Th.s. NGUYỄN VĂN LINH

GIẢI THUẬT

Được biên soạn trong khuôn khổ dự án ASVIET002CNTT
"Tăng cường hiệu quả đào tạo và năng lực tự đào tạo của sinh viên khoa Công nghệ Thông tin - Đại học Cần thơ"

ĐẠI HỌC CẦN THƠ - 12/2003

LỜI NÓI ĐẦU

N. Wirth, một nhà khoa học máy tính nổi tiếng, tác giả của ngôn ngữ lập trình Pascal, đã đặt tên cho một cuốn sách của ông là

"Cấu trúc dữ liệu + Giải thuật = Chương trình".

Điều đó nói lên tầm quan trọng của giải thuật trong lập trình nói riêng và trong khoa học máy tính nói chung. Vì lẽ đó giải thuật, với tư cách là một môn học, cần phải được sinh viên chuyên ngành tin học nghiên cứu một cách có hệ thống.

Môn học "Giải thuật" được bố trí sau môn "Cấu trúc dữ liệu" trong chương trình đào tạo kỹ sư tin học nhằm giới thiệu cho sinh viên những kiến thức cơ bản nhất, những kỹ thuật chủ yếu nhất của việc PHÂN TÍCH và THIẾT KẾ giải thuật. Các kỹ thuật được trình bày ở đây đã được các nhà khoa học tin học tổng kết và vận dụng trong cài đặt các chương trình. Việc nắm vững các kỹ thuật đó sẽ rất bổ ích cho sinh viên khi phải giải quyết một vấn đề thực tế.

Giáo trình này được hình thành trên cơ sở tham khảo cuốn sách "Data Structure and Algorithms" của A.V Aho, những kinh nghiệm giảng dạy của bản thân và các bạn đồng nghiệp.

Mặc dù đã có nhiều cố gắng trong quá trình biên soạn nhưng chắc chắn còn nhiều thiếu sót, rất mong nhận được sự đóng góp của quý bạn đọc.

Cần thơ, ngày 8 tháng 12 năm 2003

Nguyễn Văn Linh

MỤC LỤC

PHẦN TỔNG QUAN

1. Mục đích yêu cầu

Môn học giải thuật cung cấp cho sinh viên một khối lượng kiến thức tương đối hoàn chỉnh về phân tích và thiết kế các giải thuật lập trình cho máy tính. Sau khi học xong môn học này, sinh viên cần:

- Nắm được khái niệm thời gian thực hiện của chương trình, độ phức tạp của giải thuật. Biết cách phân tích, đánh giá giải thuật thông qua việc tính độ phức tạp.
- Nắm được các giải thuật sắp xếp và phân tích đánh giá được các giải thuật sắp xếp.
- Nắm được các kĩ thuật thiết kế giải thuật, vận dụng vào việc giải một số bài toán thực tế.
- Nắm được các phương pháp tổ chức lưu trữ thông tin trong tập tin và các giải thuật tìm, xen, xoá thông tin trong tập tin.

2. Đối tượng sử dụng

Môn học giải thuật được dùng để giảng dạy cho các sinh viên sau:

- Sinh viên năm thứ 3 chuyên ngành Tin học.
- Sinh viên năm thứ 3 chuyên ngành Điện tử (Viễn thông, Tự động hoá…)
- Sinh viên Toán-Tin.

3. Nội dung cốt lõi

Trong khuôn khổ 45 tiết, giáo trình được cấu trúc thành 4 chương

- **Chương 1:** Kĩ thuật phân tích đánh giá giải thuật. Chương này đặt vấn đề tại sao cần phải phân tích, đánh giá giải thuật và phân tích đánh giá theo phương pháp nào. Nội dung chương 1 tập trung vào khái niệm độ phức tạp thời gian của giải thuật và phương pháp tính độ phức tạp giải thuật của một chương trình bình thường, của chương trình có gọi các chương trình con và của các chương trình đệ quy.
- **Chương 2:** Sắp xếp. Chương này trình bày các giải thuật sắp xếp, một thao tác thường được sử dụng trong việc giải các bài toán máy tính. Sẽ có nhiều giải thuật sắp xếp từ đơn giản đến nâng cao sẽ được giới thiệu ở đây. Với mỗi giải thuật, sẽ trình bày ý tưởng giải thuật, ví dụ minh hoạ, cài đặt chương trình và phân tích đánh giá.
- **Chương 3:** Kĩ thuật thiết kế giải thuật. Chương này trình bày các kĩ thuật phổ biến để thiết kế các giải thuật. Các kĩ thuật này gồm: Chia để trị, Quy hoạch động, Tham ăn, Quay lui và Tìm kiếm địa phương. Với mỗi kĩ thuật sẽ trình bày nội dung kĩ thuật và vận dung vào giải các bài toán khá nổi tiếng như bài toán người giao hàng, bài toán cái ba lô, bài toán cây phủ tối thiểu...
- **Chương 4:** Cấu trúc dữ liệu và giải thuật lưu trữ ngoài. Chương này trình bày các cấu trúc dữ liệu được dùng để tổ chức lưu trữ tập tin trên bộ nhớ ngoài và các giải thuật tìm kiếm, xen xoá thông tin trên các tập tin đó.

4. Kiến thức tiên quyết

Để học tốt môn học giải thuật cần phải có các kiến thức sau:

- Kiến thức toán học.
- Kiến thức và kĩ năng lập trình căn bản.

- Kiến thức về cấu trúc dữ liệu và các giải thuật thao tác trên các cấu trúc dữ liệu.

Trong chương trình đào tạo, Cấu trúc dữ liệu là môn học tiên quyết của môn Giải thuật.

5. Danh mục tài liệu tham khảo

[1] **A.V. Aho, J.E. Hopcroft, J.D. Ullman**; *Data Structures and Algorithms*; Addison-Wesley; 1983.

[2] **Jeffrey H Kingston**; *Algorithms and Data Structures*; Addison-Wesley; 1998.

[3] **Đinh Mạnh Tường**; *Cấu trúc dữ liệu & Thuật toán*; Nhà xuất bản khoa học và kĩ thuật; Hà nội-2001.

[4] **Đỗ Xuân Lôi**; *Cấu trúc dữ liệu & Giải thuật*; 1995.

[5] **Nguyễn Đức Nghĩa, Tô Văn Thành**; *Toán rời rạc*; 1997.

[6] Trang web phân tích giải thuật: http://pauillac.inria.fr/algo/AofA/

[7] Trang web bài giảng về giải thuật:

http://www.cs.pitt.edu/~kirk/algorithmcourses/

[8] Trang tìm kiếm các giải thuật:

http://oopweb.com/Algorithms/Files/Algorithms.html

CHƯƠNG 1: KĨ THUẬT PHÂN TÍCH GIẢI THUẬT

1.1 TỔNG QUAN

1.1.1 Mục tiêu

Sau khi học chương này, sinh viên cần phải trả lời được các câu hỏi sau:

- Tại sao cần phân tích đánh giá giải thuật?
- Tiêu chuẩn nào để đánh giá một giải thuật là tốt?
- Phương pháp đánh giá như thế nào? (đánh giá chương trình không gọi chương trình con, đánh giá một chương trình có gọi các chương trình con không đệ quy và đánh giá chương trình đệ quy).

1.1.2 Kiến thức cơ bản cần thiết

Các kiến thức cơ bản cần thiết để học chương này bao gồm:

- Kiến thức toán học: Công thức tính tổng n số tự nhiên đầu tiên, công thức tính tổng n số hạng đầu tiên của một cấp số nhân, phương pháp chứng minh quy nạp và các kiến thức liên quan đến logarit (biến đổi logarit, tính chất đồng biến của hàm số logarit).
- Kĩ thuật lập trình và lập trình đệ quy.

1.1.3 Tài liệu tham khảo

A.V. Aho, J.E. Hopcroft, J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley. 1983. (Chapters 1, 9).
Jeffrey H Kingston; *Algorithms and Data Structures*; Addison-Wesley; 1998. (Chapter 2).

Đinh Mạnh Tường. *Cấu trúc dữ liệu & Thuật toán*. Nhà xuất bản khoa học và kĩ thuật. Hà nội-2001. (Chương 1).
Trang web phân tích giải thuật: http://pauillac.inria.fr/algo/AofA/

1.1.4 Nội dung cốt lõi

Trong chương này chúng ta sẽ nghiên cứu các vấn đề sau:

- Sự cần thiết phải phân tích các giải thuật.
- Thời gian thực hiện của chương trình.
- Tỷ suất tăng và độ phức tạp của giải thuật.
- Tính thời gian thực hiện của chương trình.
- Phân tích các chương trình đệ quy.

1.2 SỰ CẦN THIẾT PHẢI PHÂN TÍCH GIẢI THUẬT

Trong khi giải một bài toán chúng ta có thể có một số giải thuật khác nhau, vấn đề là cần phải đánh giá các giải thuật đó để lựa chọn một giải thuật tốt (nhất). Thông thường thì ta sẽ căn cứ vào các tiêu chuẩn sau:

1.- Giải thuật đúng đắn.

2.- Giải thuật đơn giản.

3.- Giải thuật thực hiện nhanh.

Với yêu cầu (1), để kiểm tra tính đúng đắn của giải thuật chúng ta có thể cài đặt giải thuật đó và cho thực hiện trên máy với một số bộ dữ liệu mẫu rồi lấy kết quả thu được so sánh với kết quả đã biết. Thực ra thì cách làm này không chắc chắn bởi vì có thể giải thuật đúng với tất cả các bộ dữ liệu chúng ta đã thử nhưng lại sai với một bộ dữ liệu nào đó. Vả lại cách làm này chỉ phát hiện ra giải thuật sai chứ chưa chứng minh được là nó đúng. Tính đúng đắn của giải thuật cần phải được chứng minh bằng toán học. Tất nhiên điều này không đơn giản và do vậy chúng ta sẽ không đề cập đến ở đây.

Khi chúng ta viết một chương trình để sử dụng một vài lần thì yêu cầu (2) là quan trọng nhất. Chúng ta cần một giải thuật dễ viết chương trình để nhanh chóng có được kết quả , thời gian thực hiện chương trình không được đề cao vì dù sao thì chương trình đó cũng chỉ sử dụng một vài lần mà thôi.

Tuy nhiên khi một chương trình được sử dụng nhiều lần thì thì yêu cầu tiết kiệm thời gian thực hiện chương trình lại rất quan trọng đặc biệt đối với những chương trình mà khi thực hiện cần dữ liệu nhập lớn do đó yêu cầu (3) sẽ được xem xét một cách kĩ càng. Ta gọi nó là hiệu quả thời gian thực hiện của giải thuật.

1.3 THỜI GIAN THỰC HIỆN CỦA CHƯƠNG TRÌNH

Một phương pháp để xác định hiệu quả thời gian thực hiện của một giải thuật là lập trình nó và đo lường thời gian thực hiện của hoạt động trên một máy tính xác định đối với tập hợp được chọn lọc các dữ liệu vào.

Thời gian thực hiện không chỉ phụ thuộc vào giải thuật mà còn phụ thuộc vào tập các chỉ thị của máy tính, chất lượng của máy tính và kĩ xảo của người lập trình. Sự thi hành cũng có thể điều chỉnh để thực hiện tốt trên tập đặc biệt các dữ liệu vào được chọn. Để vượt qua các trở ngại này, các nhà khoa học máy tính đã chấp nhận tính phức tạp của thời gian được tiếp cận như một sự đo lường cơ bản sự thực thi của giải thuật. Thuật ngữ tính hiệu quả sẽ đề cập đến sự đo lường này và đặc biệt đối với sự phức tạp thời gian trong trường hợp xấu nhất.

1.3.1 Thời gian thực hiện chương trình.

Thời gian thực hiện một chương trình là một hàm của kích thước dữ liệu vào, ký hiệu T(n) trong đó n là kích thước (độ lớn) của dữ liệu vào.

Ví dụ 1-1: Chương trình tính tổng của n số có thời gian thực hiện là T(n) = cn trong đó c là một hằng số.

Thời gian thực hiện chương trình là một hàm không âm, tức là $T(n) \geq 0 \ \forall \ n \geq 0$.

1.3.2 Đơn vị đo thời gian thực hiện.

Đơn vị của T(n) không phải là đơn vị đo thời gian bình thường như giờ, phút giây... mà thường được xác định bởi số các lệnh được thực hiện trong một máy tính lý tưởng.

Ví dụ 1-2: Khi ta nói thời gian thực hiện của một chương trình là T(n) = Cn thì có nghĩa là chương trình ấy cần Cn chỉ thị thực thi.

1.3.3 Thời gian thực hiện trong trường hợp xấu nhất.

Nói chung thì thời gian thực hiện chương trình không chỉ phụ thuộc vào kích thước mà còn phụ thuộc vào tính chất của dữ liệu vào. Nghĩa là dữ liệu vào có cùng kích thước nhưng thời gian thực hiện chương trình có thể khác nhau. Chẳng hạn chương trình sắp xếp dãy số nguyên tăng dần, khi ta cho vào dãy có thứ tự thì thời gian thực hiện khác với khi ta cho vào dãy chưa có thứ tự, hoặc khi ta cho vào một dãy đã có thứ tự tăng thì thời gian thực hiện cũng khác so với khi ta cho vào một dãy đã có thứ tự giảm.

Vì vậy thường ta coi T(n) là thời gian thực hiện chương trình trong trường hợp xấu nhất trên dữ liệu vào có kích thước n, tức là: T(n) là thời gian lớn nhất để thực hiện chương trình đối với mọi dữ liệu vào có cùng kích thước n.

1.4 TỶ SUẤT TĂNG VÀ ĐỘ PHỨC TẠP CỦA GIẢI THUẬT

1.4.1 Tỷ suất tăng

Ta nói rằng hàm không âm T(n) có tỷ suất tăng (growth rate) f(n) nếu tồn tại các hằng số C và N_0 sao cho $T(n) \leq Cf(n)$ với mọi $n \geq N_0$.

Ta có thể chứng minh được rằng "Cho một hàm không âm T(n) bất kỳ, ta luôn tìm được tỷ suất tăng f(n) của nó".

Ví dụ 1-3: Giả sử T(0) = 1, T(1) = 4 và tổng quát $T(n) = (n+1)^2$. Đặt N0 = 1 và C = 4 thì với mọi $n \geq 1$ chúng ta dễ dàng chứng minh được rằng $T(n) = (n+1)^2 \leq 4n^2$ với mọi $n \geq 1$, tức là tỷ suất tăng của T(n) là n^2.

Ví dụ 1-4: Tỷ suất tăng của hàm $T(n) = 3n^3 + 2n^2$ là n^3. Thực vậy, cho N0 = 0 và C = 5 ta dễ dàng chứng minh rằng với mọi $n \geq 0$ thì $3n^3 + 2n^2 \leq 5n^3$

1.4.2 Khái niệm độ phức tạp của giải thuật

Giả sử ta có hai giải thuật P1 và P2 với thời gian thực hiện tương ứng là $T1(n) = 100n^2$ (với tỷ suất tăng là n^2) và $T2(n) = 5n^3$ (với tỷ suất tăng là n^3). Giải thuật nào sẽ thực hiện nhanh hơn? Câu trả lời phụ thuộc vào kích thước dữ liệu vào. Với $n < 20$ thì P2 sẽ nhanh hơn P1 (T2<T1), do hệ số của $5n^3$ nhỏ hơn hệ số của $100n^2$ (5<100). Nhưng khi $n > 20$ thì ngược lại do số mũ của $100n^2$ nhỏ hơn số mũ của $5n^3$ (2<3). Ở đây chúng ta chỉ nên quan tâm đến trường hợp n>20 vì khi n<20 thì thời gian thực hiện của cả P1 và P2 đều không lớn và sự khác biệt giữa T1 và T2 là không đáng kể.

Như vậy một cách hợp lý là ta xét tỷ suất tăng của hàm thời gian thực hiện chương trình thay vì xét chính bản thân thời gian thực hiện.

Cho một hàm T(n), T(n) gọi là có độ phức tạp f(n) nếu tồn tại các hằng C, N_0 sao cho $T(n) \leq Cf(n)$ với mọi $n \geq N_0$ (tức là T(n) có tỷ suất tăng là f(n)) và kí hiệu T(n) là O(f(n)) (đọc là "ô của f(n)")

Ví dụ 1-5: $T(n)= (n+1)^2$ có tỷ suất tăng là n^2 nên $T(n)= (n+1)^2$ là $O(n^2)$

Chú ý: O(C.f(n))=O(f(n)) với C là hằng số. Đặc biệt O(C)=O(1)

Nói cách khác độ phức tạp tính toán của giải thuật là một hàm chặn trên của hàm thời gian. Vì hằng nhân tử C trong hàm chặn trên không có ý nghĩa nên ta có thể bỏ qua vì vậy hàm thể hiện độ phức tạp có các dạng thường gặp sau: $\log_2 n$, n, $n\log_2 n$, n^2, n^3, 2^n, n!, n^n. Ba hàm cuối cùng ta gọi là dạng *hàm mũ*, các hàm khác gọi là *hàm đa thức*. Một giải thuật mà thời gian thực hiện có độ phức tạp là một hàm đa thức thì chấp nhận được tức là có thể cài đặt để thực hiện, còn các giải thuật có độ phức tạp hàm mũ thì phải tìm cách cải tiến giải thuật.

Vì ký hiệu $\log_2 n$ thường có mặt trong độ phức tạp nên trong khôn khổ tài liệu này, ta sẽ dùng **logn** thay thế cho **$\log_2 n$** với mục đích duy nhất là để cho gọn trong cách viết.

Khi nói đến độ phức tạp của giải thuật là ta muốn nói đến hiệu quả của thời gian thực hiện của chương trình nên ta có thể xem việc xác định thời gian thực hiên của chương trình chính là xác định độ phức tạp của giải thuật.

1.5 CÁCH TÍNH ĐỘ PHỨC TẠP

Cách tính độ phức tạp của một giải thuật bất kỳ là một vấn đề không đơn giản. Tuy nhiên ta có thể tuân theo một số nguyên tắc sau:

1.5.1 Qui tắc cộng

Nếu T1(n) và T2(n) là thời gian thực hiện của hai đoạn chương trình P1 và P2; và T1(n)=O(f(n)), T2(n)=O(g(n)) thì thời gian thực hiện của đoạn hai chương trình đó **nối tiếp nhau** là T(n)=O(max(f(n),g(n)))

Ví dụ 1-6: Lệnh gán x:=15 tốn một hằng thời gian hay O(1), Lệnh đọc dữ liệu READ(x) tốn một hằng thời gian hay O(1).Vậy thời gian thực hiện cả hai lệnh trên nối tiếp nhau là O(max(1,1))=O(1)

1.5.2 Qui tắc nhân

Nếu T1(n) và T2(n) là thời gian thực hiện của hai đoạn chương trình P1và P2 và T1(n) = O(f(n)), T2(n) = O(g(n)) thì thời gian thực hiện của đoạn hai đoạn chương trình đó **lồng nhau** là T(n) = O(f(n).g(n))

1.5.3 Qui tắc tổng quát để phân tích một chương trình:

- Thời gian thực hiện của mỗi lệnh gán, READ, WRITE là O(1)

- Thời gian thực hiện của một chuỗi tuần tự các lệnh được xác định bằng qui tắc cộng. Như vậy thời gian này là thời gian thi hành một lệnh nào đó lâu nhất trong chuỗi lệnh.
- Thời gian thực hiện cấu trúc IF là thời gian lớn nhất thực hiện lệnh sau THEN hoặc sau ELSE và thời gian kiểm tra điều kiện. Thường thời gian kiểm tra điều kiện là O(1).
- Thời gian thực hiện vòng lặp là tổng (trên tất cả các lần lặp) thời gian thực hiện thân vòng lặp. Nếu thời gian thực hiện thân vòng lặp không đổi thì thời gian thực hiện vòng lặp là tích của số lần lặp với thời gian thực hiện thân vòng lặp.

Ví dụ 1-7: Tính thời gian thực hiện của thủ tục sắp xếp "nổi bọt"

```
PROCEDURE Bubble(VAR a: ARRAY[1..n] OF integer);
VAR i,j,temp: Integer;
BEGIN
{1}  FOR i:=1 TO n-1 DO
{2}       FOR j:=n DOWNTO i+1 DO
{3}            IF a[j-1]>a[j]THEN BEGIN{hoán vị a[i], a[j]}
{4}               temp   := a[j-1];
{5}               a[j-1] := a[j];
{6}               a[j]   := temp;
                  END;
END;
```

Về giải thuật sắp xếp nổi bọt, chúng ta sẽ bàn kĩ hơn trong chương 2. Ở đây, chúng ta chỉ quan tâm đến độ phức tạp của giải thuật.

Ta thấy toàn bộ chương trình chỉ gồm một lệnh lặp {1}, lồng trong lệnh {1} là lệnh {2}, lồng trong lệnh {2} là lệnh {3} và lồng trong lệnh {3} là 3 lệnh nối tiếp nhau {4}, {5} và {6}. Chúng ta sẽ tiến hành tính độ phức tạp theo thứ tự từ trong ra.

Trước hết, cả ba lệnh gán {4}, {5} và {6} đều tốn O(1) thời gian, việc so sánh a[j-1] > a[j] cũng tốn O(1) thời gian, do đó lệnh {3} tốn O(1) thời gian.

Vòng lặp {2} thực hiện (n-i) lần, mỗi lần O(1) do đó vòng lặp {2} tốn O((n-i).1) = O(n-i).

Vòng lặp {1} lặp có I chạy từ 1 đến n-1nên thời gian thực hiện của vòng lặp {1} và cũng là độ phức tạp của giải thuật là

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

Chú ý: Trong trường hợp vòng lặp không xác định được số lần lặp thì chúng ta phải lấy số lần lặp trong trường hợp xấu nhất.

Ví dụ 1-8: Tìm kiếm tuần tự. Hàm tìm kiếm Search nhận vào một mảng a có n số nguyên và một số nguyên x, hàm sẽ trả về giá trị logic TRUE nếu tồn tại một phần tử a[i] = x, ngược lại hàm trả về FALSE.

Giải thuật tìm kiếm tuần tự là lần lượt so sánh x với các phần tử của mảng a, bắt đầu từ a[1], nếu tồn tại a[i] = x thì dừng và trả về TRUE, ngược lại nếu tất cả các phần tử của a đều khác X thì trả về FALSE.

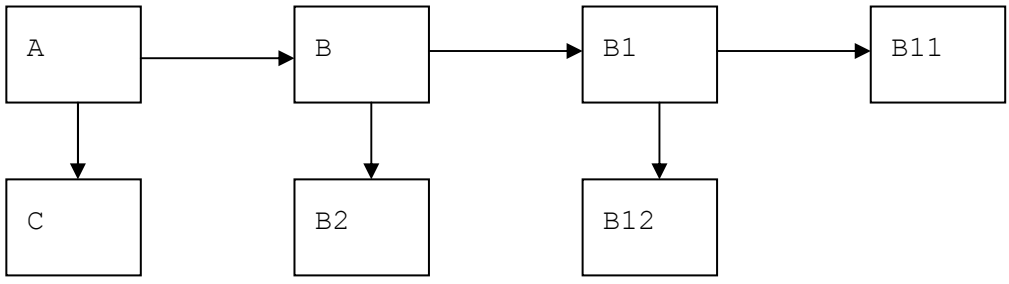
```
FUNCTION Search(a:ARRAY[1..n] OF Integer;x:Integer):Boolean;
VAR i:Integer; Found:Boolean;
BEGIN
{1}  i:=1;
{2}  Found:=FALSE;
{3}  WHILE(i<=n)AND (not Found) DO
{4}       IF A[i]=X THEN Found:=TRUE
          ELSE i:=i+1;
{5}  Search:=Found;
END;
```

Ta thấy các lệnh {1}, {2}, {3} và {5} nối tiếp nhau, do đó độ phức tạp của hàm Search chính là độ phức tạp lớn nhất trong 4 lệnh này. Dễ dàng thấy rằng ba lệnh {1}, {2} và {5} đều có độ phức tạp O(1) do đó độ phức tạp của hàm Search chính là độ phức tạp của lệnh {3}. Lồng trong lệnh {3} là lệnh {4}. Lệnh {4} có độ phức tạp O(1). Trong trường hợp xấu nhất (tất cả các phần tử của mảng a đều khác x) thì vòng lặp {3} thực hiện n lần, vậy ta có T(n) = O(n).

1.5.4 Độ phức tạp của chương trình có gọi chương trình con không đệ qui

Nếu chúng ta có một chương trình với các chương trình con không đệ quy, để tính thời gian thực hiện của chương trình, trước hết chúng ta tính thời gian thực hiện của các chương trình con không gọi các chương trình con khác. Sau đó chúng ta tính thời gian thực hiện của các chương trình con chỉ gọi các chương trình con mà thời gian thực hiện của chúng đã được tính. Chúng ta tiếp tục quá trình đánh giá thời gian thực hiện của mỗi chương trình con sau khi thời gian thực hiện của tất cả các chương trình con mà nó gọi đã được đánh giá. Cuối cùng ta tính thời gian cho chương trình chính.

Giả sử ta có một hệ thống các chương trình gọi nhau theo sơ đồ sau:



Hình 1-1: Sơ đồ gọi thực hiện các chương trình con không đệ quy

Chương trình A gọi hai chương trình con là B và C, chương trình B gọi hai chương trình con là B1 và B2, chương trình B1 gọi hai chương trình con là B11 và B12.

Để tính thời gian thực hiện của A, ta tính theo các bước sau:

1. Tính thời gian thực hiện của C, B2, B11 và B12. Vì các chương trình con này không gọi chương trình con nào cả.
2. Tính thời gian thực hiện của B1. Vì B1 gọi B11 và B12 mà thời gian thực hiện của B11 và B12 đã được tính ở bước 1.
3. Tính thời gian thực hiện của B. Vì B gọi B1 và B2 mà thời gian thực hiện của B1 đã được tính ở bước 2 và thời gian thực hiện của B2 đã được tính ở bước 1.
4. Tính thời gian thực hiện của A. Vì A gọi B và C mà thời gian thực hiện của B đã được tính ở bước 3 và thời gian thực hiện của C đã được tính ở bước 1.

Ví dụ 1-9: Ta có thể viết lại chương trình sắp xếp bubble như sau: Trước hết chúng ta viết thủ tục Swap để thực hiện việc hoàn đổi hai phần tử cho nhau, sau đso trong thủ tục Bubble, khi cần ta sẽ gọi đến thủ tục Swap này.

```
PROCEDURE Swap (VAR x, y: Integer);
VAR temp: Integer;
BEGIN
     temp := x;
     x    := y;
     y    := temp;
END;
PROCEDURE Bubble (VAR a: ARRAY[1..n] OF integer);
VAR i,j :Integer;
BEGIN
{1}  FOR i:=1 TO n-1 DO
{2}       FOR j:=n DOWNTO i+1 DO
{3}            IF a[j-1]>a[j] THEN Swap(a[j-1], a[j]);
END;
```

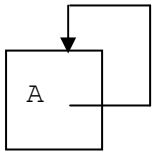
Trong cách viết trên, chương trình Bubble gọi chương trình con Swap, do đó để tính thời gian thực hiện của Bubble, trước hết ta cần tính thời gian thực hiện của Swap. Dễ thấy thời gian thực hiện của Swap là O(1) vì nó chỉ bao gồm 3 lệnh gán.

Trong Bubble, lệnh {3} gọi Swap nên chỉ tốn O(1), lệnh {2} thực hiện n-i lần, mỗi lần tốn O(1) nên tốn O(n-i). Lệnh {1} thực hiện n-1 lần nên

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

1.6 PHÂN TÍCH CÁC CHƯƠNG TRÌNH ĐỆ QUY

Với các chương trình có gọi các chương trình con đệ quy, ta không thể áp dụng cách tính như vừa trình bày trong mục 1.5.4 bởi vì một chương trình đệ quy sẽ gọi chính bản thân nó. Có thể thấy hình ảnh chương trình đệ quy A như sau:



Hình 1-2: Sơ đồ chương trình con A đệ quy

Với phương pháp tính độ phức tạp đã trình bày trong mục 1.5.4 thì không thể thực hiện được. Bởi vì nếu theo phương pháp đó thì, để tính thời gian thực hiên của chương trình A, ta phải tính thời gian thực hiện của chương trình A và cái vòng luẩn quẩn ấy không thể kết thúc được.

Với các chương trình đệ quy, trước hết ta cần thành lập các phương trình đệ quy, sau đó giải phương trình đệ quy, nghiệm của phương trình đệ quy sẽ là thời gian thực hiện của chương trình đệ quy.

1.6.1 Thành lập phương trình đệ quy

Phương trình đệ quy là một phương trình biểu diễn mối liên hệ giữa T(n) và T(k), trong đó T(n) là thời gian thực hiện chương trình với kích thước dữ liệu nhập là n, T(k) thời gian thực hiện chương trình với kích thước dữ liệu nhập là k, với k < n. Để thành lập được phương trình đệ quy, ta phải căn cứ vào chương trình đệ quy.

Thông thường một chương trình đệ quy để giải bài toán kích thước n, phải có ít nhất một trường hợp dừng ứng với một n cụ thể và lời gọi đệ quy để giải bài toán kích thước k (k<n).

Để thành lập phương trình đệ quy, ta gọi T(n) là thời gian để giải bài toán kích thước n, ta có T(k) là thời gian để giải bài toán kích thước k. Khi đệ quy dừng, ta phải xem xét khi đó chương trình làm gì và tốn hết bao nhiêu thời gian, chẳng hạn thời gian này là c(n). Khi đệ quy chưa dừng thì phải xét xem có bao nhiêu lời gọi đệ quy với kích thước k ta sẽ có bấy nhiêu T(k). Ngoài ra ta còn phải xem xét đến thời gian để phân chia bài toán và tổng hợp các lời giải, chẳng hạn thời gian này là d(n).

Dạng tổng quát của một phương trình đệ quy sẽ là:

$$T(n) = \begin{cases} C(n) \\ F(T(k)) + d(n) \end{cases}$$

Trong đó C(n) là thời gian thực hiện chương trình ứng với trường hợp đệ quy dừng. F(T(k)) là một đa thức của các T(k). d(n) là thời gian để phân chia bài toán và tổng hợp các kết quả.

Ví dụ 1-10: Xét hàm tính giai thừa viết bằng giải thuật đệ quy như sau:

```
FUNCTION Giai_thua(n:Integer): Integer;
BEGIN
     IF n=0 then Giai_thua :=1
     ELSE Giai_thua := n* Giai_thua(n-1);
END;
```

Gọi T(n) là thời gian thực hiện việc tính n giai thừa, thì T(n-1) là thời gian thực hiện việc tính n-1 giai thừa. Trong trường hợp n = 0 thì chương trình chỉ thực hiện một lệnh gán Giai_thua:=1, nên tốn O(1), do đó ta có $T(0) = C_1$. Trong trường hợp n>0 chương trình phải gọi đệ quy Giai_thua(n-1), việc gọi đệ quy này tốn T(n-1), sau khi có kết quả của việc gọi đệ quy, chương trình phải nhân kết quả đó với n và gán cho Giai_thua. Thời gian để thực hiện phép nhân và phép gán là một hằng C_2. Vậy ta có

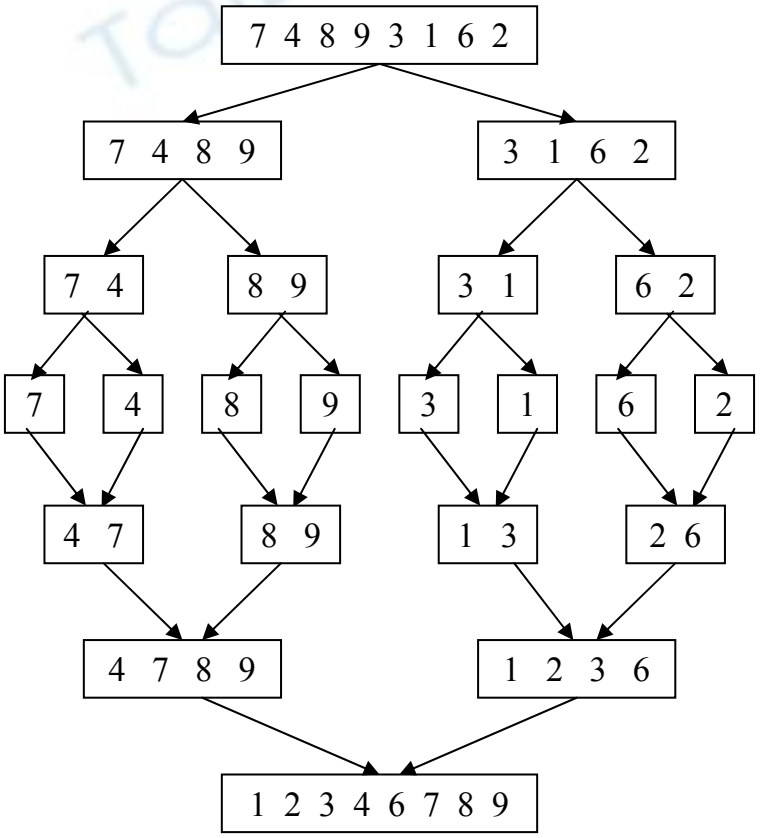
$$T(n) = \begin{cases} C_1 & \text{nêu } n = 0 \\ T(n-1) + C_2 & \text{nêu } n > 0 \end{cases}$$

Đây là phương trình đệ quy để tính thời gian thực hiện của chương trình đệ quy Giai_thua.

Ví du 1-11: Chúng ta xét thủ tục MergeSort một cách phác thảo như sau:

```
FUNCTION MergeSort (L:List; n:Integer):List;
VAR  L1,L2:List;
BEGIN
  IF n=1 THEN RETURN(L)
  ELSE BEGIN
     Chia đôi L thành L1 và L2, với độ dài n/2;
     RETURN(Merge(MergeSort(L1,n/2),MergeSort(L2,n/2)));
   END;
END;
```

Chẳng hạn để sắp xếp danh sách L gồm 8 phần tử 7, 4, 8, 9, 3, 1, 6, 2 ta có mô hình minh họa của MergeSort như sau:



Hình 1-3: Minh hoạ sắp xếp trộn

Hàm MergeSort nhận một danh sách có độ dài n và trả về một danh sách đã được sắp xếp. Thủ tục Merge nhận hai danh sách đã được sắp L1 và L2 mỗi danh sách có độ dài $\frac{n}{2}$, trộn chúng lại với nhau để được một danh sách gồm n phần tử có thứ tự.

Giải thuật chi tiết của Merge ta sẽ bàn sau, chúng ta chỉ để ý rằng thời gian để Merge các danh sách có độ dài $\frac{n}{2}$ là O(n).

Gọi T(n) là thời gian thực hiện MergeSort một danh sách n phần tử thì $T(\frac{n}{2})$ là thời gian thực hiện MergeSort một danh sách $\frac{n}{2}$ phần tử.

Khi L có độ dài 1 (n = 1) thì chương trình chỉ làm một việc duy nhất là return(L), việc này tốn $O(1) = C_1$ thời gian. Trong trường hợp n > 1, chương trình phải thực hiện gọi đệ quy MergeSort hai lần cho L1 và L2 với độ dài $\frac{n}{2}$ do đó thời gian để gọi hai lần đệ quy này là $2T(\frac{n}{2})$. Ngoài ra còn phải tốn thời gian cho việc chia danh sách L thành hai nửa bằng nhau và trộn hai danh sách kết quả (Merge). Người ta xác đinh được thời gian để chia danh sách và Merge là $O(n) = C_2 n$. Vậy ta có phương trình đệ quy như sau:

$$T(n) = \begin{cases} C_1 & \text{nêu } n = 1 \\ 2T(\frac{n}{2}) + C_2\, n & \text{nêu } n > 1 \end{cases}$$

1.6.2 Giải phương trình đệ quy

Có ba phương pháp giải phương trình đệ quy:

1.- Phương pháp truy hồi

2.- Phương pháp đoán nghiệm.

3.- Lời giải tổng quát của một lớp các phương trình đệ quy.

1.6.2.1 Phương pháp truy hồi

Dùng đệ quy để thay thế bất kỳ T(m) với m < n vào phía phải của phương trình cho đến khi tất cả T(m) với m > 1 được thay thế bởi biểu thức của các T(1) hoặc T(0). Vì T(1) và T(0) luôn là hằng số nên chúng ta có công thức của T(n) chứa các số hạng chỉ liên quan đến n và các hằng số. Từ công thức đó ta suy ra T(n).

Ví dụ 1-12: Giải phương trình $T(n) = \begin{cases} C_1 & \text{nêu } n = 0 \\ T(n-1) + C_2 & \text{nêu } n > 0 \end{cases}$

Ta có $T(n) = T(n-1) + C_2$

$T(n) = [T(n-2) + C_2] + C_2 = T(n-2) + 2C_2$

$T(n) = [T(n-3) + C_2] + 2C_2 = T(n-3) + 3C_2$

……

$T(n) = T(n-i) + iC_2$

Quá trình trên kết thúc khi n - i = 0 hay i = n. Khi đó ta có

$T(n) = T(0) + nC_2 = C_1 + n\,C_2 = O(n)$

Ví dụ 1-13: Giải phương trình $T(n) = \begin{cases} C_1 & \text{nêu } n = 1 \\ 2T(\frac{n}{2}) + C_2 n & \text{nêu } n > 1 \end{cases}$

Ta có $T(n) = 2T(\frac{n}{2}) + 2C_2 n$

$T(n) = 2[\,2T(\frac{n}{4}) + C_2\frac{n}{2}\,] + C_2 n = 4T(\frac{n}{4}) + 2C_2 n$

$T(n) = 4[\,2T(\frac{n}{8}) + C_2\frac{n}{4}\,] + 2C_2 n = 8T(\frac{n}{8}) + 3C_2 n$

.....

$T(n) = 2^i T(\frac{n}{2^i}) + iC_2 n$

Quá trình suy rộng sẽ kết thúc khi $\frac{n}{2^i} = 1$ hay $2^i = n$ và do đó i = logn. Khi đó ta có:

$T(n) = nT(1) + \log n C_2 n = C_1 n + C_2 n\log n = O(n\log n)$.

1.6.2.2 Phương pháp đoán nghiệm

Ta đoán một nghiệm f(n) và dùng chứng minh quy nạp để chứng tỏ rằng $T(n) \le f(n)$ với mọi n.

Thông thường f(n) là một trong các hàm quen thuộc như logn, n, nlogn, n^2, n^3, 2^n, n!, n^n.

Đôi khi chúng ta chỉ đoán dạng của f(n) trong đó có một vài tham số chưa xác định (chẳng hạn $f(n) = an^2$ với a chưa xác định) và trong quá trình chứng minh quy nạp ta sẽ suy diễn ra giá trị thích hợp của các tham số.

Ví dụ 1-12: Giải phương trình đệ quy $T(n) = \begin{cases} C_1 & \text{nêu } n = 1 \\ 2T(\frac{n}{2}) + C_2 n & \text{nêu } n > 1 \end{cases}$

Giả sử chúng ta đoán f(n) = anlogn. Với n = 1 ta thấy rằng cách đoán như vậy không được bởi vì anlogn có giá trị 0 không phụ thuộc vào giá trị của a. Vì thế ta thử tiếp theo f(n) = anlogn + b.

Với n = 1 ta có, $T(1) = C_1$ và f(1) = b, muốn $T(1) \le f(1)$ thì $b \ge C_1$ (*)

Giả sử rằng $T(k) \le f(k)$, tức là $T(k) \le aklogk + b$ với mọi k < n (giả thiết quy nạp). Ta phải chứng minh $T(n) \le anlogn + b$ với mọi n.

Giả sử $n \ge 2$, từ phương trình đã cho ta có $T(n) = 2T(\frac{n}{2}) + C_2 n$

Áp dụng giả thiết quy nạp với $k = \frac{n}{2} < n$ ta có:

$T(n) = 2T(\frac{n}{2}) + C_2 n \le 2[a\frac{n}{2}\log\frac{n}{2} + b] + C_2 n$

$T(n) \leq (anlogn - an + 2b) + C_2n$

$T(n) \leq (anlogn + b) + [b + (C_2 - a)n]$. Nếu lấy $a \geq C_2 + b$ (**) ta được

$T(n) \leq (anlogn + b) + [b +(C_2 - C_2 - b)n]$

$T(n) \leq (anlogn + b) + (1-n) b$

$T(n) \leq anlogn + b = f(n)$. (do $b>0$ và $1-n<0$)

Nếu ta lấy a và b sao cho cả (*) và (**) đều thoả mãn thì $T(n) \leq an\ logn + b$ với mọi n.

Ta phải giải hệ $\begin{cases} b \geq C_1 \\ a \geq C_2 + b \end{cases}$ Để đơn giản, ta giải hệ $\begin{cases} b = C_1 \\ a = C_2 + b \end{cases}$

Dễ dàng ta có $b = C_1$ và $a = C_1 + C_2$ ta được $T(n) \leq (C_1 + C_2)nlogn + C_1$ với mọi n.

Hay nói cách khác T(n) là O(nlogn).

1.6.2.3 Lời giải tổng quát cho một lớp các phương trình đệ quy

Khi thiết kế các giải thuật, người ta thường vận dụng phương pháp chia để trị mà ta sẽ bàn chi tiết hơn trong chương 3. Ở đây chỉ trình bày tóm tắt phương pháp như sau:

Để giải một bài toán kích thước n, ta chia bài toán đã cho thành a bài toán con, mỗi bài toán con có kích thước $\frac{n}{b}$. Giải các bài toán con này và tổng hợp kết quả lại để được kết quả của bài toán đã cho. Với các bài toán con chúng ta cũng sẽ áp dụng phương pháp đó để tiếp tục chia nhỏ ra nữa cho đến các bài toán con kích thước 1. Kĩ thuật này sẽ dẫn chúng ta đến một giải thuật đệ quy.

Giả thiết rằng mỗi bài toán con kích thước 1 lấy một đơn vị thời gian và thời gian để chia bài toán kích thước n thành các bài toán con kích thước $\frac{n}{b}$ và tổng hợp kết quả từ các bài toán con để được lời giải của bài toán ban đầu là d(n). (Chẳng hạn đối với ví dụ MergeSort, chúng ta có $a = b = 2$, và $d(n) = C_2n$. Xem C_1 là một đơn vị).

Tất cả các giải thuật đệ quy như trên đều có thể thành lập một phương trinh đệ quy tổng quát, chung cho lớp các bài toán ấy.

Nếu gọi T(n) là thời gian để giải bài toán kích thước n thì $T(\frac{n}{b})$ là thời gian để giải bài toán con kích thước $\frac{n}{b}$. Khi n = 1 theo giả thiết trên thì thời gian giải bài toán kích thước 1 là 1 đơn vị, tức là T(1) = 1. Khi n lớn hơn 1, ta phải giải đệ quy a bài toán con kích thước $\frac{n}{b}$, mỗi bài toán con tốn $T(\frac{n}{b})$ nên thời gian cho a lời giải đệ quy này là $aT(\frac{n}{b})$. Ngoài ra ta còn phải tốn thời gian để phân chia bài toán và tổng hợp các kết quả, thời gian này theo giả thiết trên là d(n). Vậy ta có phương trình đệ quy:

$$T(n) = \begin{cases} 1 & \text{neu } n = 1 \\ aT(\frac{n}{b}) + d(n) & \text{neu } n > 1 \end{cases} \qquad (I.1)$$

Ta sử dụng phương pháp truy hồi để giải phương trình này. Khi n > 1 ta có

$$T(n) = aT(\frac{n}{b}) + d(n)$$

$$T(n) = a[aT(\frac{n}{b^2}) + d(\frac{n}{b})] + d(n) = a^2T(\frac{n}{b^2}) + ad(\frac{n}{b}) + d(n)$$

$$T(n) = a^2[a\,T(\frac{n}{b^3}) + d(\frac{n}{b^2})] + ad(\frac{n}{b}) + d(n) = a^3T(\frac{n}{b^3}) + a^2d(\frac{n}{b^2}) + ad(\frac{n}{b}) + d(n)$$

$$= \quad .....$$

$$= a^iT(\frac{n}{b^i}) + \sum_{j=0}^{i-1} a^j d(\frac{a}{b^j})$$

Giả sử $n = b^k$, quá trình suy rộng trên sẽ kết thúc khi i = k.

Khi đó ta được $T(\frac{n}{b^k}) = T(1) = 1$. Thay vào trên ta có:

$$T(n) = a^k + \sum_{j=0}^{k-1} a^j d(b^{k-j}) \qquad (I.2)$$

1.6.2.3.1 Hàm tiến triển, nghiệm thuần nhất và nghiệm riêng

Trong phương trình đệ quy (I.1) hàm thời gian d(n) được gọi là **hàm tiến triển** (driving function)

Trong công thức (I.2), $a^k = n^{\log_b a}$ được gọi là **nghiệm thuần nhất** (homogeneous solutions).

Nghiệm thuần nhất là nghiệm chính xác khi d(n) = 0 với mọi n. Nói một cách khác, nghiệm thuần nhất biểu diễn thời gian để giải tất cả các bài toán con.

Trong công thức (I.2), $\sum_{j=0}^{k-1} a^j d(b^{k-j})$ được gọi là **nghiệm riêng** (particular solutions).

Nghiệm riêng biểu diễn thời gian phải tốn để tạo ra các bài toán con và tổng hợp các kết quả của chúng. Nhìn vào công thức ta thấy nghiệm riêng phụ thuộc vào hàm tiến triển, số lượng và kích thước các bài toán con.

Khi tìm nghiệm của phương trình (I.1), chúng ta phải tìm nghiệm riêng và so sánh với nghiệm thuần nhất. Nếu nghiệm nào lớn hơn, ta lấy nghiệm đó làm nghiệm của phương trình (I.1).

Việc xác định nghiệm riêng không đơn giản chút nào, tuy vậy, chúng ta cũng tìm được một lớp các hàm tiến triển có thể dễ dàng xác định nghiệm riêng.

1.6.2.3.2 Hàm nhân

Một hàm f(n) được gọi là hàm nhân (multiplicative function) nếu f(m.n) = f(m).f(n) với mọi số nguyên dương m và n.

Ví dụ 1-13: Hàm $f(n) = n^k$ là một hàm nhân, vì $f(m.n) = (m.n)^k = m^k.n^k = f(m)\ f(n)$

Tính nghiệm của phương trình tổng quát trong trường hợp d(n) là hàm nhân:

Nếu d(n) trong (I.1) là một hàm nhân thì theo tính chất của hàm nhân ta có $d(b^{k-j}) = [d(b)]^{k-j}$ và nghiệm riêng của (I.2) là

$$\sum_{j=0}^{k-1} a^j d\left(b^{k-j}\right) = \sum_{j=0}^{k-1} a^j [d(b)]^{k-j} = [d(b)]^k \sum_{j=0}^{k-1} \left[\frac{a}{d(b)}\right]^j = [d(b)]^k \frac{\left[\frac{a}{d(b)}\right]^k - 1}{\frac{a}{d(b)} - 1}$$

Hay nghiệm riêng $= \dfrac{a^k - [d(b)]^k}{\frac{a}{d(b)} - 1}$ (I.3)

Xét ba trường hợp sau:

1.- **Trường hợp 1**: a > d(b) thì trong công thức (I.3) ta có $a^k > [d(b)]^k$, theo quy tắc lấy độ phức tạp ta có nghiệm riêng là $O(a^k) = O(n^{\log_b a})$. Như vậy nghiệm riêng và nghiệm thuần nhất bằng nhau do đó **$T(n)$ là $O(n^{\log_b a})$**.

Trong trương hợp này ta thấy thời gian thực hiện chỉ phụ thuộc vào a, b mà không phụ thuộc vào hàm tiến triển d(n). Vì vậy **để cải tiến giải thuật ta cần giảm a hoặc tăng b.**

2.- **Trường hợp 2**: a < d(b) thì trong công thức (I.3) ta có $[d(b)]^k > a^k$, theo quy tắc lấy độ phức tạp ta cónghiệm riêng là $O([d(b)]^k) = O(n^{\log_b d(b)})$. Trong trường hợp này nghiệm riêng lớn hơn nghiệm thuần nhất nên **$T(n)$ là $O(n^{\log_b d(b)})$**.

Để cải tiến giải thuật chúng ta cần giảm d(b) hoặc tăng b.

Trường hợp đặc biệt quan trọng khi d(n) = n . Khi đó d(b) = b và $\log_b b = 1$. Vì thế nghiệm riêng là O(n) và do vậy **$T(n)$ là $O(n)$**.

3.- **Trường hợp 3**: a = d(b) thì công thức (I.3) không xác đinh nên ta phải tính trực tiếp nghiệm riêng:

$$\text{Nghiệm riêng} = [d(b)]^k \sum_{j=0}^{k-1} \left[\frac{a}{d(b)}\right]^j = a^k \sum_{j=0}^{k-1} 1 = a^k k \text{ (do } a = d(b))$$

Do $n = b^k$ nên $k = \log_b n$ và $a^k = n^{\log_b a}$. Vậy nghiệm riêng là $n^{\log_b a} \log_b n$ và nghiệm này lớn gấp $\log_b n$ lần nghiệm thuần nhất. Do đó **$T(n)$ là $O(n^{\log_b a} \log_b n)$**.

Chú ý khi giải một phương trình đệ quy cụ thể, ta phải xem phương trình đó có thuộc dạng phương trình tổng quát hay không. Nếu có thì phải xét xem hàm tiến triển có phải là hàm nhân không. Nếu có thì ta xác định a, d(b) và dựa vào sự so sánh giữa a và d(b) mà vận dụng một trong ba trường hợp nói trên.

Ví dụ 1-14: Giải các phương trình đệ quy sau với T(1) = 1 và

1/- $T(n) = 4T(\frac{n}{2}) + n$

2/- $T(n) = 4T(\frac{n}{2}) + n^2$

3/- $T(n) = 4T(\frac{n}{2}) + n^3$

Các phương trình đã cho đều có dạng phương trình tổng quát, các hàm tiến triển d(n) đều là các hàm nhân và a = 4, b = 2.

Với phương trình thứ nhất, ta có d(n) = n => d(b) = b = 2 < a, áp dụng trường hợp 1 ta có $T(n) = O(n^{\log_b a}) = O(n^{\log 4}) = O(n^2)$.

Với phương trình thứ hai, $d(n) = n^2$ => $d(b) = b^2 = 4 = a$, áp dụng trường hợp 3 ta có $T(n) = O(n^{\log_b a}\log_b n) = O(n^{\log 4}\log n) = O(n^2\log n)$.

Với phương trình thứ 3, ta có $d(n) = n^3$ => $d(b) = b^3 = 8 > a$, áp dụng trường hợp 2, ta có $T(n) = O(n^{\log_b d(b)}) = O(n^{\log 8}) = O(n^3)$.

1.6.2.3.3 Các hàm tiến triển khác

Trong trường hợp hàm tiến triển không phải là một hàm nhân thì chúng ta không thể áp dụng các công thức ứng với ba trường hợp nói trên mà chúng ta phải tính trực tiếp nghiệm riêng, sau đó so sánh với nghiệm thuần nhất để lấy nghiệm lớn nhất trong hai nghiệm đó làm nghiệm của phương trình.

Ví dụ 1-15: Giải phương trình đệ quy sau :

T(1) = 1

$T(n) = 2T(\frac{n}{2}) + n\log n$

Phương trình đã cho thuộc dạng phương trình tổng quát nhưng d(n) = nlogn không phải là một hàm nhân.

Ta có nghiệm thuần nhất = $n^{\log_b a} = n^{\log 2} = n$

Do d(n) = nlogn không phải là hàm nhân nên ta phải tính nghiệm riêng bằng cách xét trực tiếp

$$\text{Nghiệm riêng} = \sum_{j=0}^{k-1} a^j d(b^{k-j}) = \sum_{j=0}^{k-1} 2^j 2^{k-j}\log 2^{k-j} = 2k\sum_{j=0}^{k-1}(k-j) = 2^k\frac{k(k+1)}{2} = O(2^k k^2)$$

Theo giả thiết trong phương trình tổng quát thì $n = b^k$ nên $k = \log_b n$, ở đây do b = 2 nên $2^k = n$ và k = logn, chúng ta có nghiệm riêng là $O(n\log^2 n)$, nghiệm này lớn hơn nghiệm thuần nhất do đó $T(n) = O(n\log^2 n)$.