong cùng một điều kiện hai bộ xử lý cùng thực hiện một bước của thuật toán phải cho những kết quả như nhau.

-- **Tính hiệu quả:** Trước hết thuật toán cần đúng đắn, nghĩa là sau khi đưa dữ liệu vào thuật toán hoạt động và đưa ra kết quả như ý muốn.

-- **Tính phổ dụng:** Thuật toán có thể giải bất kỳ một bài toán nào trong lớp các bài toán. Cụ thể là thuật toán có thể có các đầu vào là các bộ dữ liệu khác nhau trong một miền xác định.

1.2. THUẬT TOÁN TÌM KIẾM.

1.2.1. Bài toán tìm kiếm: Bài toán xác định vị trí của một phần tử trong một bảng liệt kê sắp thứ tự thường gặp trong nhiều trường hợp khác nhau. Chẳng hạn chương trình

kiểm tra chính tả của các từ, tìm kiếm các từ này trong một cuốn từ điển, mà từ điển chẳng qua cũng là một bảng liệt kê sắp thứ tự của các từ. Các bài toán thuộc loại này được gọi là các bài toán tìm kiếm.

Bài toán tìm kiếm tổng quát được mô tả như sau: xác định vị trí của phần tử x trong một bảng liệt kê các phần tử phân biệt $a_1, a_2, ..., a_n$ hoặc xác định rằng nó không có mặt trong bảng liệt kê đó. Lời giải của bài toán trên là vị trí của số hạng của bảng liệt kê có giá trị bằng x (tức là i sẽ là nghiệm nếu $x=a_i$ và là 0 nếu x không có mặt trong bảng liệt kê).

1.2.2. Thuật toán tìm kiếm tuyến tính: Tìm kiếm tuyến tính hay tìm kiếm tuần tự là bắt đầu bằng việc so sánh x với a_1; khi $x=a_1$, nghiệm là vị trí a_1, tức là 1; khi $x\neq a_1$, so sánh x với a_2. Nếu $x=a_2$, nghiệm là vị trí của a_2, tức là 2. Khi $x\neq a_2$, so sánh x với a_3. Tiếp tục quá trình này bằng cách tuần tự so sánh x với mỗi số hạng của bảng liệt kê cho tới khi tìm được số hạng bằng x, khi đó nghiệm là vị trí của số hạng đó. Nếu toàn bảng liệt kê đã được kiểm tra mà không xác định được vị trí của x, thì nghiệm là 0. Giả mã đối với thuật toán tìm kiếm tuyến tính được cho dưới đây:

---

procedure tìm kiếm tuyến tính (x: integer, $a_1,a_2,...,a_n$: integers phân biệt)

```
i := 1
while (i ≤ n and x ≠ a_i)
        i := i + 1
if  i ≤ n then location := i
else location := 0
```

{location là chỉ số dưới của số hạng bằng x hoặc là 0 nếu không tìm được x}

1.2.3. Thuật toán tìm kiếm nhị phân: Thuật toán này có thể được dùng khi bảng liệt kê có các số hạng được sắp theo thứ tự tăng dần. Chẳng hạn, nếu các số hạng là các số thì chúng được sắp từ số nhỏ nhất đến số lớn nhất hoặc nếu chúng là các từ hay xâu ký tự thì chúng được sắp theo thứ tự từ điển. Thuật toán thứ hai này gọi là thuật toán tìm kiếm nhị phân. Nó được tiến hành bằng cách so sánh phần tử cần xác định vị trí với số hạng ở giữa bảng liệt kê. Sau đó bảng này được tách làm hai bảng kê con nhỏ hơn có kích thước như nhau, hoặc một trong hai bảng con ít hơn bảng con kia một số hạng. Sự tìm kiếm tiếp tục bằng cách hạn chế tìm kiếm ở một bảng kê con thích hợp dựa trên việc so sánh phần tử cần xác định vị trí với số hạng giữa bảng kê. Ta sẽ thấy rằng thuật toán tìm kiếm nhị phân hiệu quả hơn nhiều so với thuật toán tìm kiếm tuyến tính.

Thí dụ 2. Để tìm số 19 trong bảng liệt kê 1,2,3,5,6,7,8,10,12,13,15,16,18,19,20,22 ta tách bảng liệt kê gồm 16 số hạng này thành hai bảng liệt kê nhỏ hơn, mỗi bảng có 8 số hạng, cụ thể là: 1,2,3,5,6,7,8,10 và 12,13,15,16,18,19,20,22. Sau đó ta so sánh 19 với số hạng cuối cùng của bảng con thứ nhất. Vì 10<19, việc tìm kiếm 19 chỉ giới hạn trong bảng liệt kê con thứ 2 từ số hạng thứ 9 đến 16 trong bảng liệt kê ban đầu. Tiếp theo, ta

lại tách bảng liệt kê con gồm 8 số hạng này làm hai bảng con, mỗi bảng có 4 số hạng, cụ thể là 12,13,15,16 và 18,19,20,22. Vì 16<19, việc tìm kiếm lại được giới hạn chỉ trong bảng con thứ 2, từ số hạng thứ 13 đến 16 của bảng liệt kê ban đầu. Bảng liệt kê thứ 2 này lại được tách làm hai, cụ thể là: 18,19 và 20,22. Vì 19 không lớn hơn số hạng lớn nhất của bảng con thứ nhất nên việc tìm kiếm giới hạn chỉ ở bảng con thứ nhất gồm các số 18,19, là số hạng thứ 13 và 14 của bảng ban đầu. Tiếp theo bảng con chứa hai số hạng này lại được tách làm hai, mỗi bảng có một số hạng 18 và 19. Vì 18<19, sự tìm kiếm giới hạn chỉ trong bảng con thứ 2, bảng liệt kê chỉ chứa số hạng thứ 14 của bảng liệt kê ban đầu, số hạng đó là số 19. Bây giờ sự tìm kiếm đã thu hẹp về chỉ còn một số hạng, so sánh tiếp cho thấy19 là số hạng thứ 14 của bảng liệt kê ban đầu.

Bây giờ ta có thể chỉ rõ các bước trong thuật toán tìm kiếm nhị phân.

Để tìm số nguyên x trong bảng liệt kê $a_1,a_2,...,a_n$ với $a_1 < a_2 < ... < a_n$, ta bắt đầu bằng việc so sánh x với số hạng a_m ở giữa của dãy, với m=[(n+1)/2]. Nếu $x > a_m$, việc tìm kiếm x giới hạn ở nửa thứ hai của dãy, gồm $a_{m+1},a_{m+2},...,a_n$. Nếu x không lớn hơn a_m, thì sự tìm kiếm giới hạn trong nửa đầu của dãy gồm $a_1,a_2,...,a_m$.

Bây giờ sự tìm kiếm chỉ giới hạn trong bảng liệt kê có không hơn [n/2] phần tử. Dùng chính thủ tục này, so sánh x với số hạng ở giữa của bảng liệt kê được hạn chế. Sau đó lại hạn chế việc tìm kiếm ở nửa thứ nhất hoặc nửa thứ hai của bảng liệt kê. Lặp lại quá trình này cho tới khi nhận được một bảng liệt kê chỉ có một số hạng. Sau đó, chỉ còn xác định số hạng này có phải là x hay không. Giả mã cho thuật toán tìm kiếm nhị phân được cho dưới đây:

```
procedure tìm kiếm nhị phân (x: integer, a1,a2,...,an: integers tăng dần)
    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
    begin
        m:= [(i+j)/2]
        if x>am then i:=m+1
        else  j := m
    end
        if  x = ai  then  location := i
        else  location := 0
{location là chỉ số dưới của số hạng bằng x hoặc 0 nếu không tìm thấy x}
```

1.3. ĐỘ PHỨC TẠP CỦA THUẬT TOÁN.

1.3.1. Khái niệm về độ phức tạp của một thuật toán:

Thước đo hiệu quả của một thuật toán 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 dung lượng bộ nhớ đòi hỏi để thực hiện thuật toán khi các giá trị đầu vào có kích thước xác định. Các vấn đề như thế liên quan đến độ phức tạp tính toán của một thuật toán. Sự phân tích thời gian cần thiết để giải một bài toán có kích thước đặc biệt nào đó liên quan đến độ phức tạp thời gian của thuật toán. Sự phân tích bộ nhớ cần thiết của máy tính liên quan đến độ phức tạp không gian của thuật toán. Vệc xem xét độ phức tạp thời gian và không gian của một thuật toán là một vấn đề rất thiết yếu khi các thuật toán được thực hiện. Biết một thuật toán sẽ đưa ra đáp số trong một micro giây, trong một phút hoặc trong một tỉ năm, hiển nhiên là hết sức quan trọng. Tương tự như vậy, dung lượng bộ nhớ đòi hỏi phải là khả dụng để giải một bài toán,vì vậy độ phức tạp không gian cũng cần phải tính đến.Vì việc xem xét độ phức tạp không gian gắn liền với các cấu trúc dữ liệu đặc biệt được dùng để thực hiện thuật toán nên ở đây ta sẽ tập trung xem xét độ phức tạp thời gian.

Độ phức tạp thời gian của một thuật toán có thể được biểu diễn qua số các phép toán được dùng bởi thuật toán đó khi các giá trị đầu vào có một kích thước xác định. Sở dĩ độ phức tạp thời gian được mô tả thông qua số các phép toán đòi hỏi thay vì thời gian thực của máy tính là bởi vì các máy tính khác nhau thực hiện các phép tính sơ cấp trong những khoảng thời gian khác nhau. Hơn nữa, phân tích tất cả các phép toán thành các phép tính bit sơ cấp mà máy tính sử dụng là điều rất phức tạp.

Thí dụ 3: Xét thuật toán tìm số lớn nhất trong dãy n số a_1, a_2, ..., a_n. Có thể coi kích thước của dữ liệu nhập là số lượng phần tử của dãy số, tức là n. Nếu coi mỗi lần so sánh hai số của thuật toán đòi hỏi một đơn vị thời gian (giây chẳng hạn) thì thời gian thực hiện thuật toán trong trường hợp xấu nhất là n-1 giây. Với dãy 64 số, thời gian thực hiện thuật toán nhiều lắm là 63 giây.

Thí dụ 4: Thuật toán về trò chơi "Tháp Hà Nội"

Trò chơi "Tháp Hà Nội" như sau: Có ba cọc A, B, C và 64 cái đĩa (có lỗ để đặt vào cọc), các đĩa có đường kính đôi một khác nhau. Nguyên tắc đặt đĩa vào cọc là: mỗi đĩa chỉ được chồng lên đĩa lớn hơn nó. Ban đầu, cả 64 đĩa được đặt chồng lên nhau ở cột A; hai cột B, C trống. Vấn đề là phải chuyển cả 64 đĩa đó sang cột B hay C, mỗi lần chỉ được di chuyển một đĩa.

Xét trò chơi với n đĩa ban đầu ở cọc A (cọc B và C trống). Gọi S_n là số lần chuyển đĩa để chơi xong trò chơi với n đĩa.

Nếu n=1 thì rõ ràng là $S_1=1$.

Nếu n>1 thì trước hết ta chuyển n-1 đĩa bên trên sang cọc B (giữ yên đĩa thứ n ở dưới cùng của cọc A). Số lần chuyển n-1 đĩa là S_{n-1}. Sau đó ta chuyển đĩa thứ n từ cọc A sang cọc C. Cuối cùng, ta chuyển n-1 đĩa từ cọc B sang cọc C (số lần chuyển là S_{n-1}).

Như vậy, số lần chuyển n đĩa từ A sang C là:

$S_n=S_{n-1}+1+S_n=2S_{n-1}+1=2(2S_{n-2}+1)+1=2^2S_{n-2}+2+1=.....=2^{n-1}S_1+2^{n-2}+...+2+1=2^n-1$.

Thuật toán về trò chơi "Tháp Hà Nội" đòi hỏi $2^{64}-1$ lần chuyển đĩa (xấp xỉ 18,4 tỉ tỉ lần). Nếu mỗi lần chuyển đĩa mất 1 giây thì thời gian thực hiện thuật toán xấp xỉ 585 tỉ năm!

Hai thí dụ trên cho thấy rằng: một thuật toán phải kết thúc sau một số hữu hạn bước, nhưng nếu số hữu hạn này quá lớn thì thuật toán không thể thực hiện được trong thực tế.

Ta nói: thuật toán trong Thí dụ 3 có độ phức tạp là n-1 và là một thuật toán hữu hiệu (hay thuật toán nhanh); thuật toán trong Thí dụ 4 có độ phức tạp là 2^n-1 và đó là một thuật toán không hữu hiệu (hay thuật toán chậm).

1.3.2. So sánh độ phức tạp của các thuật toán:

Một bài toán thường có nhiều cách giải, có nhiều thuật toán để giải, các thuật toán đó có độ phức tạp khác nhau.

Xét bài toán: Tính giá trị của đa thức $P(x)=a_nx^n+a_{n-1}x^{n-1}+ ... +a_1x+a_0$ tại x_0.

Thuật toán 1:

Procedure tính giá trị của đa thức (a_0, a_1, ..., a_n, x_0: các số thực)

sum:=a_0

for i:=1 **to** n

sum:=sum+$a_ix_0^i$

{sum là giá trị của đa thức P(x) tại x_0}

Chú ý rằng đa thức P(x) có thể viết dưới dạng:

$P(x)=(...((a_nx+a_{n-1})x+a_{n-2})x...)x+a_0$.

Ta có thể tính P(x) theo thuật toán sau:

Thuật toán 2:

Procedure tính giá trị của đa thức (a_0, a_1, ..., a_n, x_0: các số thực)

P:=a_n

for i:=1 **to** n

P:=P.x_0+a_{n-i}

{P là giá trị của đa thức P(x) tại x_0}

Ta hãy xét độ phức tạp của hai thuật toán trên.

Đối với thuật toán 1: ở bước 2, phải thực hiện 1 phép nhân và 1 phép cộng với i=1; 2 phép nhân và 1 phép cộng với i=2, ..., n phép nhân và 1 phép cộng với i=n. Vậy số phép tính (nhân và cộng) mà thuật toán 1 đòi hỏi là:

$$(1+1)+(2+1)+ ... +(n+1)=\frac{n(n+1)}{2}+n=\frac{n(n+3)}{2}.$$

Đối với thuật toán 2, bước 2 phải thực hiện n lần, mỗi lần đòi hỏi 2 phép tính (nhân rồi cộng), do đó số phép tính (nhân và cộng) mà thuật toán 2 đòi hỏi là 2n.

Nếu coi thời gian thực hiện mỗi phép tính nhân và cộng là như nhau và là một đơn vị thời gian thì với mỗi n cho trước, thời gian thực hiện thuật toán 1 là n(n+3)/2, còn thời gian thực hiện thuật toán 2 là 2n.

Rõ ràng là thời gian thực hiện thuật toán 2 ít hơn so với thời gian thực hiện thuật toán 1. Hàm $f_1(n)=2n$ là hàm bậc nhất, tăng chậm hơn nhiều so với hàm bậc hai $f_2(n)=n(n+3)/2$.

Ta nói rằng thuật toán 2 (có độ phức tạp là 2n) là thuật toán hữu hiệu hơn (hay nhanh hơn) so với thuật toán 1 (có độ phức tạp là n(n+3)/2).

Để so sánh độ phức tạp của các thuật toán, điều tiện lợi là coi độ phức tạp của mỗi thuật toán như là cấp của hàm biểu hiện thời gian thực hiện thuật toán ấy.

Các hàm xét sau đây đều là hàm của biến số tự nhiên n>0.

Định nghĩa 1: Ta nói hàm f(n) có cấp thấp hơn hay bằng hàm g(n) nếu tồn tại hằng số C>0 và một số tự nhiên n_0 sao cho

$$|f(n)| \le C|g(n)| \text{ với mọi } n \ge n_0.$$

Ta viết f(n)=O(g(n)) và còn nói f(n) thoả mãn quan hệ big-O đối với g(n).

Theo định nghĩa này, hàm g(n) là một hàm đơn giản nhất có thể được, đại diện cho "sự biến thiên" của f(n).

Khái niệm big-O đã được dùng trong toán học đã gần một thế kỷ nay. Trong tin học, nó được sử dụng rộng rãi để phân tích các thuật toán. Nhà toán học người Đức Paul Bachmann là người đầu tiên đưa ra khái niệm big-O vào năm 1892.

Thí dụ 5: Hàm $f(n)=\frac{n(n+3)}{2}$ là hàm bậc hai và hàm bậc hai đơn giản nhất là n^2. Ta có:

$$f(n)=\frac{n(n+3)}{2}=O(n^2) \text{ vì } \frac{n(n+3)}{2} \le n^2 \text{ với mọi } n\ge 3 \ (C=1,\ n_0=3).$$

Một cách tổng quát, nếu $f(n)=a_kn^k+a_{k-1}n^{k-1}+ \dots +a_1n+a_0$ thì $f(n)=O(n^k)$. Thật vậy, với n>1,

$$|f(n)|| \le |a_k|n^k+|a_{k-1}|n^{k-1}+ \dots +|a_1|n+|a_0| = n^k(|a_k|+|a_{k-1}|/n+ \dots +|a_1|/n^{k-1}+a_0/n^k)$$
$$\le n^k(|a_k|+|a_{k-1}|+ \dots +|a_1|+a_0).$$

Điều này chứng tỏ $|f(n)| \le Cn^k$ với mọi n>1.

Cho $g(n)=3n+5n\log_2 n$, ta có $g(n)=O(n\log_2 n)$. Thật vậy,

$3n+5n\log_2 n = n(3+5\log_2 n) \le n(\log_2 n+5\log_2 n) = 6n\log_2 n$ với mọi $n\ge 8$ ($C=6,\ n_0=8$).

Mệnh đề: Cho $f_1(n)=O(g_1(n))$ và $f_2(n)$ là $O(g_2(n))$. Khi đó

$$(f_1 + f_2)(n) = O(\max(|g_1(n)|,|g_2(n)|)),\ \ (f_1f_2)(n) = O(g_1(n)g_2(n)).$$

Chứng minh. Theo giả thiết, tồn tại C_1, C_2, n_1, n_2 sao cho

$$|f_1(n)| \le C_1|g_1(n)| \text{ và } |f_2(n)| \le C_2|g_2(n)| \text{ với mọi } n > n_1 \text{ và mọi } n > n_2.$$

Do đó $|(f_1 + f_2)(n)| = |f_1(n) + f_2(n)| \le |f_1(n)| + |f_2(n)| \le C_1|g_1(n)| + C_2|g_2(n)| \le (C_1+C_2)g(n)$ với mọi $n > n_0=\max(n_1,n_2)$, ở đâyC=C_1+C_2 và $g(n)=\max(|g_1(n)| , |g_2(n)|)$.

$|(f_1f_2)(n)| = |f_1(n)||f_2(n)| \le C_1|g_1(n)|C_2|g_2(n)| \le C_1C_2|(g_1g_2)(n)|$ với mọi $n > n_0=\max(n_1,n_2)$.

Định nghĩa 2: Nếu một thuật toán có độ phức tạp là f(n) với f(n)=O(g(n)) thì ta cũng nói thuật toán có độ phức tạp O(g(n)).

Nếu có hai thuật toán giải cùng một bài toán, thuật toán 1 có độ phức tạp $O(g_1(n))$, thuật toán 2 có độ phức tạp $O(g_2(n))$, mà $g_1(n)$ có cấp thấp hơn $g_2(n)$, thì ta nói rằng thuật toán 1 hữu hiệu hơn (hay nhanh hơn) thuật toán 2.

1.3.3. Đánh giá độ phức tạp của một thuật toán:

1) Thuật toán tìm kiếm tuyến tính:

Số các phép so sánh được dùng trong thuật toán này cũng sẽ được xem như thước đo độ phức tạp thời gian của nó. Ở mỗi một bước của vòng lặp trong thuật toán, có hai phép so sánh được thực hiện: một để xem đã tới cuối bảng chưa và một để so sánh phần tử x với một số hạng của bảng. Cuối cùng còn một phép so sánh nữa làm ở ngoài vòng lặp. Do đó, nếu $x=a_i$, thì đã có 2i+1 phép so sánh được sử dụng. Số phép so sánh nhiều nhất, 2n+2, đòi hỏi phải được sử dụng khi phần tử x không có mặt trong bảng. Từ đó, thuật toán tìm kiếm tuyến tính có độ phức tạp là O(n).

2) Thuật toán tìm kiếm nhị phân:

Để đơn giản, ta giả sử rằng có $n=2^k$ phần tử trong bảng liệt kê $a_1,a_2,...,a_n$, với k là số nguyên không âm (nếu n không phải là lũy thừa của 2, ta có thể xem bảng là một phần của bảng gồm 2^{k+1} phần tử, trong đó k là số nguyên nhỏ nhất sao cho $n < 2^{k+1}$).

Ở mỗi giai đoạn của thuật toán vị trí của số hạng đầu tiên i và số hạng cuối cùng j của bảng con hạn chế tìm kiếm ở giai đoạn đó được so sánh để xem bảng con này còn nhiều hơn một phần tử hay không. Nếu i < j, một phép so sánh sẽ được làm để xác định x có lớn hơn số hạng ở giữa của bảng con hạn chế hay không. Như vậy ở mỗi giai đoạn, có sử dụng hai phép so sánh. Khi trong bảng chỉ còn một phần tử, một phép so sánh sẽ cho chúng ta biết rằng không còn một phần tử nào thêm nữa và một phép so sánh nữa cho biết số hạng đó có phải là x hay không. Tóm lại cần phải có nhiều nhất $2k+2=2\log_2 n+2$ phép so sánh để thực hiện phép tìm kiếm nhị phân (nếu n không phải là lũy thừa của 2, bảng gốc sẽ được mở rộng tới bảng có 2^{k+1} phần tử, với $k=[\log_2 n]$ và sự tìm kiếm đòi hỏi phải thực hiện nhiều nhất $2[\log_2 n]+2$ phép so sánh). Do đó thuật toán tìm kiếm nhị phân có độ phức tạp là $O(\log_2 n)$. Từ sự phân tích ở trên suy ra rằng thuật toán tìm kiếm nhị phân, ngay cả trong trường hợp xấu nhất, cũng hiệu quả hơn thuật toán tìm kiếm tuyến tính.

3) Chú ý: Một điều quan trọng cần phải biết là máy tính phải cần bao lâu để giải xong một bài toán. Thí dụ, nếu một thuật toán đòi hỏi 10 giờ, thì có thể còn đáng chi phí thời gian máy tính đòi hỏi để giải bài toán đó. Nhưng nếu một thuật toán đòi hỏi 10 tỉ năm để giải một bài toán, thì thực hiện thuật toán đó sẽ là một điều phi lý. Một trong những hiện tượng lý thú nhất của công nghệ hiện đại là sự tăng ghê gớm của tốc độ và lượng bộ nhớ trong máy tính. Một nhân tố quan trọng khác làm giảm thời gian cần thiết để giải một

bài toán là sự xử lý song song - đây là kỹ thuật thực hiện đồng thời các dãy phép tính. Do sự tăng tốc độ tính toán và dung lượng bộ nhớ của máy tính, cũng như nhờ việc dùng các thuật toán lợi dụng được ưu thế của kỹ thuật xử lý song song, các bài toán vài năm trước đây được xem là không thể giải được, thì bây giờ có thể giải bình thường.

1. Các thuật ngữ thường dùng cho độ phức tạp của một thuật toán:

Độ phức tạp	Thuật ngữ
O(1)	Độ phức tạp hằng số
O(logn)	Độ phức tạp lôgarit
O(*n*)	Độ phức tạp tuyến tính
O(*nlogn*)	Độ phức tạp *nlogn*
$O(n^b)$	Độ phức tạp đa thức
$O(b^n)$ (b>1)	Độ phức tạp hàm mũ
O(n!)	Độ phức tạp giai thừa

2. Thời gian máy tính được dùng bởi một thuật toán:

Kích thước của bài toán	Các phép tính bit được sử dụng					
n	logn	*N*	*nlogn*	n^2	2^n	*n*!
10	3.10^{-9} s	10^{-8} s	3.10^{-8} s	10^{-7} s	10^{-6} s	3.10^{-3} s
10^2	7.10^{-9} s	10^{-7} s	7.10^{-7} s	10^{-5} s	4.10^{13}năm	*
10^3	$1,0.10^{-8}$ s	10^{-6} s	1.10^{-5} s	10^{-3} s	*	*
10^4	$1,3.10^{-8}$ s	10^{-5} s	1.10^{-4} s	10^{-1} s	*	*
10^5	$1,7.10^{-8}$ s	10^{-4} s	2.10^{-3} s	10 s	*	*
10^6	2.10^{-8} s	10^{-3} s	2.10^{-2} s	17 phút	*	*

1.4. SỐ NGUYÊN VÀ THUẬT TOÁN.

1.4.1. Thuật toán Euclide:

Phương pháp tính ước chung lớn nhất của hai số bằng cách dùng phân tích các số nguyên đó ra thừa số nguyên tố là không hiệu quả. Lý do là ở chỗ thời gian phải tiêu tốn cho sự phân tích đó. Dưới đây là phương pháp hiệu quả hơn để tìm ước số chung lớn nhất, gọi là **thuật toán Euclide**. Thuật toán này đã biết từ thời cổ đại. Nó mang tên nhà toán học cổ Hy lạp Euclide, người đã mô tả thuật toán này trong cuốn sách "*Những yếu tố*" nổi tiếng của ông. Thuật toán Euclide dựa vào 2 mệnh đề sau đây.

Mệnh đề 1 (Thuật toán chia): Cho a và b là hai số nguyên và b≠0. Khi đó tồn tại duy nhất hai số nguyên q và r sao cho

$$a = bq+r, \quad 0 \le r < |b|.$$

Trong đẳng thức trên, b được gọi là số chia, a được gọi là số bị chia, q được gọi là thương số và r được gọi là số dư.

Khi b là nguyên dương, ta ký hiệu số dư r trong phép chia a cho b là a mod b.

Mệnh đề 2: Cho a = bq + r, trong đó a, b, q, r là các số nguyên. Khi đó

$$UCLN(a,b) = UCLN(b,r).$$

(Ở đây UCLN(a,b) để chỉ ước chung lớn nhất của a và b.)

Giả sử a và b là hai số nguyên dương với $a \geq b$. Đặt $r_0 = a$ và $r_1 = b$. Bằng cách áp dụng liên tiếp thuật toán chia, ta tìm được:

$r_0 = r_1q_1 + r_2$ $\quad 0 \leq r_2 < r_1$

$r_1 = r_2q_2 + r_3$ $\quad 0 \leq r_3 < r_2$

.....

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

$r_{n-1} = r_nq_n$.

Cuối cùng, số dư 0 sẽ xuất hiện trong dãy các phép chia liên tiếp, vì dãy các số dư

$$a = r_0 > r_1 > r_2 > ... \geq 0$$

không thể chứa quá a số hạng được. Hơn nữa, từ Mệnh đề 2 ở trên ta suy ra:

$UCLN(a,b) = UCLN(r_0,r_1) = UCLN(r_1,r_2) = ... = UCLN(r_{n-2}, r_{n-1}) = UCLN(r_{n-1},r_n) = r_n.$

Do đó, ước chung lớn nhất là số dư khác không cuối cùng trong dãy các phép chia.

Thí dụ 6: Dùng thuật toán Euclide tìm UCLN(414, 662).

662 = 441.1 + 248

414 = 248.1 + 166

248 = 166.1+ 82

166 = 82.2 + 2

82 = 2.41.

Do đó, UCLN(414, 662) = 2.

Thuật toán Euclide được viết dưới dạng giả mã như sau:

```
procedure ƯCLN (a,b: positive integers)
    x :=  a
    y := b
    while y ≠ 0
    begin
        r  := x mod y
        x := y
        y := r
    end
{UCLN (a,b) là x}
```

Trong thuật toán trên, các giá trị ban đầu của x và y tương ứng là a và b. Ở mỗi giai đoạn của thủ tục, x được thay bằng y và y được thay bằng x mod y. Quá trình này được lặp lại chừng nào $y \neq 0$. Thuật toán sẽ ngừng khi y = 0 và giá trị của x ở điểm này, đó là số dư khác không cuối cùng trong thủ tục, cũng chính là ước chung lớn nhất của a và b.

1.4.2. Biểu diễn các số nguyên:

Mệnh đề 3: Cho b là một số nguyên dương lớn hơn 1. Khi đó nếu n là một số nguyên dương, nó có thể được biểu diễn một cách duy nhất dưới dạng:

$$n = a_k b^k + a_{k-1} b^{k-1} + ... + a_1 b + a_0.$$

Ở đây k là một số tự nhiên, $a_0, a_1, ..., a_k$ là các số tự nhiên nhỏ hơn b và $a_k \neq 0$.

Biểu diễn của n được cho trong Mệnh đề 3 được gọi là khai triển của n theo cơ số b, ký hiệu là $(a_k a_{k-1} ... a_1 a_0)_b$. Bây giờ ta sẽ mô tả thuật toán xây dựng khai triển cơ số b của số nguyên n bất kỳ. Trước hết ta chia n cho b để được thương và số dư, tức là

$$n = bq_0 + a_0, \ 0 \leq a_0 < b.$$

Số dư a_0 chính là chữ số đứng bên phải cùng trong khai triển cơ số b của n. Tiếp theo chia q_0 cho b, ta được:

$$q_0 = bq_1 + a_1, \ 0 \leq a_1 < b.$$

Số dư a_1 chính là chữ số thứ hai tính từ bên phải trong khai triển cơ số b của n. Tiếp tục quá trình này, bằng cách liên tiếp chia các thương cho b ta sẽ được các chữ số tiếp theo trong khai triển cơ số b của n là các số dư tương ứng. Quá trình này sẽ kết thúc khi ta nhận được một thương bằng 0.

Thí dụ 7: Tìm khai triển cơ số 8 của $(12345)_{10}$.

$12345 = 8.1543 + 1$
$1543 = 8.192 + 7$
$192 = 8.24 + 0$
$24 = 8.3 + 0$
$3 = 8.0 + 3.$

Do đó, $(12345)_{10} = (30071)_8$.

Đoạn giả mã sau biểu diễn thuật toán tìm khai triển cơ số b của số nguyên n.

---

procedure *khai triển theo cơ số b* (n: positive integer)
    q := n
    k := 0
    while $q \neq 0$
    begin
        a_k := q **mod** b
        q := $[\frac{q}{b}]$
        k := k + 1
    end

1.4.3. Thuật toán cho các phép tính số nguyên:

Các thuật toán thực hiện các phép tính với những số nguyên khi dùng các khai triển nhị phân của chúng là cực kỳ quan trọng trong số học của máy tính. Ta sẽ mô tả ở

đây các thuật toán cộng và nhân hai số nguyên trong biểu diễn nhị phân. Ta cũng sẽ phân tích độ phức tạp tính toán của các thuật toán này thông qua số các phép toán bit thực sự được dùng. Giả sử khai triển nhị phân của hai số nguyên dương a và b là:

$$a = (a_{n-1}a_{n-2} \dots a_1 a_0)_2 \text{ và } b = (b_{n-1} b_{n-2} \dots b_1 b_0)_2$$

sao cho a và b đều có n bit (đặt các bit 0 ở đầu mỗi khai triển đó, nếu cần).

1) Phép cộng: Xét bài toán cộng hai số nguyên viết ở dạng nhị phân. Thủ tục thực hiện phép cộng có thể dựa trên phương pháp thông thường là cộng cặp chữ số nhị phân với nhau (có nhớ) để tính tổng của hai số nguyên.

Để cộng a và b, trước hết cộng hai bit ở phải cùng của chúng, tức là:

$$a_0 + b_0 = c_0.2 + s_0.$$

Ở đây s_0 là bit phải cùng trong khai triển nhị phân của a+b, c_0 là số nhớ, nó có thể bằng 0 hoặc 1. Sau đó ta cộng hai bit tiếp theo và số nhớ

$$a_1 + b_1 + c_0 = c_1.2 + s_1.$$

Ở đây s_1 là bit tiếp theo (tính từ bên phải) trong khai triển nhị phân của a+b và c_1 là số nhớ. Tiếp tục quá trình này bằng cách cộng các bit tương ứng trong hai khai triển nhị phân và số nhớ để xác định bit tiếp sau tính từ bên phải trong khai triển nhị phân của tổng a+b. Ở giai đoạn cuối cùng, cộng a_{n-1}, b_{n-1} và c_{n-2} để nhận được $c_{n-1}.2+s_{n-1}$. Bit đứng đầu của tổng là $s_n=c_{n-1}$. Kết quả, thủ tục này tạo ra được khai triển nhị phân của tổng, cụ thể là $a+b = (s_n s_{n-1} s_{n-2} \dots s_1 s_0)_2$.

Thí dụ 8: Tìm tổng của $a = (11011)_2$ và $b = (10110)_2$.

$a_0 + b_0 = 1 + 0 = 0.2 + 1$ ($c_0 = 0$, $s_0 = 1$), $a_1 + b_1 + c_0 = 1 + 1 + 0 = 1.2 + 0$ ($c_1 = 1$, $s_1 = 0$), $a_2 + b_2 + c_1 = 0 + 1 + 1 = 1.2 + 0$ ($c_2 = 1$, $s_2 = 0$), $a_3 + b_3 + c_2 = 1 + 0 + 1 = 1.2 + 0$ ($c_3 = 1$, $s_3 = 0$), $a_4 + b_4 + c_3 = 1 + 1 + 1 = 1.2 + 1$ ($s_5 = c_4 = 1$, $s_4 = 1$).

Do đó, $a + b = (110001)_2$.

Thuật toán cộng có thể được mô tả bằng cách dùng đoạn giả mã như sau.

---

procedure cộng (a,b: positive integers)

c := 0

for j := 0 **to** n-1

begin

d := $\left\lfloor \frac{a_j + b_j + c}{2} \right\rfloor$

$s_j := a_j + b_j + c - 2d$

c := d

end

s_n := c

{khai triển nhị phân của tổng là $(s_n s_{n-1} \dots s_1 s_0)_2$}

Tổng hai số nguyên được tính bằng cách cộng liên tiếp các cặp bit và khi cần phải cộng cả số nhớ nữa. Cộng một cặp bit và số nhớ đòi ba hoặc ít hơn phép cộng các bit. Như vậy, tổng số các phép cộng bit được sử dụng nhỏ hơn ba lần số bit trong khai triển nhị phân. Do đó, độ phức tạp của thuật toán này là O(n).

2) Phép nhân: Xét bài toán nhân hai số nguyên viết ở dạng nhị phân. Thuật toán thông thường tiến hành như sau. Dùng luật phân phối, ta có:

$$ab = a\sum_{j=0}^{n-1} b_j 2^j = \sum_{j=0}^{n-1} a(b_j 2^j).$$

Ta có thể tính ab bằng cách dùng phương trình trên. Trước hết, ta thấy rằng ab_j=a nếu b_j=1 và ab_j=0 nếu b_j=0. Mỗi lần ta nhân một số hạng với 2 là ta dịch khai triển nhị phân của nó một chỗ về phía trái bằng cách thêm một số không vào cuối khai triển nhị phân của nó. Do đó, ta có thể nhận được $(ab_j)2^j$ bằng cách dịch khai triển nhị phân của ab_j đi j chỗ về phía trái, tức là thêm j số không vào cuối khai triển nhị phân của nó. Cuối cùng, ta sẽ nhận được tích ab bằng cách cộng n số nguyên $ab_j.2^j$ với j=0, 1, ..., n-1.

Thí dụ 9: Tìm tích của a = $(110)_2$ và b = $(101)_2$.

Ta có $ab_0.2^0 = (110)_2.1.2^0 = (110)_2$, $ab_1.2^1 = (110)_2.0.2^1 = (0000)_2$, $ab_2.2^2 = (110)_2.1.2^2 = (11000)_2$. Để tìm tích, hãy cộng $(110)_2$, $(0000)_2$ và $(11000)_2$. Từ đó ta có ab= $(11110)_2$.

Thủ tục trên được mô tả bằng đoạn giả mã sau:

```
procedure nhân (a,b: positive integers)
    for j := 0 to n-1
    begin
        if  bj = 1 then cj := a được dịch đi j chỗ
        else cj := 0
    end
{c0, c1,..., cn-1 là các tích riêng phần}
    p := 0
    for j := 0 to n-1
        p := p + cj
{p là giá trị của tích ab}
```

Thuật toán trên tính tích của hai số nguyên a và b bằng cách cộng các tích riêng phần c_0, c_1, c_2, ..., c_{n-1}. Khi b_j=1, ta tính tích riêng phần c_j bằng cách dịch khai triển nhị phân của a đi j bit. Khi b_j=0 thì không cần có dịch chuyển nào vì c_j=0. Do đó, để tìm tất cả n số nguyên $ab_j.2^j$ với j=0, 1, ..., n-1, đòi hỏi tối đa là

$$0 + 1 + 2 + ... + n-1 = \frac{n(n-1)}{2}$$

phép dịch chỗ. Vì vậy, số các dịch chuyển chỗ đòi hỏi là $O(n^2)$.

Để cộng các số nguyên ab_j từ j=0 đến n−1, đòi hỏi phải cộng một số nguyên n bit, một số nguyên n+1 bit, ... và một số nguyên 2n bit. Ta đã biết rằng mỗi phép cộng đó đòi hỏi O(n) phép cộng bit. Do đó, độ phức tạp của thuật toán này là $O(n^2)$.

1.5. THUẬT TOÁN ĐỆ QUY.

1.5.1. Khái niệm đệ quy:

Đôi khi chúng ta có thể quy việc giải bài toán với tập các dữ liệu đầu vào xác định về việc giải cùng bài toán đó nhưng với các giá trị đầu vào nhỏ hơn. Chẳng hạn, bài toán tìm UCLN của hai số a, b với a > b có thể rút gọn về bài toán tìm ƯCLN của hai số nhỏ hơn, a mod b và b. Khi việc rút gọn như vậy thực hiện được thì lời giải bài toán ban đầu có thể tìm được bằng một dãy các phép rút gọn cho tới những trường hợp mà ta có thể dễ dàng nhận được lời giải của bài toán. Ta sẽ thấy rằng các thuật toán rút gọn liên tiếp bài toán ban đầu tới bài toán có dữ liệu đầu vào nhỏ hơn, được áp dụng trong một lớp rất rộng các bài toán.

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 cũng như vậy nhưng có dữ liệu đầu vào nhỏ hơn.

Thí dụ 10: Tìm thuật toán đệ quy tính giá trị a^n với a là số thực khác không và n là số nguyên không âm.

Ta xây dựng thuật toán đệ quy nhờ định nghĩa đệ quy của a^n, đó là $a^{n+1}=a.a^n$ với n>0 và khi n=0 thì $a^0=1$. Vậy để tính a^n ta quy về các trường hợp có số mũ n nhỏ hơn, cho tới khi n=0.

---

procedure power (a: số thực khác không; n: số nguyên không âm)

    if n = 0 **then** power(a,n) := 1

    else power(a,n) := a * power(a,n-1)

Thí dụ 11: Tìm thuật toán đệ quy tính UCLN của hai số nguyên a,b không âm và a > b.

---

procedure UCLN (a,b: các số nguyên không âm, a > b)

    if b = 0 **then** UCLN (a,b) := a

    else UCLN (a,b) := UCLN (a **mod** b, b)

Thí dụ 12: Hãy biểu diễn thuật toán tìm kiếm tuyến tính như một thủ tục đệ quy.

Để tìm x trong dãy tìm kiếm $a_1,a_2,...,a_n$ trong bước thứ i của thuật toán ta so sánh x với a_i. Nếu x bằng a_i thì i là vị trí cần tìm, ngược lại thì việc tìm kiếm được quy về dãy có số phần tử ít hơn, cụ thể là dãy $a_{i+1},...,a_n$. Thuật toán tìm kiếm có dạng thủ tục đệ quy như sau.

Cho *search* (i,j,x) là thủ tục tìm số x trong dãy a_i, a_{i+1},..., a_j. Dữ liệu đầu vào là bộ ba (1,n,x). Thủ tục sẽ dừng khi số hạng đầu tiên của dãy còn lại là x hoặc là khi dãy còn lại chỉ có một phần tử khác x. Nếu x không là số hạng đầu tiên và còn có các số hạng khác thì lại áp dụng thủ tục này, nhưng dãy tìm kiếm ít hơn một phần tử nhận được bằng cách xóa đi phần tử đầu tiên của dãy tìm kiếm ở bước vừa qua.

procedure *search* (i,j,x)

if a_i = x **then** loacation := i

else if i = j **then** loacation := 0

else *search* (i+1,j,x)

Thí dụ 13: Hãy xây dựng phiên bản đệ quy của thuật toán tìm kiếm nhị phân.

Giả sử ta muốn định vị x trong dãy a_1, a_2, ..., a_n bằng tìm kiếm nhị phân. Trước tiên ta so sánh x với số hạng giữa $a_{[(n+1)/2]}$. Nếu chúng bằng nhau thì thuật toán kết thúc, nếu không ta chuyển sang tìm kiếm trong dãy ngắn hơn, nửa đầu của dãy nếu x nhỏ hơn giá trị giữa của của dãy xuất phát, nửa sau nếu ngược lại. Như vậy ta rút gọn việc giải bài toán tìm kiếm về việc giải cũng bài toán đó nhưng trong dãy tìm kiếm có độ dài lần lượt giảm đi một nửa.

---

procedure *binary search* (x,i,j)

m := [(i+j)/2]

if x = a_m **then** loacation := m

else if (x < a_m and i < m) **then** *binary search* (x,i,m-1)

else if (x > a_m and j > m) **then** *binary search* (x,m+1,j)

else loacation := 0

1.5.2. Đệ quy và lặp:

Thí dụ 14. Thủ tục đệ quy sau đây cho ta giá trị của n! với n là số nguyên dương.

---

procedure *factorial* (n: positive integer)

if n = 1 **then** *factorial(n)* := 1

else `factorial(n) := n * factorial(n-1)`

Có cách khác tính hàm giai thừa của một số nguyên từ định nghĩa đệ quy của nó. Thay cho việc lần lượt rút gọn việc tính toán cho các giá trị nhỏ hơn, ta có thể xuất phát từ giá trị của hàm tại 1và lần lượt áp dụng định nghĩa đệ quy để tìm giá trị của hàm tại các số nguyên lớn dần. Đó là thủ tục lặp.

---

procedure iterative factorial (n: positive integer)

x := 1

for i := 1 **to** n

x := i * x

{x là n!}

Thông thường để tính một dãy các giá trị được định nghĩa bằng đệ quy, nếu dùng phương pháp lặp thì số các phép tính sẽ ít hơn là dùng thuật toán đệ quy (trừ khi dùng các máy đệ quy chuyên dụng). Ta sẽ xem xét bài toán tính số hạng thứ n của dãy Fibonacci.

---

procedure *fibonacci* (n: nguyên không âm)

```
if n = 0 the fibonacci(n) := 0
else if  n = 1 then fibonacci(n) := 1
        else fibonacci(n) := fibonacci(n - 1) + fibonacci(n - 2)
```

Theo thuật toán này, để tìm 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, cứ tiếp tục như vậy cho tới khi f_0 và f_1 xuất hiện thì được thay bằng các giá trị của chúng theo định nghĩa. Do đó để tính f_n cần f_{n+1}-1 phép cộng.

Bây giờ ta sẽ tính các phép toán cần dùng để tính f_n khi sử dụng phương pháp lặp. Thủ tục này khởi tạo x là $f_0 = 0$ và y là $f_1 = 1$. Khi vòng lặp được duyệt qua tổng của x và y được gán cho biến phụ z. Sau đó x được gán giá trị của y và y được gán giá trị của z. Vậy sau khi đi qua vòng lặp lần 1, ta có $x = f_1$ và $y = f_0 + f_1 = f_2$. Khi qua vòng lặp lần n-1 thì $x = f_{n-1}$. Như vậy chỉ có n – 1 phép cộng được dùng để tìm f_n khi n > 1.

```
procedure Iterative fibonacci (n: nguyên không âm)
    if n = 0 then y := 0
    else
    begin
        x := 0 ; y := 1
        for i := 1 to n - 1
        begin
            z := x + y
            x := y ; y := z
        end
    end
{y là số Fibonacci thứ n}
```

Ta đã chỉ ra rằng số các phép toán dùng trong thuật toán đệ quy nhiều hơn khi dùng phương pháp lặp. Tuy nhiên đôi khi người ta vẫn thích dùng thủ tục đệ quy hơn ngay cả khi nó tỏ ra kém hiệu quả so với thủ tục lặp. Đặc biệt, có những bài toán chỉ có thể giải bằng thủ tục đệ quy mà không thể giải bằng thủ tục lặp.

BÀI TẬP CHƯƠNG I:

1. Tìm một số nguyên n nhỏ nhất sao cho f(x) là $O(x^n)$ đối với các hàm f(x) sau:

a) $f(x) = 2x^3 + x^2 \log x$.

b) $f(x) = 2x^3 + (\log x)^4$.

c) $f(x) = \dfrac{x^4 + x^2 + 1}{x^3 + 1}$

d) $f(x) = \dfrac{x^5 + 5\log x}{x^4 + 1}$.

2. Chứng minh rằng
a) $x^2 + 4x + 7$ là $O(x^3)$, nhưng x^3 không là $O(x^2 + 4x + 17)$.
b) $x\log x$ là $O(x^2)$, nhưng x^2 không là $O(x\log x)$.

3. Cho một đánh giá big-O đối với các hàm cho dưới đây. Đối với hàm g(x) trong đánh giá f(x) là O(g(x)), hãy chọn hàm đơn giản có bậc thấp nhất.
a) $n\log(n^2 + 1) + n^2\log n$.
b) $(n\log n + 1)^2 + (\log n + 1)(n^2 + 1)$.
c) $n^{2^n} + n^{n^2}$.

4. Cho H_n là số điều hoà thứ n:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}$$

Chứng minh rằng H_n là $O(\log n)$.

5. Lập một thuật toán tính tổng tất cả các số nguyên trong một bảng.

6. Lập thuật toán tính x_n với x là một số thực và n là một số nguyên.

7. Mô tả thuật toán chèn một số nguyên x vào vị trí thích hợp trong dãy các số nguyên $a_1, a_2, ..., a_n$ xếp theo thứ tự tăng dần.

8. Tìm thuật toán xác định vị trí gặp đầu tiên của phần tử lớn nhất trong bảng liệt kê các số nguyên, trong đó các số này không nhất thiết phải khác nhau.

9. Tìm thuật toán xác định vị trí gặp cuối cùng của phần tử nhỏ nhất trong bảng liệt kê các số nguyên, trong đó các số này không nhất thiết phải khác nhau.

10. Mô tả thuật toán đếm số các số 1 trong một xâu bit bằng cách kiểm tra mỗi bit của xâu để xác định nó có là bit 1 hay không.

11. *Thuật toán tìm kiếm tam phân*. Xác định vị trí của một phần tử trong một bảng liệt kê các số nguyên theo thứ tự tăng dần bằng cách tách liên tiếp bảng liệt kê đó thành ba bảng liệt kê con có kích thước bằng nhau (hoặc gần bằng nhau nhất có thể được) và giới hạn việc tìm kiếm trong một bảng liệt kê con thích hợp. Hãy chỉ rõ các bước của thuật toán đó.

12. Lập thuật toán tìm trong một dãy các số nguyên số hạng đầu tiên bằng một số hạng nào đó đứng trước nó trong dãy.