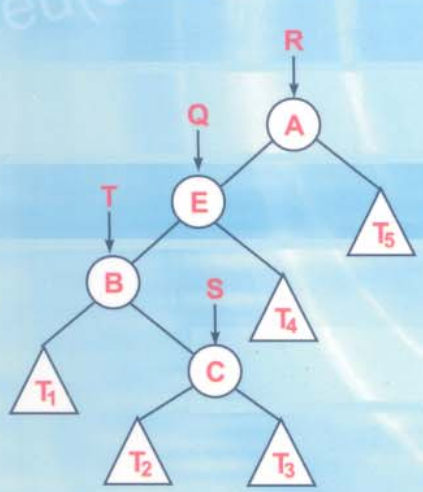
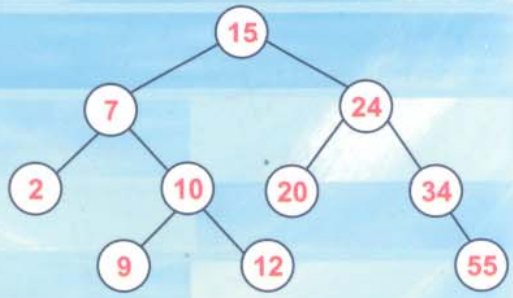
TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP HÀ NỘI
KHOA CÔNG NGHỆ THÔNG TIN
ThS. **An Văn Minh** - ThS. **Trần Hùng Cường**

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT





NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG

TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP HÀ NỘI
KHOA CÔNG NGHỆ THÔNG TIN
ThS. AN VĂN MINH - ThS. TRẦN HÙNG CƯỜNG

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG
Hà Nội - 2009

DANH SÁCH HỘI ĐỒNG THẨM ĐỊNH

PGS. TS. Đoàn Văn Ban:	Chủ tịch
ThS. Ngô Đức Vĩnh:	Thư ký
PGS. TS. Ngô Quốc Tạo:	Ủy viên
ThS. Lê Anh Thắng:	Ủy viên
ThS. Nguyễn Mạnh Cường:	Ủy viên

Mã số: GD 14 HM 09

LỜI NÓI ĐẦU

Công nghệ thông tin ngày càng được ứng dụng rộng rãi và hiệu quả trong mọi lĩnh vực khoa học tự nhiên và xã hội. Để thực hiện một đề án tin học là chuyển bài toán thực tế thành bài toán có thể giải quyết được trên máy tính thì vấn đề thiết kế, lựa chọn cấu trúc dữ liệu và giải thuật là một giai đoạn quan trọng trong qui trình thiết kế và xây dựng phần mềm.

Nhằm giới thiệu những kiến thức cơ bản về cấu trúc dữ liệu và giải thuật, Khoa Công nghệ Thông tin - Trường Đại học Công nghiệp Hà Nội đã phối hợp với Nhà xuất bản Thông tin và Truyền thông xuất bản cuốn sách ***"Cấu trúc dữ liệu và giải thuật"***. Cuốn sách do ThS. An Văn Minh và ThS. Trần Hùng Cường biên soạn dựa theo đề cương chi tiết qui định của Trường Đại học Công nghiệp Hà Nội và đã được Hội đồng khoa học của Trường thẩm định. Đây là môn học cơ sở cùng tên trong chương trình đào tạo kỹ sư công nghệ thông tin. Bài giảng gồm 5 chương với nội dung như sau:

Chương 1: Tổng quan về cấu trúc dữ liệu và giải thuật, bao gồm các khái niệm về cấu trúc dữ liệu và giải thuật, mối quan hệ giữa chúng, vấn đề thiết kế cấu trúc dữ liệu, thiết kế và phân tích giải thuật, đánh giá độ phức tạp của giải thuật.

Chương 2: Đệ quy và giải thuật đệ quy, một phương pháp thiết kế giải thuật khá quan trọng, nhất là với các giải thuật biểu diễn các thao tác xử lý cấu trúc dữ liệu dạng cây.

Chương 3: Sắp xếp và tìm kiếm, tập trung vào vấn đề mô tả, thiết kế và đánh giá các giải thuật sắp xếp và tìm kiếm thông dụng, cũng như vấn đề cài đặt các giải thuật này trong bài toán ứng dụng.

Chương 4: Danh sách tuyến tính, một loại cấu trúc dữ liệu rất phổ biến trong các bài toán tin học. Trong chương này trình bày các phương pháp lưu trữ danh sách và các thao tác xử lý tương ứng với mỗi loại danh sách.

Chương 5: Cây, một dạng cấu trúc dữ liệu phi tuyến tính, chương này chủ yếu nói về cây nhị phân và các ứng dụng của chúng.

Bài tập sau mỗi chương đã được chọn lọc ở mức độ phù hợp với sinh viên, qua đó giúp cho sinh viên hiểu sâu sắc thêm về bài giảng, củng cố thêm về kỹ thuật cài đặt chương trình và nắm bắt được một số kiến thức không được trực tiếp giới thiệu trong bài giảng.

Để học tốt môn học này đòi hỏi sinh viên phải thành thạo ít nhất một ngôn ngữ lập trình cơ bản như Pascal, C hay C++, v.v..., thành thạo các kỹ thuật lập trình như: cấu trúc rẽ nhánh, cấu trúc lặp, kỹ thuật lập trình đơn thể (sử dụng hàm, thủ tục).

Mặc dù nhóm tác giả có nhiều cố gắng trong công tác biên soạn song sẽ khó tránh khỏi thiếu sót, chúng tôi rất mong nhận được ý kiến đóng góp của các bạn đồng nghiệp và bạn đọc để lần xuất bản sau được hoàn thiện hơn.

Mọi ý kiến đóng góp xin gửi về Khoa Công nghệ Thông tin – Trường Đại học Công nghiệp Hà Nội.

Xin trân trọng giới thiệu./.

Hà Nội, tháng 9 năm 2009

TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP HÀ NỘI

TỔNG QUAN VỀ CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

1. VAI TRÒ CỦA CẤU TRÚC DỮ LIỆU

Thực hiện một đề án tin học là chuyển bài toán thực tế thành bài toán có thể giải quyết trên máy tính. Mỗi bài toán bất kỳ đều bao gồm các đối tượng dữ liệu và các yêu cầu xử lý trên các đối tượng đó. Vì thế, để xây dựng một mô hình tin học phản ánh được bài toán thực tế cần chú trọng đến hai vấn đề:

Tổ chức biểu diễn các đối tượng thực tế

Các thành phần dữ liệu thực tế rất đa dạng, phong phú và thường chứa đựng những quan hệ nào đó với nhau. Do đó, trong mô hình tin học của bài toán, cần phải biểu diễn chúng một cách thích hợp nhất, để vừa có thể phản ánh chính xác các dữ liệu thực tế này, vừa có thể dễ dàng dùng máy tính để xử lý. Công việc này gọi là xây dựng cấu trúc dữ liệu cho bài toán.

Xây dựng các thao tác xử lý dữ liệu

Từ những yêu cầu xử lý của bài toán, cần tìm ra các giải pháp tương ứng để giải quyết, mỗi giải pháp cần phải xác định trình tự các thao tác máy tính phải thi hành để cho ra kết quả mong muốn. Đây là bước xây dựng giải thuật cho bài toán. Có thể sử dụng các giải thuật có sẵn, hoặc tự xây dựng.

Tuy nhiên, khi giải quyết bài toán, ta thường có khuynh hướng chỉ chú trọng đến việc xây dựng các giải thuật mà không chú trọng tới tầm quan trọng của việc tổ chức dữ liệu trong bài toán. Giải thuật phản ánh các phép xử lý, còn đối tượng xử lý của giải thuật lại là dữ liệu.

Chính dữ liệu chứa đựng các thông tin cần thiết để thực hiện giải thuật. Để xác định được giải thuật phù hợp ta cần phải biết nó tác động đến loại dữ liệu nào và khi lựa chọn cấu trúc dữ liệu cũng cần phải hiểu rõ những thao tác nào tác động lên dữ liệu đó. Như vậy trong một bài toán, cấu trúc dữ liệu và giải thuật có mối quan hệ chặt chẽ với nhau, được thể hiện qua "công thức":

Cấu trúc dữ liệu + Giải thuật = Chương trình

Với một cấu trúc dữ liệu đã chọn, sẽ có những giải thuật tương ứng, phù hợp. Khi cấu trúc dữ liệu thay đổi, thường giải thuật cũng phải thay đổi theo để tránh việc xử lý "gượng ép", thiếu tự nhiên trên một cấu trúc không phù hợp. Hơn nữa, một cấu trúc dữ liệu tốt sẽ giúp giải thuật xử lý trên đó có thể phát huy tác dụng tốt hơn, vừa nhanh vừa tiết kiệm, giải thuật cũng đơn giản và dễ hiểu hơn.

Ví dụ 1: Một chương trình quản lý điểm thi của sinh viên cần lưu các điểm số của 3 sinh viên. Do mỗi sinh viên có 4 điểm số tương ứng với 4 môn học khác nhau nên dữ liệu có dạng như sau:

Sinh viên	Môn 1	Môn 2	Môn 3	Môn 4
SV1	7	9	7	5
SV2	5	4	2	7
SV3	8	9	6	7

Xét thao tác xử lý là xuất điểm số các môn của từng sinh viên.

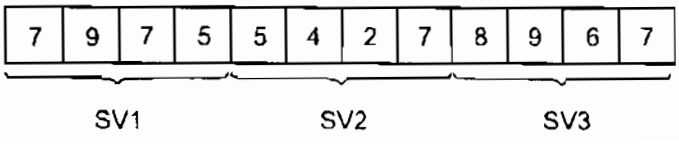
Giả sử có các phương án tổ chức lưu trữ như sau:

Phương án 1: *Sử dụng mảng một chiều:*

Có tất cả 3(SV) * 4(Môn) = 12 điểm số cần lưu trữ, do đó ta khai báo mảng như sau:

```
int a[12];
```

Khi đó mảng a các phần tử sẽ được lưu trữ như sau:



Và truy xuất điểm số môn j của sinh viên i là phần tử tại dòng i cột j trong bảng. Để truy xuất đến phần tử này ta phải sử dụng công thức xác định chỉ số tương ứng trong mảng a:

Bảng điểm (dòng i, cột j) ⇒ a[(i -1)*số cột + j]

Ngược lại, với một phần tử bất kỳ trong mảng, muốn biết đó là điểm số của sinh viên nào, môn gì, phải dùng công thức xác định sau:

a[i] ⇒ bảng điểm (dòng(i/cột) + 1), cột (i % số cột))

Với phương án này, giải thuật xử lý được viết như sau:

```
void xuat(int a[ ])
{ int i, mon, so_mon;
  so_mon = 4;
  for (i = 0; i < 12; i++)
  {       sv = i/so_mon;
          mon = i % so_mon;
          cout<<"\nĐiểm môn:"<<mon;
          cout<<" của sinh viên "<<sv;
          cout<<" là:"<< a[i]<<endl;
  }
}
```

Phương án 2: *Sử dụng mảng hai chiều*

Khai báo mảng hai chiều a có kích thước 3 dòng * 4 cột như sau:

int *a[3][4];*

	Cột 1	**Cột 2**	**Cột 3**	**Cột 4**
Dòng 1	a[1][1] = 7	a[1][2] = 9	a[1][3] = 7	a[1][4] = 5
Dòng 2	a[2][1] = 5	a[2][2] = 4	a[2][3] = 2	a[2][4] = 7
Dòng 3	a[3][1] = 8	a[3][2] = 9	a[3][3] = 6	a[3][4] = 7

Và truy xuất điểm số môn j của sinh viên i là phần tử tại dòng i cột j trong bảng cũng chính là phần tử ở dòng i cột j trong mảng.

Bảngđiểm (dòng i, cột j) ⇒ a[i,j]

Với phương án này, giải thuật xử lý được viết như sau:

```
void xuat( int a[3][4])
{
  int i, j, so_sv, so_mon;
  so_mon = 4; so_sv = 3;
  for (i = 0; i < so_sv; i++)
        {
              for (j = 0; j < so_mon; j++)
              {cout<<"\nĐiểm môn:"<<mon;
               cout<<"của sinh viên"<<sv;
               cout<<"là:"<< a[i][j]<<endl;
              }
        }
}
```

Nhận xét:

Có thể thấy rõ phương án 2 cung cấp một cấu trúc dữ liệu lưu trữ phù hợp với dữ liệu thực tế hơn phương án 1, và do vậy giải thuật xử lý trên cấu trúc dữ liệu của phương án 2 cũng đơn giản hơn, tự nhiên hơn.

2. CÁC TIÊU CHUẨN ĐÁNH GIÁ CẤU TRÚC DỮ LIỆU

Do tầm quan trọng của cấu trúc dữ liệu đã được trình bày trong phần trên, nên nhất thiết phải chú trọng đến việc lựa chọn một phương án tổ chức dữ liệu thích hợp cho bài toán cụ thể. Một cấu trúc dữ liệu tốt phải thoả mãn các tiêu chuẩn sau:

**** Phản ánh đúng thực tế***: Đây là tiêu chuẩn quan trọng nhất, quyết định tính đúng đắn của toàn bộ bài toán. Cần xem xét kỹ lưỡng cũng như dự trù các trạng thái biến đổi của dữ liệu trong chu trình

sống để có thể lựa chọn cấu trúc dữ liệu lưu trữ thể hiện chính xác đối tượng thực tế.

Ví dụ: Một số tình huống sau chọn cấu trúc lưu trữ sai

- Chọn biến số nguyên "***int***" để lưu trữ tiền thưởng bán hàng (được tính theo công thức tiền thưởng bán hàng = trị giá hàng *5%), do vậy khi làm tròn mọi giá trị tiền thưởng sẽ gây thiệt hại cho nhân viên bán hàng. Trường hợp này phải sử dụng biến số thực để phản ánh đúng kết quả của công thức tính thực tế.

- Trong trường Trung học, mỗi lớp có thể nhận tối đa 25 học sinh. Lớp hiện có 20 học sinh, mỗi tháng, mỗi học sinh đóng học phí 15.000 đồng. Chọn một biến số nguyên (khả năng trong phạm vi (-32768 ÷ 32767)) để lưu trữ tổng số học phí của lớp học trong tháng, nếu xảy ra trường hợp có thêm 5 học sinh nữa vào lớp thì giá trị tổng học phí thu được là 375000 đồng, vượt khả năng lưu trữ của biến đã chọn, gây ra tình trạng tràn số và sai lệch.

* ***Phù hợp với các giải thuật xử lý trên đó***: Tiêu chuẩn này giúp tăng hiệu quả khi giải quyết bài toán, việc phát triển các giải thuật đơn giản, tự nhiên hơn và chương trình đạt hiệu quả cao hơn về tốc độ xử lý.

* ***Tiết kiệm tài nguyên hệ thống***: Cấu trúc dữ liệu chỉ nên sử dụng tài nguyên vừa đủ để đảm nhiệm được chức năng của nó. Thông thường có hai loại tài nguyên cần lưu ý nhất là bộ vi xử lý (CPU) và bộ nhớ. Tiêu chuẩn này nên cân nhắc tuỳ vào tình huống cụ thể khi thực hiện bài toán. Nếu tổ chức sử dụng bài toán cần có những xử lý nhanh thì khi chọn cấu trúc dữ liệu yếu tố tiết kiệm thời gian xử lý phải được ưu tiên hơn tiêu chuẩn sử dụng tối đa bộ nhớ, và ngược lại.

Ví dụ: Một số tình huống chọn cấu trúc lưu trữ lãng phí

- Sử dụng biến "***int***" (2 byte) để lưu trữ một giá trị cho biết tháng hiện hành. Trong tình huống này ta chỉ cần sử dụng biến kiểu "***char***" là đủ.

- Để lưu trữ danh sách học viên trong một lớp, sử dụng mảng 60 phần tử (giới hạn số học viên trong lớp tối đa là 60). Nếu số lượng học

viên thật sự ít hơn 60, thì gây lãng phí bộ nhớ. Hơn nữa, số học viên có thể thay đổi theo từng kỳ, từng năm. Trong trường hợp này ta cần có một cấu trúc dữ liệu linh động hơn mảng, chẳng hạn danh sách móc nối.

3. CÁC CẤU TRÚC DỮ LIỆU CƠ SỞ

Máy tính thực sự chỉ có thể lưu trữ dữ liệu ở dạng nhị phân thô sơ. Nếu muốn phản ánh được dữ liệu thực tế vốn rất đa dạng và phong phú, cần phải xây dựng những phép ánh xạ, những qui tắc tổ chức phức tạp che lên tầng dữ liệu thô, nhằm đưa ra những khái niệm logic về hình thức lưu trữ khác nhau thường được gọi là *kiểu dữ liệu*. Như đã phân tích ở phần đầu, giữa hình thức lưu trữ và các thao tác xử lý trên đó có quan hệ mật thiết với nhau. Từ đó có thể đưa ra một định nghĩa cho kiểu dữ liệu như sau.

3.1. Định nghĩa kiểu dữ liệu

Kiểu dữ liệu T được xác định bởi bộ <V,O>, với:

- V: tập các giá trị hợp lệ mà đối tượng kiểu T có thể lưu trữ.
- O: tập các thao tác xử lý có thể thi hành trên đối tượng kiểu T.

Ví dụ:

- Kiểu dữ liệu **Ký tự alphabet** = <V_c, O_c> với:

  V_c = {a - z, A - Z}

  O_c = {lấy mã ASCII của ký tự, biến đổi ký tự thường thành ký tự hoa,...}

- Kiểu dữ liệu **Số nguyên** = <V_i, O_i> với:

  V_i = {-32768 ÷32767}

  O_i = {+, -, *, /, %, các phép so sánh, các phép toán logic nhị phân}

Như vậy, muốn sử dụng một kiểu dữ liệu cần nắm vững cả nội dung dữ liệu được phép lưu trữ và các xử lý tác động trên đó.

3.2. Các thuộc tính của một kiểu dữ liệu

Một kiểu dữ liệu bao gồm các thuộc tính sau:

- Tên kiểu dữ liệu
- Miền giá trị
- Kích thước lưu trữ
- Tập các toán tử tác động lên kiểu dữ liệu.

3.3. Các kiểu dữ liệu cơ bản

Thông thường trong một hệ kiểu của ngôn ngữ lập trình sẽ có một số kiểu dữ liệu được gọi là *kiểu dữ liệu đơn* hay *kiểu dữ liệu nguyên tử* (atomic).

Thông thường, các kiểu dữ liệu cơ bản bao gồm:

- Kiểu có thứ tự rời rạc: số nguyên, ký tự, logic, liệt kê, miền con
- Kiểu không rời rạc: số thực.

Tuỳ từng ngôn ngữ lập trình, các kiểu dữ liệu định nghĩa sẵn này có thể khác nhau đôi chút. Chẳng hạn, với ngôn ngữ lập trình C/C++, các kiểu dữ liệu này chỉ gồm số nguyên, số thực, ký tự. Và trong ngôn ngữ lập trình C/C++, kiểu ký tự thực chất cũng là kiểu số nguyên về mặt lưu trữ, chỉ khác về cách sử dụng. Ngoài ra, giá trị logic đúng (TRUE) và giá trị logic sai (FALSE) được biểu diễn trong ngôn ngữ lập trình C/C++ như là các giá trị nguyên khác 0 và bằng 0. Trong khi đó ngôn ngữ lập trình Pascal định nghĩa tất cả các kiểu dữ liệu đã liệt kê ở trên và phân biệt chúng một cách chặt chẽ.

Các kiểu dữ liệu của C/C++ được cho trong bảng sau:

Tên kiểu	Phạm vi	Kích thước	Giải thích
int	-32768 ÷ +32767	2 byte	Số nguyên có dấu
char	-128 ÷ +127	1 byte	
long	-2147483648 ÷ +2147483647	4 byte	
unsigned char	0 ÷ 255	1 byte	Số nguyên không dấu
unsigned int	0 ÷ 65535	2 byte	
unsigned long	0 ÷ 4294967259	4 byte	

Tên kiểu	Phạm vi	Kích thước	Giải thích
float	$1.2*10^{-38} \div 3.4*10^{38}$	4 byte	Số thực (dấu chấm động)
double	$2.2*10^{-308} \div 1.8*10^{308}$	8 byte	
long double	$3.5*10^{-3942} \div 3.4*10^{4932}$	10 byte	

3.4. Các kiểu dữ liệu có cấu trúc

Khi giải quyết các bài toán phức tạp, nếu chỉ sử dụng các dữ liệu các dữ liệu đơn là không đủ, ta phải cần đến các *cấu trúc dữ liệu*. Một cấu trúc dữ liệu bao gồm một tập hợp các *dữ liệu nguyên tử*, các thành phần này kết hợp với nhau theo một phương thức được qui định bởi ngôn ngữ lập trình. Đa số các ngôn ngữ lập trình đều cài đặt sẵn một số kiểu có cấu trúc cơ bản như mảng, chuỗi, tập tin, cấu trúc, v.v... và cung cấp cơ chế cho lập trình viên tự định nghĩa kiểu dữ liệu mới.

3.4.1. Mảng một chiều

Trong ngôn ngữ lập trình C/C++ và trong nhiều ngôn ngữ thông dụng khác có một cách đơn giản nhất để tổ chức lưu trữ các đối tượng trong một tập hợp, đó là cách sắp xếp các đối tượng đó thành một dãy. Để lưu trữ dãy đối tượng trong máy tính người ta sử dụng mảng một chiều. Khi đó ta có một cấu trúc dữ liệu được gọi là mảng (array). Như vậy, có thể nói một mảng là một cấu trúc dữ liệu gồm một dãy xác định các dữ liệu thành phần cùng một kiểu (mảng số nguyên, mang số thực, mảng các cấu trúc, v.v...).

Trong C/C++ việc khai báo một mảng là khá đơn giản, cần chỉ ra kiểu dữ liệu của phần tử, tên mảng, kích thước mảng, mẫu như sau:

<Kiểu phần tử> <tên mảng> <[kích thước]>;

Ví dụ: ***int*** A[10]; //khai báo mảng A chứa 10 số nguyên (chỉ số từ 0 đến 9).

3.4.2. Chuỗi

Trong C/C++, chuỗi thực chất là một mảng các ký tự, tuy nhiên có khác là trong chuỗi có chứa ký tự kết thúc '\0'. Việc nhập và hiển

thị chuỗi cũng đơn giản hơn mảng, ta có thể sử dụng các hàm nhập xuất chuẩn trong thư viện "stdio.h" như *scanf(), gets(), printf(), puts()*.

C/C++ cũng định nghĩa một số hàm xử lý chuỗi (thư viện string.h) như: *strcpy(), strlen(), strcmp(), strchr(), strcat(), strstr(),...*

3.4.3. Mảng nhiều chiều

Mảng nhiều chiều được sử dụng nhiều nhất là mảng 2 chiều (mảng của mảng), có thể hình dung mảng 2 chiều giống như một bảng gồm các dòng và các cột, chẳng hạn, bảng ghi nhiệt độ trung bình trong 5 năm ở năm thành phố.

	2003	2004	2005	2006	2007
Hà Nội	27	27,5	28,5	30	27
TP. Hồ Chí Minh	32,5	32	30,5	31	30
Huế	30	31	32	28	29
Hải Phòng	27,5	26,5	26	27,5	27
Hạ Long	25	27	26	28	27

Cấu trúc khai báo của mảng nhiều chiều được viết như sau:

<Kiểu phần tử> <tên mảng ><[kích thước chiều 1]...[kích thước chiều n]>;

Sau tên mảng, mỗi cặp ngoặc vuông [ ] được tính là một chiều. Chữ số ghi trong cặp ngoặc [ ] là số phần tử của chiều đó.

Ví dụ: ***float*** temp [5][5]; // mảng 2 chiều temp, kích thước 5x5, mảng các số thực.

3.4.4. Cấu trúc

Cấu trúc là tập hợp các mẫu dữ liệu khác nhau của một đối tượng (các mẫu dữ liệu có thể có kiểu khác nhau). Các mẫu dữ liệu đó được gọi là thành phần dữ liệu của cấu trúc. Các cấu trúc có thể được sử dụng để tạo nên các kiểu dữ liệu khác, chẳng hạn như mảng cấu trúc.

Giả sử T_1, T_2,...,T_n là các kiểu đã cho, và F_1, F_2,..., F_n là các tên thành phần. Khi đó ta có thể thành lập kiểu cấu trúc ST với n thành phần dữ liệu, thành phần thứ i có tên là F_i và các giá trị của nó có kiểu T_i với i = 1, 2,..., n.

```
struct ST
{
        T1      F1;
        T2      F2;
        ...
        Tn      Fn;
};
struct ST s1, s2, d[20];
```

trong khai báo này s1, s2 là các biến cấu trúc, d là một mảng cấu trúc.

Để truy nhập vào một thành phần dữ liệu của một cấu trúc ta viết theo mẫu:

<tên biến cấu trúc>.<tên thành phần>

Ví dụ: s1.F1, d[2].Fn

Mỗi giá trị của kiểu cấu trúc ST là một bộ k giá trị (t_1, t_2,..., t_k), trong đó $t_i \in T_i$ (i = 1, 2, ..., k).

Ví dụ: Khai báo cấu trúc lưu trữ phân số gồm tử số và mẫu số là các số nguyên.

```
struct phan_so
{
        int tu_so;
        int mau_so;
};
//khai báo các biến lưu trữ phân số
struct phan_so p1, p2, ps[100];
```

Ở đây, p1, p2 là hai biến cấu trúc lưu trữ phân số, còn ps là mảng lưu trữ một dãy nhiều nhất là 100 phân số, ps được gọi là mảng cấu trúc.

3.4.5. Kiểu con trỏ

Một phương pháp quan trọng nữa để kiến tạo các cấu trúc dữ liệu, đó là sử dụng con trỏ.

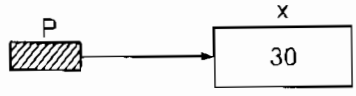
Con trỏ là biến được sử dụng để lưu địa chỉ của một biến khác.

Con trỏ được khai báo theo mẫu:

<kiểu dữ liệu> *<tên con trỏ>;

Ví dụ: ***int*** *P, x;

P=&x; *//P chứa địa chỉ của biến x, hay P trỏ vào x*



Hình 1.1: Biểu diễn con trỏ

Sau này con trỏ được dùng để tạo ra kiểu danh sách móc nối, hoặc cây là các cấu trúc dữ liệu rất quan trọng, ta sẽ có dịp tìm hiểu kỹ hơn cách sử dụng con trỏ trong chương 3.

3.4.6. Kiểu file (tệp tin)

Khác với các kiểu dữ liệu trước đây, số nguyên, số thực, mảng, chuỗi, v.v... dữ liệu được lưu ở bộ nhớ trong. Vì thế, khi chương trình kết thúc, dữ liệu cũng bị xóa. Để khắc phục trường hợp này, dữ liệu cần được lưu ở bộ nhớ ngoài, đó là các tệp tin.

- Theo cách lưu trữ này, khi thao tác cần một tên tệp tin (gồm cả đường dẫn), một con trỏ tệp (FILE * tên_con_trỏ).

- Khi thao tác với tệp tin cũng cần các hàm xử lý, bạn đọc có thể xem trong [3].

3.5. Các phép toán trong các kiểu dữ liệu của C/C++

Như đã nói ở trên, với mỗi kiểu dữ liệu ta chỉ có thể thực hiện một số phép toán nhất định trên các dữ liệu của kiểu. Ta không thể áp dụng một số phép toán trên các dữ liệu thuộc kiểu này cho các dữ liệu thuộc kiểu khác. Các phép toán rất quan trọng, nó là công cụ để thao

tác dữ liệu. Ta có thể chia tập hợp các phép toán trên các kiểu dữ liệu của C/C++ thành hai lớp sau:

3.5.1. Các phép toán truy nhập

Phép toán này dùng để truy nhập đến các thành phần của một đối tượng dữ liệu, chẳng hạn truy nhập đến các phần tử của một mảng, đến các thành phần dữ liệu của cấu trúc.

Ví dụ:

- Giả sử A là một mảng một chiều với n phần tử, (i = 0, 1,..., n-1) khi đó A[i] cho phép ta truy nhập đến thành phần thứ i+1 của mảng.

- Nếu X là một biến cấu trúc thì việc truy nhập đến trường F của nó được thực hiện bởi phép toán X.F.

3.5.2. Các phép toán kết hợp dữ liệu

Ngôn ngữ lập trình C/C++ có một tập hợp phong phú các phép toán kết hợp một hoặc nhiều dữ liệu đã cho thành dữ liệu mới. Sau đây là một số nhóm các phép toán chính.

*** Các phép toán số học**: Đó là các phép toán +, - * , / trên tập số thực; các phép toán +, -, * , /, %, trên tập số nguyên.

*** Các phép toán so sánh**: Trên các đối tượng thuộc các kiểu có thứ tự, ta có thể thực hiện các phép toán so sánh = = (bằng), != (khác), < (nhỏ hơn), > (lớn hơn), <= (nhỏ hơn hoặc bằng), >= (lớn hơn hoặc bằng). Cần chú ý rằng, kết quả của các phép toán này là một giá trị logic (true/false).

*** Các phép toán logic**: Đó là các phép toán &&, ||, !, được thực hiện trên hai giá trị *false* và *true*. Trong C/C++ không có kiểu logic, mà false là giá trị bằng không, và true là giá trị khác không.

4. GIẢI THUẬT - PHÂN TÍCH VÀ ĐÁNH GIÁ GIẢI THUẬT

4.1. Giải thuật

Giải thuật (algorithm) là một trong những khái niệm quan trọng nhất trong tin học. Thuật ngữ giải thuật xuất phát từ nhà toán học

A-rập Abu Ja'far Mohammed ibn Musa al Khowarizmi (khoảng năm 825). Tuy nhiên, trong lúc bấy giờ và trong nhiều thế kỷ sau, nó không mang nội dung như ngày nay chúng ta quan niệm. Giải thuật nổi tiếng nhất, có từ thời cổ Hy Lạp là giải thuật Euclid, giải thuật tìm ước chung lớn nhất của hai số nguyên. Có thể mô tả giải thuật này như sau:

**** Giải thuật Euclid***

Vào: m, n nguyên dương

Ra: d, ước số chung lớn nhất của m và n.

Phương pháp

Bước 1: Tìm r, phần dư của phép chia m cho n

Bước 2: Nếu r = 0, thì d ← n (gán giá trị của n cho d) và dừng lại

Ngược lại, thì m ← n, n ← r và quay lại bước 1

4.1.1. Khái niệm

Giải thuật là một dãy hữu hạn các bước, mỗi bước mô tả chính xác các phép toán hoặc hành động cần thực hiện để giải quyết vấn đề đặt ra.

4.1.2. Đặc trưng của giải thuật

Định nghĩa trên, còn chứa đựng nhiều điều chưa rõ ràng. Để hiểu đầy đủ ý nghĩa của giải thuật, chúng ta nêu ra 5 đặc trưng của nó:

* ***Bộ dữ liệu vào:*** Mỗi giải thuật cần có một số lượng (có thể bằng 0) dữ liệu vào (input). Đó là các giá trị cần đưa vào khi giải thuật bắt đầu làm việc. Các dữ liệu này cần được lấy từ các tập hợp giá trị cụ thể nào đó. Chẳng hạn, trong giải thuật Euclid trên, m và n là các dữ liệu vào lấy từ tập các số nguyên dương.

* ***Dữ liệu ra***: Mỗi giải thuật cần có một hoặc nhiều dữ liệu ra. Đó là các giá trị có quan hệ hoàn toàn xác định với các dữ liệu vào và là kết quả của sự thực hiện giải thuật. Trong giải thuật Euclid có một dữ liệu ra, đó là d, khi thực hiện đến bước 2 và phải dừng lại (trường hợp r = 0), giá trị của d là ước số chung lớn nhất của m và n.

* ***Tính xác định***: Mỗi bước của giải thuật cần phải được mô tả một các chính xác, chỉ có một cách hiểu duy nhất. Hiển nhiên đây là một đòi hỏi rất quan trọng. Bởi vì, nếu một bước có thể hiểu theo nhiều cách khác nhau, thì cùng một dữ liệu vào, những người thực hiện giải thuật khác nhau có thể dẫn đến các kết quả khác nhau. Để đảm bảo được tính xác định giải thuật cần phải được mô tả trong các ngôn ngữ lập trình. Trong các ngôn ngữ này, các mệnh đề được tạo thành theo qui tắc, cú pháp nghiêm ngặt và chỉ có một ý nghĩa duy nhất.

* ***Tính khả thi***: Tất cả các phép toán có mặt trong các bước của giải thuật phải đủ đơn giản. Điều này có nghĩa là, người lập trình có thể thực hiện chỉ bằng giấy trắng và bút trong khoảng thời gian hữu hạn. Chẳng hạn, với giải thuật Euclid, ta chỉ cần thực hiện các phép chia số nguyên, các phép gán và phép so sánh để biết được $r = 0$ hay $r \neq 0$.

* ***Tính dừng***: Với mọi bộ dữ liệu vào thoả mãn các điều kiện của dữ liệu vào, giải thuật phải dừng lại sau một số hữu hạn các bước thực hiện. Chẳng hạn, giải thuật Euclid thoả mãn điều kiện này. Bởi vì, khi thực hiện bước 1 thì giá trị của r nhỏ hơn n, nếu $r \neq 0$ thì giá trị của n ở bước 2 là giá trị của r ở bước trước, ta có $n > r = n_1 > r_1 = n_2 > r_2$,... Dãy số nguyên dương giảm dần cần phải kết thúc ở 0, do đó sau một số hữu hạn bước giá trị của r phải bằng 0, giải thuật dừng.

4.2. Biểu diễn giải thuật

Có nhiều phương pháp biểu diễn giải thuật. Có thể biểu diễn giải thuật bằng danh sách các bước, các bước được diễn đạt bằng ngôn ngữ tự nhiên và các ký hiệu toán học. Có thể biểu diễn bằng sơ đồ khối. Tuy nhiên, như đã trình bày, để đảm bảo tính xác định của giải thuật thì nên biểu diễn nó bằng ngôn ngữ lập trình.

4.3. Phân tích giải thuật

Giả sử đối với một bài toán nào đó chúng ta có một số giải thuật để giải. Một câu hỏi đặt ra là, chúng ta cần chọn giải thuật nào trong

số các giải thuật đã có để giải bài toán một cách hiệu quả nhất. Sau đây ta phân tích giải thuật và đánh giá độ phức tạp tính toán của nó.

4.3.1. Tính hiệu quả của giải thuật

Khi giải quyết một vấn đề, chúng ta cần chọn trong số các giải thuật, một giải thuật mà chúng ta cho là tốt nhất. Vậy ta cần lựa chọn giải thuật dựa trên cơ sở nào? Thông thường ta dựa trên hai tiêu chuẩn sau đây:

1. Giải thuật đơn giản, dễ hiểu, dễ cài đặt (để viết chương trình)

2. Giải thuật sử dụng tiết kiệm nhất nguồn tài nguyên của máy tính và đặc biệt, chạy nhanh nhất có thể được.

Khi ta viết một chương trình chỉ để sử dụng một số ít lần và cái giá của thời gian viết chương trình vượt xa cái giá của chạy chương trình thì tiêu chuẩn (1) là quan trọng nhất. Nhưng có trường hợp ta cần viết các chương trình (thủ tục hoặc hàm) để sử dụng nhiều lần, cho nhiều người sử dụng, khi đó giá của thời gian chạy chương trình sẽ vượt xa giá viết nó. Chẳng hạn, các thủ tục sắp xếp, tìm kiếm được sử dụng rất nhiều lần, cho rất nhiều người trong các bài toán khác nhau. Trong trường hợp này ta cần dựa trên tiêu chuẩn (2). Ta sẽ cài đặt giải thuật có thể rất phức tạp, miễn là chương trình nhận được chạy nhanh hơn các giải thuật khác.

Tiêu chuẩn (2) được xem là tính hiệu quả của giải thuật. Tính hiệu quả của giải thuật bao gồm hai nhân tố cơ bản

- Dung lượng nhớ cần thiết để lưu giữ các dữ liệu vào, các kết quả tính toán trung gian và các kết quả của giải thuật.

- Thời gian cần thiết để thực hiện giải thuật (ta gọi là thời gian chạy chương trình, thời gian này không phụ thuộc vào các yếu tố vật lý của máy tính như tốc độ xử lý của máy tính, ngôn ngữ viết chương trình, v.v...).

Chúng ta sẽ chỉ quan tâm đến thời gian thực hiện giải thuật. Vì vậy khi nói đến đánh giá độ phức tạp của giải thuật, có nghĩa là ta nói

đến đánh giá thời gian thực hiện. Một giải thuật có hiệu quả được xem là giải thuật có thời gian chạy ít hơn các giải thuật khác.

4.3.2. Đánh giá thời gian thực hiện của giải thuật

Có hai cách tiếp cận để đánh giá thời gian thực hiện của một giải thuật.

a. Phương pháp thử nghiệm:

Chương trình được viết và cho chạy với các dữ liệu vào khác nhau trên một máy tính nào đó. Thời gian chạy chương trình phụ thuộc vào các yếu tố sau đây:

1. Các dữ liệu vào

2. Chương trình dịch, để chuyển chương trình mã nguồn thành chương trình mã máy.

3. Tốc độ thực hiện các phép toán của máy tính được sử dụng để chạy chương trình.

Vì thời gian chạy chương trình phụ thuộc vào nhiều yếu tố, nên ta không thể biểu diễn chính xác thời gian chạy là bao nhiêu đơn vị thời gian chuẩn, chẳng hạn nó là bao nhiêu giây nếu không nêu các cấu hình hệ máy thực hiện.

b. Phương pháp lý thuyết:

Ta sẽ coi thời gian thực hiện của giải thuật như là một hàm số của kích thước dữ liệu vào. Kích thước của dữ liệu vào là một tham số đặc trưng cho dữ liệu vào, nó có ảnh hưởng quyết định đến thời gian thực hiện chương trình. Đơn vị tính kích thước của dữ liệu vào phụ thuộc vào các giải thuật cụ thể. Chẳng hạn, đối với các giải thuật sắp xếp mảng, thì kích thước của dữ liệu vào là số thành phần (phần tử) của mảng, đối với giải thuật giải hệ n phương trình tuyến tính với n ẩn, ta chọn n là kích thước. Thông thường dữ liệu vào là một số nguyên dương n. Ta sẽ sử dụng hàm số T(n), trong đó n là kích thước dữ liệu vào, để biểu diễn thời gian thực hiện của một giải thuật.

Có thể xác định thời gian thực hiện T(n) là số phép toán sơ cấp cần phải tiến hành khi thực hiện giải thuật. Các phép toán sơ cấp là các phép toán mà thời gian thực hiện bị chặn trên bởi một hằng số chỉ phụ thuộc vào cách cài đặt được sử dụng. Chẳng hạn các phép toán số học +, -, *, /, các phép toán so sánh =, !=,... là các phép toán sơ cấp.

4.3.3. Đánh giá độ phức tạp tính toán của giải thuật

Khi đánh giá thời gian thực hiện bằng phương pháp toán học, chúng ta sẽ bỏ qua yếu tố phụ thuộc vào cách cài đặt, chỉ tập trung vào xác định độ lớn của thời gian thực hiện T(n). Ký hiệu toán học O (đọc là ô lớn) được sử dụng để mô tả độ lớn của hàm T(n).

Giả sử n là số nguyên không âm, T(n) và f(n) là các hàm thực không âm. Ta viết T(n) = O(f(n)) (đọc: T(n) là ô lớn của f(n)), nếu và chỉ nếu tồn tại các hằng số dương c và n_0 sao cho $T(n) \le c.f(n)$, với $\forall n > n_0$.

Nếu một giải thuật có thời gian thực hiện T(n) = O(f(n)), chúng ta sẽ nói rằng giải thuật có thời gian thực hiện cấp f(n).

Ví dụ: Giả sử $T(n) = 10n^2 + 4n + 4$

Ta có: $T(n) \le 10n^2 + 4n^2 + 4n^2 = 18n^2, \forall n \ge 1$

Vậy $T(n) = O(n^2)$. Trong trường hợp này ta nói giải thuật có độ phức tạp (có thời gian thực hiện) cấp n^2.

Bảng sau đây cho ta các cấp thời gian thực hiện giải thuật được sử dụng rộng rãi nhất và tên gọi thông thường của chúng.

Ký hiệu ô lớn (O)	Tên gọi thông thường
O(1)	Hằng
$O(\log_2 n)$	logarit
O(n)	Tuyến tính
$O(n\log_2 n)$	$n\log_2 n$
$O(n^2)$	Bình phương
$O(n^3)$	Lập phương
$O(2^n)$	Mũ

Danh sách trên sắp xếp theo thứ tự tăng dần của cấp thời gian thực hiện.

Các hàm như $\log_2 n$, n, $n\log_2 n$, n^2, n^3 được gọi là các hàm đa thức. Giải thuật với thời gian thực hiện có cấp hàm đa thức thì thường chấp nhận được.

Các hàm như 2^n, n!, n^n được gọi là hàm loại mũ. Một giải thuật mà thời gian thực hiện của nó là các hàm loại mũ thì tốc độ rất chậm.

4.3.4. Xác định độ phức tạp tính toán

Xác định độ phức tạp tính toán của một giải thuật bất kỳ có thể dẫn đến những bài toán phức tạp. Tuy nhiên, trong thực tế, đối với một số giải thuật ta cũng có thể phân tích được bằng một số qui tắc đơn giản.

a. Qui tắc tổng:

Giả sử $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của hai giai đoạn chương trình P_1 và P_2 mà $T_1(n) = O(f(n))$; $T_2(n) = O(g(n))$ thì thời gian thực hiện đoạn P_1 rồi P_2 tiếp theo sẽ là $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$.

Ví dụ: Trong một chương trình có 3 bước thực hiện mà thời gian thực hiện từng bước lần lượt là $O(n^2)$, $O(n^3)$ và $O(n\log_2 n)$ thì thời gian thực hiện 2 bước đầu là $O(\max(n^2, n^3)) = O(n^3)$. Khi đó thời gian thực hiện chương trình sẽ là $O(\max(n^3, n\log_2 n)) = O(n^3)$.

b. Qui tắc nhân:

Nếu tương ứng với P_1 và P_2 là $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$ thì thời gian thực hiện P_1 và P_2 lồng nhau sẽ là: $T_1(n).T_2(n) = O(f(n).g(n))$

Trong tài liệu quốc tế các giải thuật thường được trình bày dưới dạng các thủ tục hoặc hàm trong ngôn ngữ tựa Pascal/C. Để đánh giá thời gian thực hiện giải thuật, ta cần biết cách đánh giá thời gian thực

hiện các câu lệnh của Pascal/C. Các câu lệnh trong Pascal/C được định nghĩa đệ qui như sau:

1. Các phép gán, đọc, viết, **goto** là các câu lệnh. Các lệnh này gọi là lệnh đơn

2. Nếu S_1, S_2,..., S_n là các câu lệnh thì:

$\{S_1; S_2;...; S_n\}$

là câu lệnh và được gọi là lệnh hợp thành (hoặc khối lệnh).

3. Nếu S_1 và S_2 là các câu lệnh và E là biểu thức logic thì:

if (E) S_1; ***else*** S_2;

và ***if*** (E) S_1;

là câu lệnh và được gọi là lệnh ***if* - lệnh rẽ nhánh điều kiện.**

4. Nếu S_1, S_2,..., S_{n+1} là các câu lệnh, E là biểu thức có kiểu thứ tự đếm được, và v_1, v_2, ..., v_n là các giá trị cùng kiểu với E thì:

```
switch (E)
{
        v1 : S1 ; break;
        v2 : S2 ; break;
        ...
        vn : Sn ; break;
        [default: Sn+1;]
}
```

là câu lệnh và được gọi là lệnh ***switch* - lệnh rẽ nhánh lựa chọn.**

5. Nếu S là câu lệnh và E là biểu thức logic thì:

while (E) S;

là câu lệnh và được gọi là lệnh ***while* - vòng lặp.**

6. Nếu S_1, S_2,..., S_n là các câu lệnh, E là biểu thức logic thì:

do $\{S_1, S_2,..., S_n\}$ ***while*** (E);

là câu lệnh và được gọi là lệnh ***do ... while* - vòng lặp**

7. Với S là câu lệnh, E_1 và E_2 là các biểu thức có cùng một kiểu thứ tự đếm được, thì:

for (i= E_1; i<=E_2; i++) S;

là câu lệnh, và

for (i= E_2; i>=E_1; i--) S;

là câu lệnh. Lệnh này được gọi là lệnh ***for* - vòng lặp.**

Giả sử rằng, các lệnh gán không chứa các lời gọi hàm. Khi đó để đánh giá thời gian thực hiện một chương trình, ta có thể áp dụng phương pháp đệ qui sau:

1. Thời gian thực hiện các lệnh đơn: gán, đọc, viết là O(1).

2. Lệnh hợp thành: thời gian thực hiện lệnh hợp thành được xác định bởi luật tổng.

3. Lệnh ***if***: Giả sử thời gian thực hiện các lệnh S_1, S_2 là O(f(n)) và O(g(n)) tương ứng. Khi đó thời gian thực hiện lệnh ***if*** là O(max (f(n), g(n))).

4. Lệnh ***switch***: Lệnh này được đánh giá như lệnh ***if***.

5. Lệnh ***while***: Giả sử thời gian thực hiện lệnh S (thân của ***while***) là O(f(n)) và g(n) là số tối đa các lần thực hiện lệnh S, khi đó thời gian thực hiện lệnh ***while*** là O(f(n).g(n)).

6. Lệnh ***do...while***: Giả sử thời gian thực hiện khối {S_1, S_2, ..., S_n} là O(f(n)) và g(n) là số lần lặp tối đa. Khi đó thời gian thực hiện lệnh **do...*while*** là O(f(n).g(n)).

7. Lệnh ***for***: Lệnh này được đánh giá tương tự như lệnh ***do...while*** và ***while***.

Đánh giá thủ tục (hoặc hàm) đệ qui:

Trước hết chúng ta xét một ví dụ cụ thể, ta sẽ đánh giá thời gian thực hiện của hàm đệ qui sau:

```
int fact (int n)
{
        if (n <= 1) return 1;
        else return n* fact (n - 1);
}
```

Trong hàm này kích thước của dữ liệu vào là n, giả sử thời gian thực hiện hàm là T(n).

- Với n = 1, chỉ cần thực hiện lệnh gán fact = 1, do đó T(1) = O(1).

- Với n > 1, cần thực hiện lệnh gán fact = n*fact(n - 1). Do đó thời gian T(n) là O(1) (để thực hiện phép nhân và phép gán) cộng với T(n -1) (để thực hiện lời gọi đệ qui fact(n - 1)).

Tóm lại, ta có quan hệ đệ qui sau:

$$T(1) = O(1)$$

$$T(n) = O(1) + T(n - 1)$$

Thay các O(1) bởi các hằng nào đó, ta nhận được quan hệ đệ quy sau:

$$T(1) = C_1$$

$$T(n) = C_2 + T(n - 1)$$

Để giải phương trình đệ qui, tìm T(n), chúng ta áp dụng phương pháp thế lặp. Ta có phương trình đệ qui:

$$T(m) = C_2 + T(m - 1), \text{ với } m > 1$$

Thay m lần lượt bởi 2, 3,..., n - 1, n, ta nhận được các quan hệ sau:

$$T(2) = C_2 + T(1)$$

$$T(3) = C_2 + T(2)$$

...

$$T(n -1) = C_2 + T(n -2)$$

$$T(n) = C_2 + T(n - 1)$$

Bằng các phép thế liên tiếp, ta nhận được:

$$T(n) = (n - 1) C_2 + T(1)$$

hay $T(n) = (n - 1) C_2 + C_1$, trong đó C_1 và C_2 là các hằng nào đó.

Do đó, $T(n) = O(n)$.

Từ ví dụ trên, suy ra phương pháp tổng quát sau đây để đánh giá thời gian thực hiện thủ tục (hàm) đệ qui. Để đơn giản, ta giả thiết rằng các thủ tục (hàm) là đệ qui trực tiếp. Điều đó có nghĩa là các thủ tục (hàm) chỉ chứa các lời gọi đệ qui đến chính nó. Giả sử thời gian thực hiện thủ tục là $T(n)$, với n là kích thước dữ liệu vào. Khi đó thời gian thực hiện các lời gọi đệ qui được đánh giá thông qua các bước sau:

- Đánh giá thời gian thực hiện $T(n_0)$, với n_0 là cỡ dữ liệu vào nhỏ nhất có thể được (trong ví dụ trên, đó là $T(1)$).

- Đánh giá thân của thủ tục theo qui tắc 1-7 (qui tắc đánh giá thời gian thực hiện các câu lệnh) ta sẽ nhận được quan hệ đệ qui sau:

$$T(n) = F(T(m_1), T(m_2), ..., T(m_k))$$

Trong đó $m_1, m_2, ..., m_k < n$. Giải phương trình đệ qui này, ta sẽ nhận được sự đánh giá của $T(n)$.

4.4. Phân tích một số giải thuật

Ví dụ 1: Phân tích giải thuật Euclid

```
int Euclid (int m, int n)
{
  int r ;
  r = m % n;                          //(1)
  while (r!= 0)                       //(2)
        {
              m = n;                  //(3)
              n = r;                  //(4)
              r = m % n;              //(5)
        }
  return n;                           //(6)
}
```

Thời gian thực hiện giải thuật phụ thuộc vào số nhỏ nhất trong hai số m và n. Giả sử $m \geq n > 0$, khi đó cỡ của dữ liệu vào là n. Các lệnh (1) và (6) có thời gian thực hiện là O(1) vì chúng là các câu lệnh gán. Do đó thời gian thực hiện giải thuật là thời gian thực hiện lệnh ***while***, ta đánh giá thời gian thực hiện câu lệnh (2). Thân của lệnh này là khối gồm ba lệnh (3), (4) và (5). Mỗi lệnh có thời gian thực hiện là O(1). Do đó khối có thời gian thực hiện là O(1). Ta còn phải đánh giá số lớn nhất các lần thực hiện lặp khối.

Ta có: $m = n.q_1 + r_1$, $0 \leq r_1 < n$

$n = r_1.q_2 + r_2$, $0 \leq r_2 < r_1$

Nếu $r_1 \leq n/2$ thì $r_2 < r_1 \leq n/2$, do đó $r_2 < n/2$

Nếu $r_1 > n/2$ thì $q_2 = 1$, tức là $n = r_1 + r_2$, do đó $r_2 < n/2$.

Tóm lại, ta luôn có $r_2 < n/2$.

Như vậy cứ hai lần thực hiện khối lệnh thì phần dư r giảm đi còn một nửa của n. Gọi k là số nguyên lớn nhất sao cho $2^k \leq n$. Suy ra số lần lặp tối đa là $2k + 1 \leq 2\log_2 n + 1$. Do đó thời gian thực hiện lệnh ***while*** là $O(\log_2 n)$. Đó cũng là thời gian thực hiện của giải thuật.

Ví dụ 2: Giải thuật tính giá trị của e^x tính theo công thức gần đúng

$e^x = 1 + x/1! + x^2/2! + ... + x^n/n!$, với x và n cho trước

```
float Exp1 (int n, float x)
{
        //Tính từng số hạng sau đó cộng dồn lại
        float s, p;
        int i, j;
        s = 1;                                  //(1)
        for (i =1; i<=n; i++)                   //(2)
        {       p = 1;                          //(3)
                for (j =1; j<=i; j++)           //(4)
                    p = p*x/j;                  //(5)
                    s = s + p;                  //(6)
        }
  return s;                                     //(7)
}
```

Ta thấy câu lệnh (1) và (7) là các câu lệnh gán nên chúng có thời gian thực hiện là O(1). Do đó, thời gian thực hiện của giải thuật phụ thuộc vào câu lệnh (2). Ta đánh giá thời gian thực hiện câu lệnh này. Trong thân của câu lệnh này bao gồm các lệnh (3), (4), (5) và (6). Hai câu lệnh (3) và (7) có thời gian thực hiện là O(n) vì mỗi câu lệnh được thực hiện n lần. Riêng câu lệnh (5) thì thời gian thực hiện nó còn phụ thuộc vào câu lệnh (4) nên ta còn phải đánh giá thời gian thực hiện câu lệnh (4).

Với i = 1 thì câu lệnh (5) được thực hiện 1 lần

Với i = 2 thì câu lệnh này được thực hiện 2 lần

...

Với i = n thì câu lệnh này được thực hiện n lần

Suy ra tổng số lần thực hiện câu lệnh (5) là:

$$1 + 2 + ... + n = n(n + 1)/2 \text{ lần}$$

Do đó thời gian thực hiện câu lệnh này là $O(n^2)$ và đây cũng là thời gian thực hiện của giải thuật.

Ta có thể viết giải thuật này theo cách khác: Dựa vào số hạng trước để tính số hạng sau:

$$\frac{x^2}{2!} = \frac{x}{1!}.\frac{x}{2} \ldots\ldots \frac{x^n}{n!} = \frac{x^{n-1}}{(n-1)!}.\frac{x}{n}$$

Giải thuật được viết dưới dạng hàm như sau:

```
float Exp2 (int n, float x)
{ float s, p;
  int i;
  s = 1;                                //(1)
  p = 1;                                //(2)
  for (i =1; i<= n; i++)                //(3)
  {       p = p*x/i;                    //(4)
          s = s + p;                    //(5)
  }
  return s;                             //(6)
}
```

Tương tự như giải thuật trước, các câu lệnh (1), (2), (6) có thời gian thực hiện là O(1). Do đó thời gian thực hiện giải thuật phụ thuộc vào câu lệnh (3). Vì hai câu lệnh (4) và (5) đều có thời gian thực hiện là O(n) nên thời gian thực hiện của giải thuật là O(n).

Như vậy từ hai giải thuật trên ta có thể nói rằng giải thuật thứ hai có nhiều ưu điểm hơn giải thuật thứ nhất với n đủ lớn (với n nhỏ thì thời gian thực hiện hai giải thuật này tương đương nhau).

Ví dụ 3: Tìm trong dãy số $s_1, s_2, ..., s_n$ một phần tử có giá trị bằng x cho trước

Vào: Dãy $s_1, s_2, ..., s_n$ và khoá cần tìm x

Ra: Vị trí phần tử có khoá x hoặc là n + 1 nếu không tìm thấy.

```
int linear_search(int s[ ], int n, int x)
{ int i;
  i = 0;
  do
        {
              i = i + 1;
        }
  while (i<= n || s[i] != x);
  return i;
}
```

Ví dụ này ta không thể đánh giá như hai ví dụ trên. Do quá trình tìm kiếm không những phụ thuộc vào kích thước của dữ liệu vào, mà còn phụ thuộc vào tình trạng của dữ liệu. Tức là thời gian thực hiện giải thuật còn phụ thuộc vào vị trí của phần tử trong dãy bằng x. Quá trình tìm kiếm chỉ dừng khi tìm thấy phần tử bằng x, hoặc duyệt hết dãy mà không tìm thấy. Vì vậy, trong những trường hợp như trên ta cần phải đánh giá thời gian tính tốt nhất, tồi nhất và trung bình của giải thuật với kích thước đầu vào n. Rõ ràng thời gian tính của giải thuật có thể được đánh giá bởi số lần thực hiện câu lệnh i = i + 1 (gọi là phép toán tích cực).

Nếu s[1] = x thì câu lệnh i = i + 1 trong thân vòng lặp ***do...while*** thực hiện 1 lần. Do đó thời gian tính tốt nhất của giải thuật là O(1).

Nếu x không xuất hiện trong dãy khoá đã cho, thì câu lệnh i = i + 1 được thực hiện n lần. Vì thế thời gian tính tồi nhất là O(n).

Cuối cùng ta tính thời gian tính trung bình của giải thuật. Nếu x được tìm thấy ở vị trí thứ i của dãy thì câu lệnh i = i + 1 phải thực hiện i lần (i = 1, 2,..., n), còn nếu x không xuất hiện trong dãy thì câu lệnh i = i + 1 phải thực hiện n lần.

Từ đó suy ra số lần trung bình phải thực hiện câu lệnh i = i + 1 là:

$$[(1 + 2 + ... + n) + n] /(n + 1)$$

Ta có:

$$[(1 + 2 + ... + n) + n] /(n + 1) \leq (n^2 + n)/(n + 1) = n$$

Vậy thời gian tính trung bình của giải thuật là O(n).

Nhận xét:

Việc xác định T(n) trong trường hợp trung bình thường gặp nhiều khó khăn vì sẽ phải dùng tới công cụ toán đặc biệt, hơn nữa tính trung bình có nhiều cách quan niệm. Trong các trường hợp mà T(n) trung bình thường khó xác định, người ta thường đánh giá giải thuật qua giá trị xấu nhất của T(n). Hơn nữa, trong một số lớp giải thuật, việc xác định trường hợp xấu nhất là rất quan trọng.

Qua chương này ta thấy rằng giai đoạn thiết kế hoặc lựa chọn các cấu trúc dữ liệu và các giải thuật khi phát triển dự án phần mềm là hết sức quan trọng, nó quyết định một phần sự thành bại của một dự án tin học. Chất lượng của dự án cũng phụ thuộc vào việc đánh giá các cấu trúc dữ liệu và các giải thuật khi chúng được thiết kế và lựa chọn.

Trong các chương sau, cuốn sách sẽ trình bày một số các giải thuật và cấu trúc dữ liệu cơ bản, được áp dụng chủ yếu trong các bài toán quản lý như các giải thuật tìm kiếm, sắp xếp, cấu trúc danh sách tuyến tính, cây, v.v... cùng với cách thiết kế, đánh giá và áp dụng chúng trong bài toán thực tế.

BÀI TẬP CHƯƠNG 1

1. Nêu khái niệm thuật toán, các đặc trưng và các phương pháp biểu diễn thuật toán. Cho ví dụ minh họa.
2. Phân biệt khái niệm thuật toán và thuật giải. Cho ví dụ minh họa.
3. Tìm thêm các ví dụ minh họa mối quan hệ giữa cấu trúc dữ liệu và giải thuật (kiểu số nguyên, kiểu số thực,...).
4. Phân biệt cấu trúc dữ liệu và cấu trúc lưu trữ. Cho ví dụ minh họa.
5. Nêu nguyên tắc của phương pháp thiết kế top-down. Cho ví dụ minh họa.
6. Chứng minh rằng nếu $T(n) = O(n)$ thì $1(n) = O(n^2)$.
7. Cho $f(n) = 10n^2 + n - 5$. Chứng minh rằng $f(n) = O(n^2)$.
8. Chứng minh rằng $\lg n! = O(n \lg n)$.
9. Với các đoạn chương trình dưới đây, hãy xác định về thời gian của giải thuật bằng ký pháp chữ O lớn.

a.

```
for (i = 1; i <= n; i++)
        for (j = 1; j <= i; j++)
                x = x + 1 ;
```

b.

```
j = n;
while (j >= 1)
        {
                for (i = 1; i <= j; i++)
                        x = x + 1;
                        j = j/2;
        }
```

10. Cho $f(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$ ($a_n \neq 0$). Chứng minh rằng $f(x) = O(x^n)$.

Chương 2

ĐỆ QUY VÀ GIẢI THUẬT ĐỆ QUY

1. KHÁI NIỆM VỀ ĐỆ QUY

Ta nói một đối tượng là đệ quy nếu nó bao gồm chính nó như một bộ phận hoặc nó được định nghĩa dưới dạng của chính nó.

Ví dụ: Trong toán học ta gặp các định nghĩa đệ quy sau:

+ *Số tự nhiên:*

- 1 là số tự nhiên.

- n là số tự nhiên nếu n-1 là số tự nhiên.

+ *Hàm n giai thừa: n!*

- 0! = 1

- Nếu n > 0 thì n! = n(n-1)!

2. GIẢI THUẬT ĐỆ QUY VÀ THỦ TỤC ĐỆ QUY

2.1. Giải thuật đệ quy

Nếu lời giải của của một bài toán T được giải bằng lời giải của một bài toán T1, có dạng giống như T, thì lời giải đó được gọi là lời giải đệ quy. Giải thuật tương ứng với lời giải đệ quy gọi là giải thuật đệ quy.

Ở đây T1 có dạng giống T nhưng theo một nghĩa nào đó T1 phải "nhỏ" hơn T.

Chẳng hạn với bài toán tính n!, thì tính n! là bài toán T còn tính (n-1)! là bài toán T1 ta thấy T1 cùng dạng với T nhưng nhỏ hơn (n-1 < n).

Hay với bài toán tìm một từ trong quyển từ điển. Có thể nêu giải thuật như sau:

if (từ điển là một trang)

tìm từ trong trang này

else

{ *Mở từ điển vào trang "giữa"*

Xác định xem nửa nào của từ điển chứa từ cần tìm;

if *(từ đó nằm ở nửa trước) tìm từ đó ở nửa trước*

else *tìm từ đó ở nửa sau.*

}

Giải thuật này được gọi là giải thuật đệ quy. Việc tìm từ trong quyển từ điển được được giải quyết bằng bài toán nhỏ hơn đó là việc tìm từ trong một nửa thích hợp của quyển từ điển.

Ta thấy có hai điểm chính cần lưu ý:

1. Sau mỗi lần từ điển được tách làm đôi thì một nửa thích hợp sẽ lại được tìm bằng một chiến thuật như đã dùng trước đó (nửa này lại được tách đôi).

2. Có một trường hợp đặc biệt, đó là sau nhiều lần tách đôi từ điển chỉ còn một trang. Khi đó việc tách đôi ngừng lại và bài toán trở thành đủ nhỏ để ta có thể tìm từ mong muốn bằng cách tìm tuần tự. Trường hợp này gọi là ***trường hợp suy biến***.

2.2. Thủ tục đệ quy

Với giải thuật tìm kiếm như trên ta viết một thủ tục tương ứng như sau:

SEARCH(dict, word) //Tìm từ word trong từ điển dict

{

if (Từ điển chỉ còn là một trang)

tìm từ word trong trang này

else

{ mở từ điển vào trang giữa

```
            xác định xem nửa nào của từ điển chứa từ word
            if (từ word nằm ở nửa trước của từ điển)
                    return SEARCH(dict\{nửa sau}, word);
            else    return SEARCH(dict\{nửa trước}, word);
    }
}
```

Thủ tục trên được gọi là thủ thục đệ quy. Nó có những đặc điểm cơ bản sau:

1. Trong thủ tục đệ quy có lời gọi đến chính thủ tục đó. Ở đây trong thủ tục SEARCH có lời gọi call SEARCH (lời gọi này được gọi là lời gọi đệ quy).

2. Sau mỗi lần có lời gọi đệ quy thì kích thước của bài toán được thu nhỏ hơn trước. Ở đây khi có lời gọi call SEARCH thì kích thước từ điển chỉ còn bằng một nửa so với trước đó.

3. Có một trường hợp đặc biệt, trường hợp suy biến là khi lời gọi call SEARCH với từ điển dict chỉ còn là một trang. Khi trường hợp này xảy ra thì bài toán còn lại sẽ được giải quyết theo một cách khác hẳn (tìm từ word trong trang đó bằng cách tìm kiếm tuần tự) và việc gọi đệ quy cũng kết thúc. Chính tình trạng kích thước bài toán giảm dần sau mỗi lần gọi đệ quy, dẫn tới trường hợp suy biến.

Một số ngôn ngữ cấp cao như: Pascal, C, Algol, v.v... cho phép viết các thủ tục đệ quy. Nếu thủ tục đệ quy chứa lời gọi đến chính nó thì gọi là đệ quy trực tiếp. Cũng có trường hợp thủ tục chứa lời gọi đến thủ tục khác mà ở thủ tục này lại chứa lời gọi đến nó. Trường hợp này gọi là đệ quy gián tiếp.

3. THIẾT KẾ GIẢI THUẬT ĐỆ QUY

Khi bài toán đang xét hoặc dữ liệu đang xử lý được định nghĩa dưới dạng đệ quy thì việc thiết kế các giải thuật đệ quy tỏ ra rất thuận lợi. Hầu như nó phản ánh rất sát nội dung của định nghĩa đó.

Không có giải thuật đệ quy vạn năng cho tất cả các bài toán đệ quy, nghĩa là mỗi bài toán cần thiết kế một giải thuật đệ quy riêng.

Ta xét một số bài toán sau:

3.1. Hàm n!

Hàm này được định nghĩa như sau:

$$Factorial(n) = \begin{cases} 1 & \text{nếu } n = 0 \\ n * \text{Factorial}(n - 1) & \text{nếu } n > 0 \end{cases}$$

Giải thuật đệ quy được viết dưới dạng hàm dưới đây:

```
Factorial (n)
{
        if (n==0) return 1;
        else    return n*Factorial(n-1);
}
```

Trong hàm trên lời gọi đến nó nằm ở câu lệnh gán sau ***else***.

Mỗi lần gọi đệ quy đến Factorial, thì giá trị của n giảm đi 1. Ví dụ, Factorial(4) gọi đến Factorial(3), gọi đến Factorial(2), gọi đến Factorial(1), gọi đến Factorial(0) đây là trường hợp suy biến, nó được tính theo cách đặc biệt Factorial(0) = 1.

3.2. Bài toán dãy số FIBONACCI

Dãy số Fibonacci bắt nguồn từ bài toán cổ về việc sinh sản của các cặp thỏ. Bài toán được đặt ra như sau:

- Các con thỏ không bao giờ chết.

- Hai tháng sau khi ra đời một cặp thỏ mới sẽ sinh ra một cặp thỏ con.

- Khi đã sinh, thì cứ sau mỗi tháng chúng lại sinh được một cặp con mới.

Giả sử bắt đầu từ một cặp thỏ mới sinh, hỏi đến tháng thứ n sẽ có bao nhiêu cặp?

Ví dụ: Với n = 6, ta thấy:

Tháng thứ 1: 1 cặp (cặp ban đầu)

Tháng thứ 2: 1 cặp (cặp ban đầu vẫn chưa sinh con)

Tháng thứ 3: 2 cặp (đã có thêm 1 cặp con do cặp ban đầu sinh ra)

Tháng thứ 4: 3 cặp (cặp ban đầu vẫn sinh thêm)

Tháng thứ 5: 5 cặp (cặp con bắt đầu sinh)

Tháng thứ 6: 8 cặp (cặp con vẫn sinh tiếp)

Đặt F(n) là số cặp thỏ ở tháng thứ n. Ta thấy chỉ những cặp thỏ đã có ở tháng thứ n-2 mới sinh con ở tháng thứ n do đó số cặp thỏ ở tháng thứ n là:

F(n) = F(n-2) + F(n-1), vì vậy F(n) có thể được tính như sau:

$$F(n) = \begin{cases} 1 & \text{nếu } n \le 2 \\ F(n-2) + F(n-1) & \text{nếu } n > 2 \end{cases}$$

Dãy số thể hiện F(n) ứng với các giá trị của n = 1, 2, 3, 4..., có dạng

1 1 2 3 5 8 13 21 34 55....

nó được gọi là dãy số Fibonacci. Nó là mô hình của rất nhiều hiện tượng tự nhiên và cũng được sử dụng nhiều trong tin học.

Sau đây là giải thuật đệ quy dạng hàm thể hiện việc tính F(n).

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

Ở đây trường hợp suy biến ứng với 2 giá trị F(1) = 1 và F(2) = 1.

Chú ý:

Đối với hai bài toán nêu trên thì việc thiết kế các giải thuật đệ quy tương ứng khá thuận lợi vì cả hai đều thuộc dạng tính giá trị hàm mà định nghĩa của nó xác định được một cách dễ dàng.

Nhưng không phải lúc nào tính đệ quy trong cách giải bài toán cũng thể hiện rõ nét và đơn giản như vậy. Mà việc thiết kế một giải thuật đệ quy đòi hỏi phải giải đáp được các câu hỏi sau:

- Có thể định nghĩa được bài toán dưới dạng một bài toán cùng loại, nhưng nhỏ hơn như thế nào?

- Như thế nào là kích thước của bài toán được giảm đi ở mỗi lần gọi đệ quy?

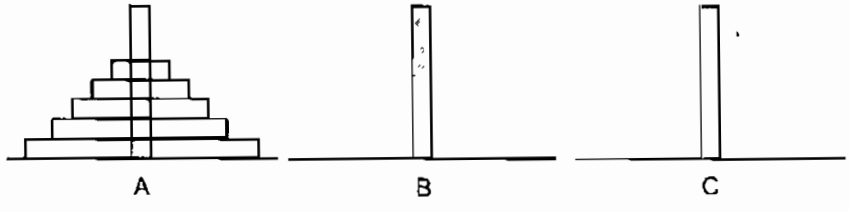
- Trường hợp đặc biệt nào của bài toán được gọi là trường hợp suy biến?

Sau đây ta xét thêm bài toán phức tạp hơn.

3.4. Bài toán "Tháp Hà Nội"

Bài toán này mang tính chất là một trò chơi, nội dung như sau:

Có n đĩa, kích thước nhỏ dần, mỗi đĩa có lỗ ở giữa. Có thể xếp chồng chúng lên nhau xuyên qua một cọc, đĩa to ở dưới, đĩa nhỏ ở trên để cuối cùng có một chồng đĩa dạng như hình tháp như hình 2.1.



Hình 2.1: Chồng đĩa trước khi chuyển

**** Yêu cầu đặt ra là:***

Chuyển chồng đĩa từ cọc A sang cọc khác, chẳng hạn cọc C, theo những điều kiện:

- Mỗi lần chỉ được chuyển một đĩa.

- Không khi nào có tình huống đĩa to ở trên đĩa nhỏ (dù là tạm thời).

- Được phép sử dụng một cọc trung gian, chẳng hạn cọc B để đặt tạm đĩa.

Để đi tới cách giải tổng quát, trước hết ta xét vài trường hợp đơn giản.

** Trường hợp có 1 đĩa:*

- Chuyển đĩa từ cọc A sang cọc C.

** Trường hợp 2 đĩa:*

- Chuyển đĩa thứ nhất từ cọc A sang cọc B.

- Chuyển đĩa thứ hai từ cọc A sang cọc C.

- Chuyển đĩa thứ nhất từ cọc B sang cọc C.

Ta thấy với trường hợp n đĩa (n>2) nếu coi n-1 đĩa ở trên, đóng vai trò như đĩa thứ nhất thì có thể xử lý giống như trường hợp 2 đĩa được, nghĩa là:

- Chuyển n-1 đĩa trên từ A sang B.

- Chuyển đĩa thứ n từ A sang C.

- Chuyển n-1 đĩa từ B sang C.

Lược đồ thể hiện 3 bước này như hình 5.1.

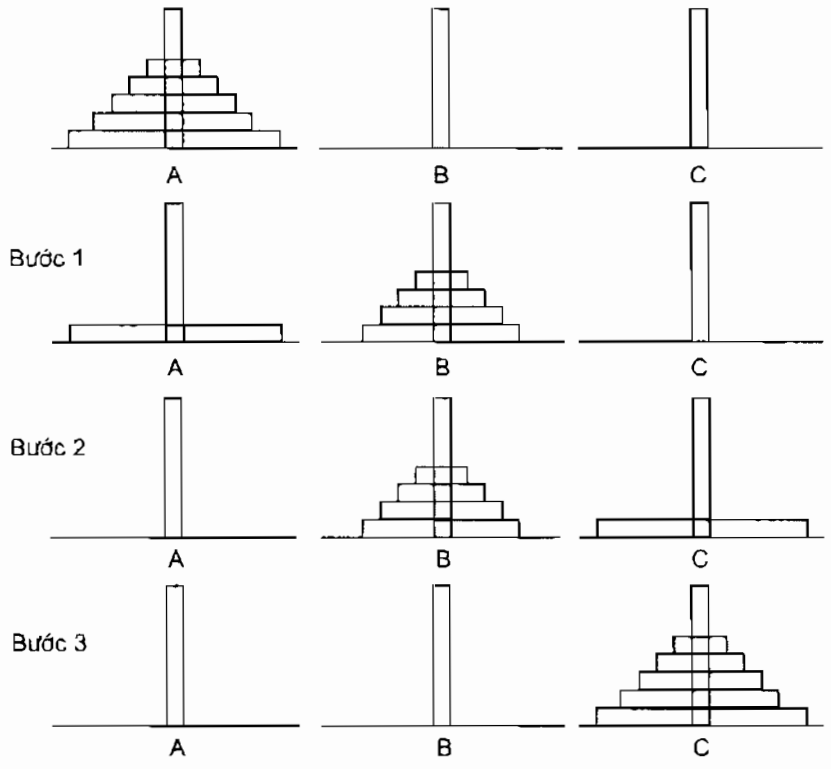
Như vậy, bài toán ***"Tháp Hà Nội"*** tổng quát với n đĩa đã được dẫn đến bài toán tương tự với kích thước nhỏ hơn, chẳng hạn từ chỗ chuyển n đĩa từ cọc A sang cọc C nay là chuyển n-1 đĩa từ cọc A sang cọc B và ở mức này thì giải thuật lại là:

- Chuyển n-2 đĩa từ cọc A sang cọc C.

- Chuyển 1 đĩa từ cọc A sang cọc B.

- Chuyển n-2 đĩa từ cọc B sang cọc C.

và cứ như thế cho tới khi trường hợp suy biến xảy ra, đó là trường hợp ứng với bài toán chuyển 1 đĩa.



Hình 2.2: Các bước chuyển đĩa

Vậy thì các đặc điểm của đệ quy trong giải thuật đã được xác định và ta có thể viết giải thuật đệ quy của bài toán "Tháp Hà Nội" như sau:

```
Chuyen(n, A, B, C)
{
        if (n= =1) chuyển đĩa từ A sang C;
        else
        {       call Chuyen(n-1, A, C, B);
                call Chuyen(1, A, B, C);
                call Chuyen(n-1, B, A, C);
        }
}
```

Bạn có thể tự cài đặt giải thuật này với cấu trúc dữ liệu biểu diễn bằng mảng một chiều, nếu tốt hơn có thể biểu diễn giải thuật bằng đồ họa.

4. HIỆU LỰC CỦA ĐỆ QUY

Qua các ví dụ trên ta có thể thấy: đệ quy là một công cụ để giải quyết các bài toán. Có những bài toán, bên cạnh giải thuật đệ quy vẫn có những giải thuật lặp khá đơn giản và hữu hiệu. Chẳng hạn giải thuật lặp tính n! có thể viết:

```
Factorial(n)
{
        if (n==0 || n==1) return 1;
  else
  {     gt=1;
        for (i=2; i<=n; i++)
                gt = gt*i;
        return gt;
  }
}
```

Hoặc ta xét giải thuật lặp tính số Fibonacci thứ n:

```
Fibonacci(n)
{
  if (n<=2)         return 1;
  else
  {     Fib1 = 1; Fib2 = 1;
        for (i=3; i<=n; i++)
        {
                Fibn = Fib1 + Fib2;
                Fib1= Fib2;
                Fib2 = Fibn;
        }
        return Fibn;
  }
}
```

Tuy vậy, đệ quy vẫn có vai trò xứng đáng của nó. Có những bài toán việc nghĩ ra giải thuật đệ quy thuận lợi hơn nhiều so với giải thuật lặp và có những giải thuật đệ quy thực sự có hiệu lực cao, chẳng hạn giải thuật sắp xếp kiểu phân đoạn (Quick Sort) hoặc các giải thuật duyệt cây nhị phân mà ta đã có dịp xét trong các chương trước của môn học này.

Một điều nữa cần nói thêm là: về mặt định nghĩa, công cụ đệ quy đã cho phép xác định một tập vô hạn các đối tượng bằng một phát biểu hữu hạn. Ta sẽ thấy vai trò của công cụ này trong định nghĩa văn phạm, định nghĩa cú pháp ngôn ngữ, định nghĩa một số cấu trúc dữ liệu v.v...

Chú thích: khi thay các giải thuật đệ quy bằng các giải thuật lặp tương ứng ta gọi là khử đệ quy. Tuy nhiên, có những bài toán việc khử đệ quy tương đối đơn giản (ví dụ: giải thuật tính n!, tính số fibonaci...), nhưng có những bài toán việc khử đệ quy là rất phức tạp (ví dụ: bài toán Tháp Hà Nội, giải thuật sắp xếp phân đoạn...).

BÀI TẬP CHƯƠNG 2

Bài 1: Xét định nghĩa đệ quy:

$$\text{Acker}(m, n) = \begin{cases} n + 1 \text{ nếu } m = 0 \\ \text{Acker}(m - 1, 1) \text{ nếu } n = 0 \\ \text{Acker}(m - 1, \text{Acker}(m, n - 1)) \text{ với các trường hợp khác} \end{cases}$$

- Hãy xác định Acker(1, 2)
- Viết hàm đệ quy thực hiện tính giá trị của hàm này.

Bài 2: Cho dãy số A = {4, 3, -2, -6, -5, 0, 4, ...}

- Xây dựng định nghĩa đệ quy cho việc tính A_n ($n \geq 0$)
- Xây dựng giải thuật đệ quy tính A_n
- Tính A_{10}.

Bài 3: Cho hàm số:

$$F(x) = \begin{cases} \cos(x) & \text{nếu } x = 0 \\ x & \text{nếu } x < 0 \\ F(x - \pi) + F(x - \pi/2) & \text{trong các trường hợp còn lại} \end{cases}$$

Yêu cầu:

- Tính $F(5\pi/2)$ và giải thích cách tính
- Thiết kế giải thuật đệ quy và viết hàm đệ quy để tính giá trị của hàm F.

Bài 4: Giải thuật tính ước số chung lớn nhất của hai số nguyên dương p, q (p > q) được cho như sau:

Gọi r là số dư trong phép chia p cho q.

Nếu r = 0 thì ước số chung lớn nhất là q.

Nếu r ≠ 0 thì gán cho p giá trị q, gán cho q giá trị của r và lặp lại quá trình.

a. Hãy xây dựng định nghĩa đệ quy cho hàm USCLN(p, q).

b. Viết một giải thuật đệ quy và một giải thuật lặp thể hiện hàm đó.

c. Hãy nêu rõ các đặc điểm của một giải thuật đệ quy được thể hiện trong trường hợp này.

d. Xử lý trường hợp p < q.

Bài 5: Nêu rõ các bước thực hiện khi có lời gọi hàm chuyen(3, A, B, C) trong bài toán Tháp Hà Nội.

Bài 6: Viết một hàm lặp và một hàm đệ quy thực hiện việc in ngược một chuỗi ký tự. Ví dụ "PASCAL" thì in ra là "LACSAP".

Bài 7: Viết một hàm lặp và một hàm đệ quy thực hiện việc đếm số chữ số của một số nguyên dương.

Bài 8: Cài đặt chương trình cho bài toán Tháp Hà Nội với cấu trúc dữ liệu là mảng một chiều (nếu xây dựng được chương trình biểu diễn bài toán bằng đồ họa thì càng tốt).

Chương 3 SẮP XẾP VÀ TÌM KIẾM

Sắp xếp và tìm kiếm là những vấn đề hết sức quen thuộc với chúng ta những người học và làm việc trong ngành công nghệ thông tin. Hơn nữa, với những người làm công tác quản lý hồ sơ, đây là công việc mà họ thường xuyên phải thực hiện, nó tiêu tốn khá nhiều thời gian. Nhưng với sự hỗ trợ của phần mềm tin học, công việc này trở nên rất đơn giản và rất nhanh, chỉ bằng những thao tác lựa chọn trên màn hình máy tính. Tuy nhiên, với chúng ta, những người làm tin học cần phải tìm hiểu gốc rễ của vấn đề, đó là các giải thuật được sử dụng để cài đặt cho máy tính, chúng sẽ được trình bày cụ thể trong chương này.

1. CÁC PHƯƠNG PHÁP SẮP XẾP

1.1. Khái niệm sắp xếp

Yêu cầu sắp xếp thường xuyên xuất hiện trong các ứng dụng tin học với các mục đích khác nhau, sắp xếp dữ liệu lưu trữ trong máy tính để thuận lợi cho việc tìm kiếm, sắp xếp các kết quả để in ra trên bảng biểu, v.v...

1.1.1. Khái niệm

Sắp xếp được hiểu là một quá trình bố trí lại vị trí các phần tử của một tập đối tượng nào đó theo một thứ tự xác định, chẳng hạn như thứ tự tăng dần của một dãy số, thứ tự từ điển đối với các chữ cái,...

Nhìn chung, dữ liệu được sắp xếp có thể xuất hiện dưới nhiều dạng khác nhau, nhưng ở đây ta quy ước tập đối tượng sắp xếp là tập các bản ghi, mỗi bản ghi gồm một số trường, dữ liệu tương ứng với

những thuộc tính khác nhau. Tuy nhiên, không phải toàn bộ các trường dữ liệu của bản ghi đều được xem xét đến trong quá trình sắp xếp mà chỉ dựa vào một hoặc một vài trường nào đó, trường như vậy được gọi là khoá. Quá trình sắp xếp sẽ được tiến hành dựa vào giá trị của khoá, vì vậy bài toán sắp xếp được phát biểu như sau:

1.1.2. Phát biểu bài toán sắp xếp

Vào: Dãy n đối tượng, mỗi đối tượng có một khóa sắp xếp

Ra: Dãy đối tượng với trật tự mới

Khóa sắp xếp ở đây phụ thuộc vào mục đích sắp xếp và dữ liệu cần sắp xếp. Chẳng hạn, sắp xếp dãy số thì giá trị của các số là khóa, còn sắp xếp danh sách sinh viên theo tên, thì tên của sinh viên là khóa sắp xếp, v.v...

Dưới đây ta sẽ tìm hiểu một số phương pháp sắp xếp hay dùng.

1.2. Ba phương pháp sắp xếp đơn giản

Để đơn giản, nhưng không mất tính tổng quát, chúng tôi mô tả các phương pháp sắp xếp thông qua bài toán sắp xếp dãy số nguyên, thực chất là sắp xếp mảng.

Bài toán sắp xếp:

Vào: Dãy **X** có **n** số nguyên: $X_1, X_2, \ldots, X_n$

Ra: Dãy **X** với các phần tử theo thứ tự tăng dần

1.2.1. Phương pháp lựa chọn (Selection Sort)

Một trong những phương pháp đơn giản nhất để thực hiện sắp xếp một mảng là sắp xếp dựa trên phép lựa chọn.

a. Nguyên tắc sắp xếp

Để sắp xếp dãy X theo chiều tăng dần người ta làm như sau:

- Thực hiện n-1 lần duyệt dãy từ trái sang phải.

- Mỗi lần duyệt, chọn phần tử nhỏ nhất, đổi chỗ cho phần tử đầu dãy được xét.

Cụ thể như sau:

Ở lần duyệt thứ i (i = 1, 2,..., n-1), ta duyệt dãy từ X_i đến X_n và chọn khoá có giá nhỏ nhất, giả sử là X_m (m = i, i+1,..., n) và đổi chỗ X_m với X_i.

Như vậy: sau i lần duyệt, i khoá nhỏ hơn đã lần lượt ở các vị trí thứ nhất, thứ hai,..., thứ i theo đúng thứ tự sắp xếp.

Phương pháp sắp xếp lựa chọn được mô tả qua ví dụ sau:

Ví dụ:

Bảng 3.1: Mô tả phương pháp sắp xếp lựa chọn

Lần duyệt	X_1	X_2	X_3	X_4	X_5	X_6	X_7	Giải thích
	42	23	74	**11**	65	58	34	
i = 1	11	**23**	74	42	65	58	34	Duyệt từ X_1 đến X_7, X_4 nhỏ nhất đổi chỗ cho X_1
i = 2	11	23	74	42	65	58	**34**	Duyệt từ X_2 đến X_7, X_2 nhỏ nhất. Đây là trường hợp đặc biệt, phần tử nhỏ nhất là phần tử đầu dãy. Vì thế thực tế không cần đổi chỗ
i = 3	11	23	34	**42**	65	58	74	Duyệt từ X_3 đến X_7, X_7 nhỏ nhất đổi chỗ cho X_3
i = 4	11	23	34	42	65	**58**	74	Giống trường hợp i = 2
i = 5	11	23	34	42	58	**65**	74	Duyệt từ X_5 đến X_7, X_6 nhỏ nhất đổi chỗ cho X_5
i = 6	11	23	34	42	58	65	74	Giống trường hợp i = 2

Trong mô tả trên, vị trí gạch chân là phần tử đầu dãy được xét, vị trí in đậm là phần tử có giá trị nhỏ nhất trong dãy được xét.

Bạn đọc có thể tự mình lấy ví dụ minh họa trong trường hợp dãy được sắp xếp theo chiều giảm dần.

b. Thiết kế giải thuật

Với nguyên tắc sắp xếp như trên, việc thiết kế giải thuật khá đơn giản với kỹ thuật lặp gồm các bước như sau:

1. Đếm số lần duyệt

for (i=1; i<=n-1; i++)

2. Duyệt dãy và chọn phần tử nhỏ nhất (*tương tự như giải thuật tìm max*).

Đặt m = i; //Giả định X_i là phần tử nhỏ nhất

for (j=i+1; j<=n; j++)

if ($X_j < X_m$) m = j;

3. Đổi chỗ X_m và X_i

Dưới đây là giải thuật sắp xếp bằng phương pháp lựa chọn, trong trường hợp dãy được sắp theo chiều tăng dần:

```
void SELECTION_SORT (int X[ ], int n)
//Sắp xếp dãy khóa X có n phần tử
{
  for (i = 0; i < n-1; i++)
  {
        m = i;
        for (j = i+1; j < n; j++)
              if (X[j] < X[m] )
                    m = j;
        if (m != i )
        {
              tg = X[i];
              X[i] = X[m];
              X[m] = tg;
        }
  }
}
```

1.2.2. Sắp xếp kiểu thêm dần (Insertion Sort)

a. Nguyên tắc sắp xếp:

Nguyên tắc sắp xếp của phương pháp này dựa theo kinh nghiệm của những người chơi bài. Khi có i-1 lá bài đã được sắp xếp ở trên tay, nếu rút thêm lá bài thứ i nữa thì sắp xếp thế nào? Câu trả lời là có thể so sánh lá bài mới lần lượt với lá bài thứ i-1, thứ i-2,... để tìm ra vị trí thích hợp của lá bài mới và chèn vào vị trí đó.

Dựa trên nguyên tắc này, có thể triển khai một cách sắp xếp như sau:

- Đầu tiên, dãy được coi chỉ gồm một khóa X_1 đã được sắp xếp.

- Xét thêm X_2, so sánh X_2 với X_1 để xác định vị trí chèn X_2, ta được một dãy gồm 2 khóa đã được sắp xếp.

- Xét thêm X_3, lại so sánh X_3 với X_2 và X_1 để xác định vị trí chèn X_3, tương tự như vậy đối với các khoá X_4, X_5,..., và cuối cùng là X_n, ta được bảng khoá đã sắp xếp hoàn toàn.

Có thể mô tả ngắn gọn phương pháp này như sau:

Để sắp xếp dãy X, có n phần tử X_1, X_2,..., X_n ta thực hiện n-1 lần chia dãy thành dãy đích và dãy nguồn:

- Dãy đích gồm các phần tử từ X_1 đến X_i (với i = 1, 2,..., n-1)

- Dãy nguồn gồm các phần tử từ X_{i+1} đến X_n

Mỗi lần chia lấy phần tử đầu dãy nguồn (là X_{i+1}) chèn vào vị trí thích hợp trong dãy đích.

Ví dụ:

Bảng 3.2: Mô tả phương pháp sắp xếp thêm dần

Lần chia	X_1	X_2	X_3	X_4	X_5	X_6	X_7	Giải thích
	42	23	74	11	65	58	34	
i = 1	42 →	**23**	74	11	65	58	34	Lấy t = X_2 = 23 chèn vào dãy đích
	42	42	74	11	65	58	34	
i = 2	23 →	42	**74**	11	65	58	34	Lấy t = X_3 = 74 chèn vào dãy đích
i = 3	23 →	42 →	74 →	**11**	65	58	34	Lấy t = X_4 = 11 chèn vào dãy đích
	23	23	42	74	65	58	34	
i = 4	11	23	42	74 →	**65**	58	34	Lấy t = X_5 = 65 chèn vào dãy đích
	11	23	42	***74***	74	58	34	
i = 5	11	23	42	65 →	74 →	**58**	34	Lấy t = X_6 = 58 chèn vào dãy đích
	11	23	42	***65***	65	74	34	
i = 6	11	23	42 →	58 →	65 →	74 →	**34**	Lấy t = X_7 = 34 chèn vào dãy đích
	11	23	***42***	42	58	65	74	
	11	23	34	42	58	65	74	Dãy được sắp

Trong mô tả này, các phần tử in đậm là dãy đích, phần tử gạch chân-in đậm là phần tử đầu dãy nguồn, các mũi tên chỉ việc dịch các phần tử sang phải, phần tử nghiêng-đậm là vị trí thích hợp trong dãy đích, để chèn phần tử đầu dãy nguồn.

Tương tự, bạn đọc có thể tự mô tả trường hợp dãy được sắp xếp theo chiều giảm dần.

b. Giải thuật:

Qua mô tả trên ta xây dựng giải thuật sắp xếp bằng phương pháp thêm dần dưới dạng một hàm như sau:

```
void INSERTION_SORT (int X[ ], int n)
{
  for (i = 0; i < n; i++)
  {
        t = X[i+1];
        j = i;
        while (X[j] > 0 && j > -1)
        {
              X[j+1] = X[j];
              j = j-1;
        }
        X[j+1] = t;
  }
}
```

1.2.3. Sắp xếp kiểu nổi bọt (Bubble Sort)

Trong hai phương pháp sắp xếp nêu trên, kỹ thuật đổi chỗ đều đã được sử dụng nhưng chưa trở thành điểm nổi bật. Ở phương pháp này, việc đổi chỗ các cặp khoá kế cận khi chúng ngược thứ tự sẽ được thực hiện thường xuyên cho tới khi toàn bộ dãy khoá được sắp xếp.

a. Nguyên tắc sắp xếp:

Với dãy X đã cho gồm n phần tử, để sắp xếp dãy theo chiều tăng dần người ta làm như sau:

- Thực hiện n-1 lần duyệt dãy từ trái sang phải.

- Mỗi lần duyệt, lần lượt so sánh các cặp phần tử liên tiếp, giả sử là X_j và X_{j+1} nếu chúng ngược thứ tự thì đổi chỗ.

Nguyên tắc nêu trên được mô tả trong bảng sau:

Bảng 3.3: Mô tả phương pháp sắp xếp nổi bọt

Lần duyệt	X_1	X_2	X_3	X_4	X_5	X_6	X_7	Giải thích
	42	23	74	11	65	58	34	
i = 1	**42**	**23**	74	11	65	58	34	So sánh cặp X_1, X_2 và đổi chỗ
	23	**42**	**74**	11	65	58	34	So sánh cặp X_2, X_3 và đổi chỗ
	23	42	**74**	**11**	65	58	34	So sánh cặp X_3, X_4 và đổi chỗ
	23	42	11	**74**	**65**	58	34	So sánh cặp X_4, X_5 và đổi chỗ
	23	42	11	65	**74**	**58**	34	So sánh cặp X_5, X_6 và đổi chỗ
	23	42	11	65	58	**74**	**34**	So sánh cặp X_6, X_7 và đổi chỗ
	23	42	11	65	58	34	**74**	Phần tử lớn nhất ở cuối dãy
i = 2	23	11	42	58	34	**65**	74	Tương tự với lần duyệt i = 1
i = 3	11	23	42	24	**58**	65	74	
i = 4	11	23	24	**42**	58	65	74	
i = 5	11	23	**24**	42	58	65	74	
i = 6	11	**23**	24	42	58	65	74	Dãy được sắp

Qua mô tả trên ta thấy:

- Sau mỗi lần duyệt ta được 1 phần tử đứng đúng vị trí.

- Cụ thể: sau lần duyệt thứ nhất (i = 1) ta có phần tử lớn nhất đứng cuối dãy.

- Ở lần duyệt thứ 2 (i = 2) ta chỉ cần xét các phần tử từ X_1 đến X_{n-1} (làm tương tự lần duyệt 1 ta có phần tử lớn nhất trong số n-1 phần tử còn lại đứng ở vị trí n-1)

- Vậy ở lần duyệt thứ i (i = 1, 2,..., n-1) ta chỉ xét n-i+1 phần tử đầu dãy.

- Vì so sánh các cặp liên tiếp nên trong lần duyệt thứ i (i = 1, 2,..., n-1) ta chỉ duyệt từ X_1 đến X_{n-i}.

Ta có giải thuật như sau:

b. Giải thuật:

```
void BUBBLE_SORT (int X[ ], int n)
{
  for (i = 1; i <= n-1; i++)
        for (j = 0; j < n-i; j++)
              if (X[j] > X[j + 1])
              {
                    tg = X[j]; X[j] = X[j + 1]; X[j + 1] = tg;
              }
}
```

Trên đây ta vừa tìm hiểu ba phương pháp sắp xếp đơn giản, ta sẽ xem xét tính hiệu quả của nó trong phần dưới đây.

1.2.4. Đánh giá ba phương pháp sắp xếp đơn giản

Đối với một số phương pháp sắp xếp, khi xét tới hiệu lực của nó, ngoài những đánh giá về mặt không gian nhớ cần thiết, người ta thường lưu ý đặc biệt tới chi phí về thời gian. Mà thời gian thì chủ yếu phụ thuộc vào việc thực hiện các phép so sánh giá trị khoá và các phép chuyển chỗ bản ghi khi sắp xếp. Vì vậy thông thường người ta lấy số lượng trung bình các phép so sánh và các phép chuyển chỗ làm đại lượng đặc trưng cho chi phí về thời gian thực hiện của từng phương pháp. Tuy nhiên, ở đây ta chỉ xét tới phép so sánh và coi nó như một "phép toán tích cực" của các giải thuật sắp xếp.

Đối với phương pháp sắp xếp kiểu lựa chọn, ta thấy: ở lượt thứ i (i = 1, 2,..., n-1) để tìm khoá nhỏ nhất bao giờ cũng cần $C_i = (n-i)$ phép so sánh. Số lượng phép so sánh này không phụ thuộc gì vào tình trạng ban đầu của dãy khoá cả. Từ đó suy ra:

$$C_{min} = C_{max} = C_{tb} = \sum_{i=1}^{n-1}(n-1) = \frac{n(n-1)}{2}$$

Còn với phương pháp sắp xếp kiểu thêm dần (giải thuật INSERTION_SORT) thì có hơi khác. Rõ ràng số lượng phép so sánh phụ thuộc vào dãy khoá ban đầu. Trường hợp thuận lợi nhất ứng với dãy khoá đã được sắp xếp rồi. Như vậy ở mỗi lượt chỉ cần 1 phép so sánh. Do đó

$$C_{min} = \sum_{i=2}^{n} 1 = n-1$$

Nhưng nếu dãy khoá ban đầu có thứ tự ngược với thứ tự sắp xếp thì ở lượt thứ i phải cần có: $C_i = (i-1)$ phép so sánh. Vì vậy:

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

Nếu giả sử mọi giá trị khoá đều xuất hiện đồng khả năng thì trung bình ở lượt thứ i có thể coi như $C_i = i/2$ phép so sánh. Suy ra:

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

Nhìn vào các kết quả đánh giá ở trên ta thấy INSERTION_SORT tỏ ra có "tốt hơn" so với hai phương pháp kia. Tuy nhiên, với n khá lớn, chi phí về thời gian thực hiện được đánh giá qua cấp độ lớn, thì cả ba phương pháp đều có cấp $O(n^2)$ và đây vẫn là một chi phí cao so với một số phương pháp mà ta sẽ xét thêm sau đây.

1.3. Sắp xếp phân đoạn

1.3.1. Giới thiệu phương pháp

Sắp xếp kiểu phân đoạn (PARTITION_SORT) là một cải tiến của phương pháp sắp xếp đổi chỗ và là một phương pháp sắp xếp khá tốt. Chính vì thế, C.A.R. Hoare người tạo ra phương pháp này đã đặt tên cho nó là sắp xếp nhanh (QUICK_SORT).

Nguyên tắc của phương pháp này có thể được mô tả như sau:

- Chọn một khoá làm "*chốt*", thông thường để thuận lợi cho việc cài đặt, người ta chọn khóa trung tâm của "*đoạn được xét*" làm chốt.

- Xếp các khoá nhỏ hơn chốt về bên trái chốt (phía đầu dãy), các khoá lớn hơn chốt về bên phải chốt (phía cuối dãy) bằng cách:

Các phần tử trong dãy được so sánh với khoá chốt và sẽ đổi vị trí cho nhau hoặc cho chốt nếu chúng nằm trước chốt mà lại lớn hơn chốt hoặc nằm sau chốt mà lại nhỏ hơn chốt.

- Kết thúc một lượt đổi chỗ, dãy khoá được chia thành hai đoạn: một đoạn gồm các khoá nhỏ hơn chốt, một đoạn gồm các khoá lớn hơn chốt và (có thể) khoá chốt nằm ở giữa hai đoạn, cũng chính là vị trí đúng của nó trong dãy.

Lặp lại quá trình trên đối với từng phân đoạn cho đến khi toàn bộ dãy khoá được sắp xếp. Ở các lượt xử lý tiếp theo, chỉ có một phân đoạn được xử lý, còn phân đoạn còn lại phải được ghi nhớ và xử lý sau.

1.3.2. Minh hoạ phương pháp phân đoạn

Sắp xếp dãy khoá X:

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
42	23	74	11	65	58	94	36	99	87

Với quy ước chọn khoá chốt là khóa trung tâm trong dãy đang xét, như vậy với dãy khoá ban đầu thì khoá chốt là $t = X_5 = 65$.

Giả sử cần sắp xếp một đoạn từ X_{left} đến X_{right} trên dãy. Để xác định vị trí các khoá cần đổi chỗ cho nhau, ta dùng hai biến i và j để duyệt từ hai đầu của dãy khóa như sau:

(1) Ban đầu, i = left; j = right;

(lượt sắp đầu tiên left = 1, right = 10, và $t = X_5 = 65$ với dãy trên)

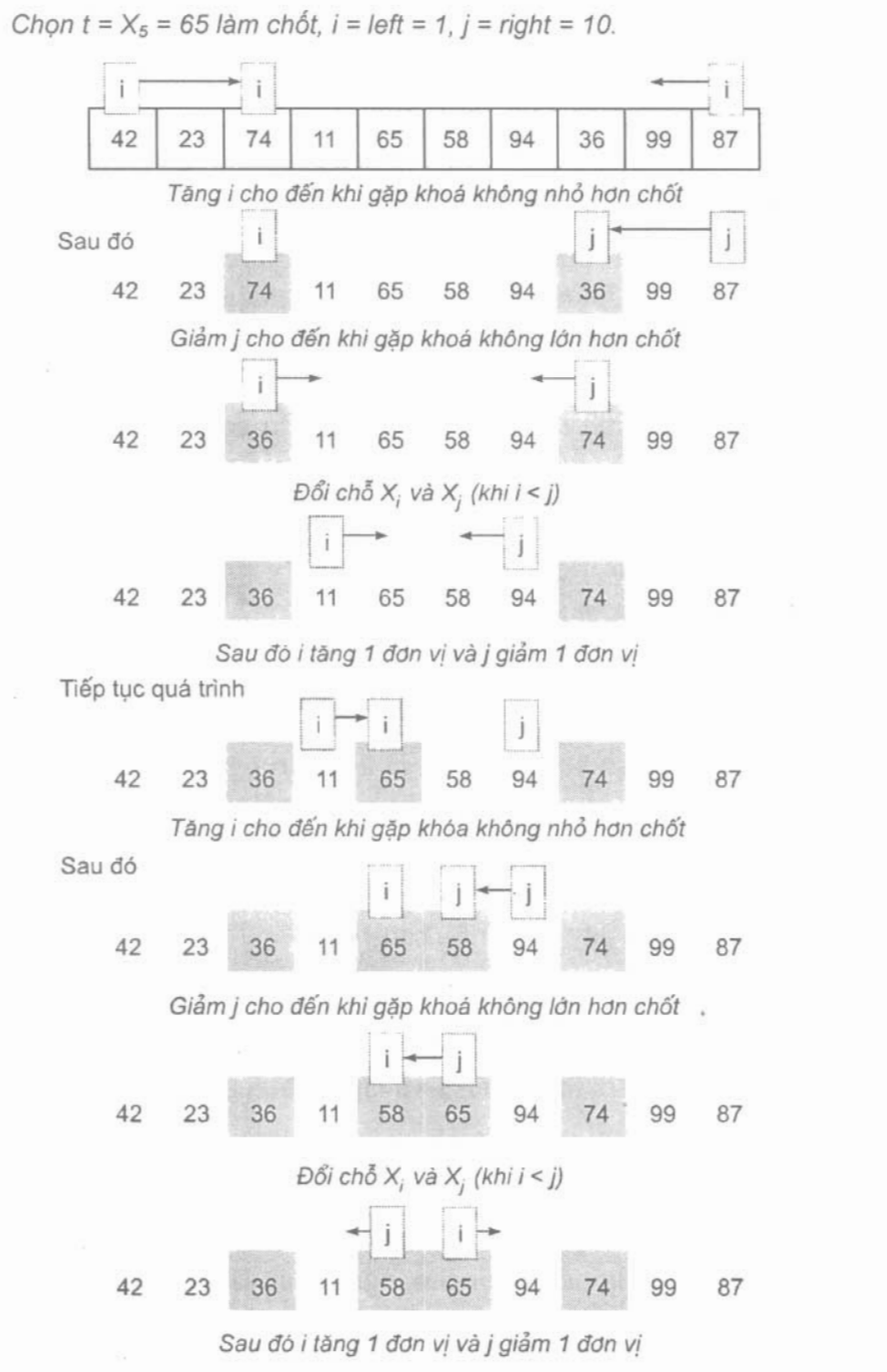
(2) Nếu $X_i < t$ thì tăng i lên 1 và lặp lại quá trình đó

Nếu $X_j > t$ thì j được giảm đi 1 và lặp lại quá trình đó

(3) Đổi chỗ X_i và X_j cho nhau (nếu $i < j$)

Quay lại bước (2) cho đến khi $i > j$ thì dãy X được chia thành 3 dãy nếu vị trí chốt không bị đổi chỗ, ngược lại dãy X được chia thành 2 dãy và kết thúc một lượt sắp xếp. Từ X_l đến X_j là dãy gồm các phần tử nhỏ hơn chốt, và từ X_i đến X_r là các phần tử lớn hơn chốt.

Diễn biến của lượt đầu được mô tả như sau:



Hình 3.1: Mô tả phương pháp sắp xếp phân đoạn

Đến đây kết thúc sắp xếp lượt một vì j < i.

Như vậy, sau một lượt duyệt, dãy khoá được chia thành hai phân đoạn, phân đoạn 1 từ X_{left} đến X_j gồm các phần tử nhỏ hơn chốt và phân đoạn 2 từ X_i đến X_{right} gồm các phần tử lớn hơn chốt.

Tiếp tục sắp xếp phân đoạn 1 và phân đoạn 2 cũng theo kỹ thuật như ở lượt đầu. Quá trình phân đoạn kết thúc khi mỗi phân đoạn chỉ còn một phần tử. Đến đây ta hoàn thành việc sắp xếp dãy.

Các phân đoạn được sắp với cùng một kỹ thuật, vì vậy khi cài đặt giải thuật sắp xếp người ta sử dụng kỹ thuật đệ quy.

1.3.3. Giải thuật

Dưới đây là giải thuật sắp xếp bằng phương pháp phân đoạn được viết tựa ngôn ngữ C, đây là một giải thuật được thiết kế theo kỹ thuật đệ quy.

```
void Quick_Sort (int X[ ], int left, int right)
/*Sắp xếp dãy khoá X với phân đoạn từ Xleft đến Xright */
{
  if ( left < right )
  //Việc phân đoạn chỉ thực hiện với phân đoạn có 2 phần tử trở lên
  {
        int i = left;
        int j = right;
        int k = (left+right)/2;
        int t = X[k];
        while (i <= j)
        {
              while (X[i] < t) i = i + 1;
              while (X[j] > t) j = j - 1;
              if (i <= j)
              {
                    int tg = X[i]; X[i] = X[j]; X[j] = tg;
```

```
                    i++; j--;
                }
            }
            Quick_Sort (X, left, j);
            Quick_Sort (X, i, right);
        }
    }
```

1.3.4. Đánh giá phương pháp

Trước hết ta hãy để ý đến một vài chi tiết có ảnh hưởng tới hiệu lực của phương pháp, đồng thời cũng thể hiện rõ đặc điểm của phương pháp này.

a. Vấn đề chọn "chốt":

Trong phần minh họa trên ta chọn chốt là phần tử trung tâm của phân đoạn cần sắp. Tuy nhiên, bạn có thể chọn phần tử bất kỳ trong phân đoạn để làm chốt. Nhưng rõ ràng khi thể hiện giải thuật ta phải định ra một cách thức chọn chốt cụ thể.

Vấn đề là chốt mà ta chọn được có tốt không? Nếu chốt ta chọn rơi vào đúng khoá nhỏ nhất (hoặc lớn nhất) của phân đoạn cần xử lý thì sau mỗi lượt ta chỉ tách ra được một phân đoạn con có kích thước nhỏ hơn trước là 1 phần tử (vì đã bớt đi một phần tử là "chốt") và phân đoạn này tiếp tục được xử lý. Như vậy ta đã quay trở lại phương pháp sắp xếp kiểu nổi bọt đơn giản. Việc chọn chốt như thế này đã dẫn đến tình huống xấu nhất của phương pháp.

Nếu gọi "*trung vị*" (median) của một dãy khoá là khoá sẽ đứng ở giữa dãy đó sau khi dãy đã được sắp xếp, nghĩa là nó lớn hơn một nửa số khoá của dãy và nhỏ hơn số còn lại, thì tốt nhất vẫn là chọn được đúng trung vị làm "chốt". Lúc đó sau mỗi lượt ta sẽ tách ra được hai phân đoạn con có độ dài gần như nhau và phân đoạn xử lý tiếp theo có kích thước chỉ bằng "nửa" phân đoạn đã chứa nó.

Nhưng làm sao có thể chọn được đúng trung vị? Nếu giả thiết: sự xuất hiện của các khoá trong dãy là đồng khả năng thì trung vị có

thể là bất kỳ một khoá nào trong dãy. Trong giải thuật trên, ta chọn khoá đứng đầu làm chốt là dựa trên cơ sở này. Nhưng với cách chọn này, nếu dãy khoá có khuynh hướng đã theo thứ tự sắp xếp thì khả năng xấu nhất lại xuất hiện. Tuy nhiên nếu khuynh hướng này hay xuất hiện thì việc chọn khoá đang được đứng ở giữa dãy lại gặp thuận lợi.

Để dung hoà với cách chọn như trên, đồng thời cũng để kết hợp với một đề nghị sau này của Hoare là: "Chọn trung vị của một dãy khoá nhỏ hơn, thuộc dãy khoá cho, làm chốt", R.C.Singleton đã đưa ra một cách chọn là:

Chọn X[q] là "chốt" với X[q] là trung vị của ba khoá X[1eft], X[(left+right)/2] và X[right] trong đó left và right là chỉ số của khoá đầu và khoá cuối của phân đoạn. Các kiểu chọn khoá chốt còn được nhiều tác giả khác nữa đưa ra và cũng có nhiều kết quả đáng chú ý.

b. Vấn đề phối hợp với cách sắp xếp khác

Khi kích thước của các phân đoạn đã khá nhỏ, việc tiếp tục phân đoạn nữa theo phương pháp QUICK_SORT thực ra sẽ không có lợi. Lúc đó sử dụng một số phương pháp sắp xếp đơn giản lại tiện hơn. Vì vậy, sắp xếp NHANH thường không tiến hành triệt để mà dừng lại ở lúc cần thiết để gọi tới một phương pháp sắp xếp đơn giản, giao cho nó tiếp tục thực hiện sắp xếp với các phân đoạn nhỏ còn lại. Kunth (1974) có nêu: 9 có thể coi là kích thước giới hạn của phân đoạn để sau đó QUICK_SORT gọi tới phương pháp sắp xếp đơn giản.

Ngoài ra việc chọn phân đoạn nào để xử lý tiếp theo cũng là một vấn đề cần được xem xét tới.

Bây giờ ta sẽ đánh giá giải thuật QUICK_SORT.

Gọi T(n) là thời gian thực hiện giải thuật ứng với một bảng n khoá, P(n) là thời gian để phân đoạn một bảng n khoá thành hai bảng con. Ta có thể viết:

$$T(n) = P(n) + T(j\text{-left}) + T(right\text{-}j)$$

Chú ý rằng $P(n) = Cn$ với C là hằng số.

Trường hợp xấu nhất xảy ra khi bảng khoá vốn đã có thứ tự sắp xếp: sau khi phân đoạn một trong hai bảng con là rỗng (j = left hoặc j = right).

Giả sử j = left, ta có:

$$\begin{aligned} T_x(n) &= P(n) + T_x(0) + T_x(n-1) \\ &= Cn + T_x(n-1) \\ &= Cn + C(n-1) + T_x(n-2) \\ &\cdots \\ &= \sum_{k=1}^{n} C_k + T_x(0) \\ &= C.\frac{n(n+1)}{2} = O(n^2) \end{aligned}$$

Trường hợp này QUICK_SORT không hơn gì các phương pháp đã nêu trước đây.

Trường hợp tốt nhất xảy ra khi bảng luôn luôn được chia đôi, nghĩa là j = (left+right)/2. Lúc đó:

$$\begin{aligned} T_1(n) &= P(n)b + 2T_1(n/2) \\ &= Cn + 2T_1(n/2) \\ &= Cn + 2C(n/2) + 4T_1(n/4) = 2Cn_n + 2^2T_1(n/4) \\ &= Cn + 2C(n/2) + 4C(n/4) + 8T_1(n/8) = 3C_n + 2^3T_1(n/8) \\ &\cdots \\ &= (\log_2 n)\, Cn + 2^{long}2^mT_1(1) \\ &= O(n\log_2 n) \end{aligned}$$

Việc xác định giá trị trung bình $T_{tb}(n)$ không còn đơn giản như hai trường hợp trên, nên ta sẽ không xét chi tiết. Kết quả mà ta cần ghi nhận là: người ta đã chứng minh được:

$$T_{tb}(n) = O(n\log_2 n)$$

Như vậy rõ ràng là khi n khá lớn, QUICK_SORT đã tỏ ra có hiệu lực hơn hẳn ba phương pháp đã nêu. Tuy nhiên, việc sử dụng kỹ thuật đệ qui sẽ tiêu tốn rất nhiều bộ nhớ khi thực hiện giải thuật trên máy tính.

Sau đây ta xem xét một phương pháp sắp xếp với thời gian cũng rất tốt đó là phương pháp vun đống.

1.4. Sắp xếp vun đống

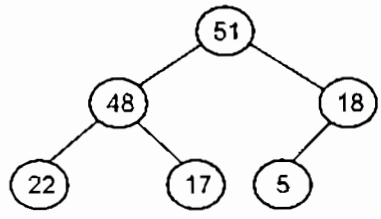
1.4.1. Giới thiệu phương pháp

Trước tiên, ta xem xét khái niệm đống (HEAP).

Đống là một cây nhị phân hoàn chỉnh mà mỗi nút được gán một giá trị khoá sao cho khoá ở nút "cha" bao giờ cũng lớn hơn khóa ở các nút "con".

Đống được lưu trữ kế tiếp trên máy (bởi mảng một chiều).

Như vậy đối với đống, nếu nút cha ở vị trí thứ i thì 2 nút con (nếu có) sẽ ở vị trí thứ 2*i và 2*i+1.



Hình 3.2: Đống

Ví dụ: đống là cây T có dạng như hình 2.2.

1.4.2. Nguyên tắc sắp xếp

Sắp xếp kiểu vun đống (HEAP_SORT) được chia thành hai giai đoạn:

- Giai đoạn tạo đống: Cây nhị phân biểu diễn dãy khoá ban đầu được biến đổi để tạo thành đống.

- Giai đoạn sắp xếp: ở giai đoạn này, ta tiến hành nhiều lượt hai phép xử lý sau:

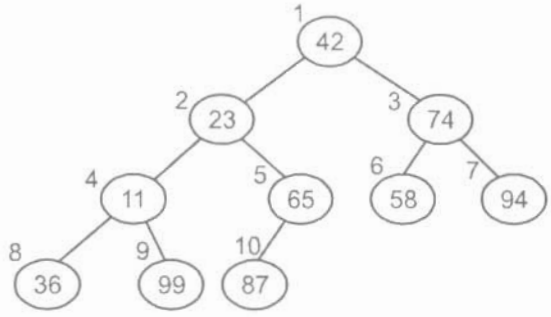
  + Đưa khóa trội về đúng vị trí, bằng cách thực hiện hoán vị các khoá.

  + Vun lại cây gồm các khoá còn lại (sau khi đã loại khoá trội) thành đống.

Ví dụ mô tả: Giả sử với dãy khoá X

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
42	23	74	11	65	58	94	36	99	87

Cây nhị phân biểu diễn dãy khoá ban đầu như hình 3.3(a).

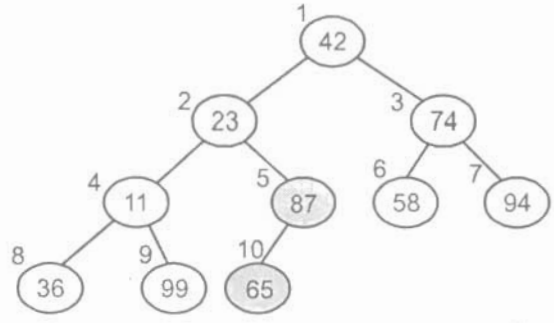


Hình 3.3(a): Cây nhị phân biểu diễn dãy khóa

Lưu ý, cây nhị phân hình 3.3(a) chỉ là mô hình giúp ta hình dung được quá trình sắp xếp theo phương pháp vun đống. Còn quá trình sắp xếp thực sự vẫn là việc hoán đổi vị trí trực tiếp các phần tử mảng.

Với cây này để nó trở thành đống ta làm như sau:

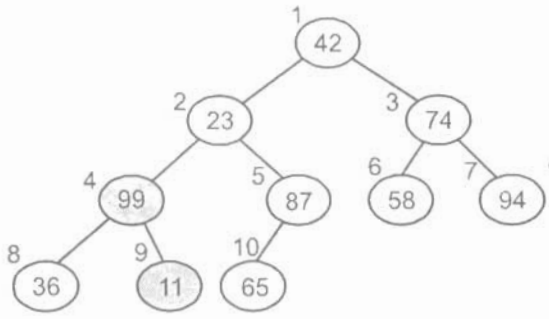
+ Cây con có nút gốc là 65 (vị trí thứ 5 trở thành đống, hình 3.3(b)).



Hình 3.3(b): Xử lý nút khóa X_5

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
42	23	74	11	87	58	94	36	99	65

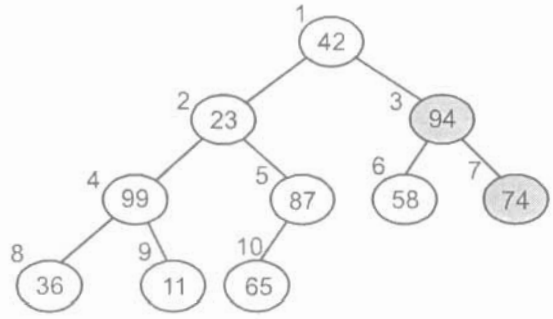
+ Cây con có nút gốc 11 (nút thứ 4) trở thành đống (hình 3.3(c)).



Hình 3.3(c): Xử lý nút khóa X_4

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
42	23	74	99	87	58	94	36	11	65

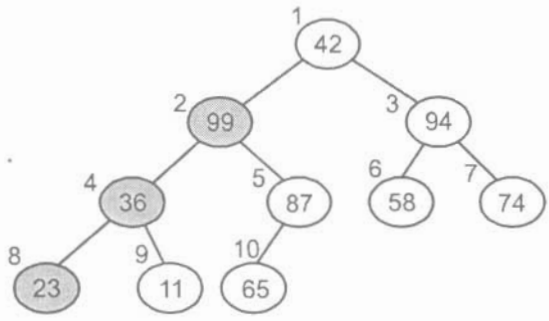
+ Cây con có nút gốc 74 (nút thứ 3) trở thành đống (hình 3.3(d)).



Hình 3.3(d): Xử lý nút khóa X_3

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
42	23	94	99	87	58	74	36	11	65

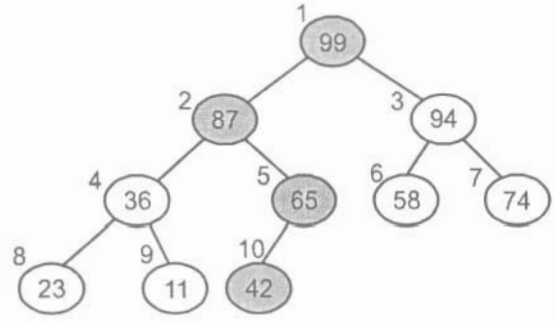
+ Cây con có nút gốc 23 trở thành đống (nút thứ 2) - hình 3.3(e).



Hình 3.3(e): Xử lý nút khóa X_2

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
42	99	94	36	87	58	74	23	11	65

+ Cuối cùng, cây con có nút gốc 42 trở thành đống (nút thứ 1) - hình 3.3(f).

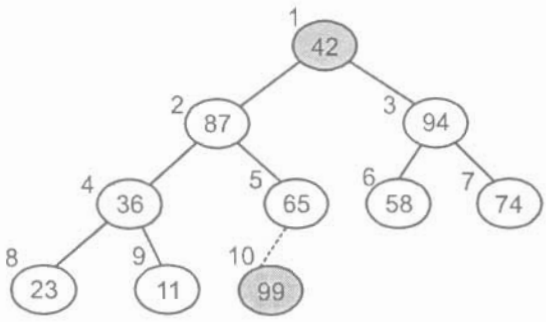


Hình 3.3(f): Xử lý nút khóa X_1 - Đống đầu tiên

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
99	87	94	36	65	58	74	23	11	42

Đến đây quá trình tạo đống đầu tiên kết thúc, ta thấy khóa trội được chuyển về về vị trí đầu tiên của mảng (nút gốc của cây-đống).

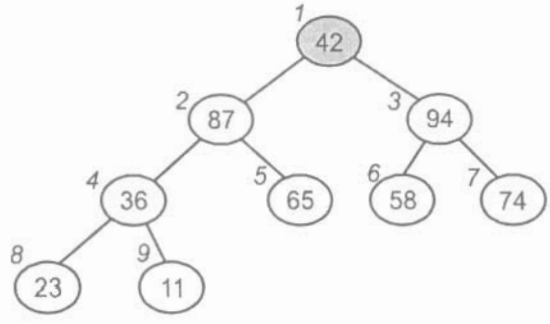
Sang giai đoạn 2, khoá trội 99 được chuyển đến vị trí cuối cùng bằng cách hoán vị cho khoá 42, sau khi hoán vị, nút ứng với khóa trội 99 coi như được loại khỏi cây - hình 3.4.



Hình 3.4: Hoán đổi nút đầu với nút cuối

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
42	87	94	36	65	58	74	23	11	99

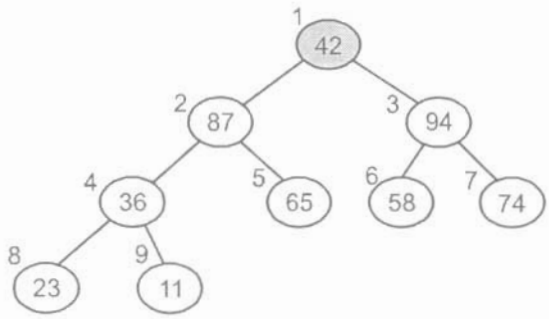
Cây còn lại được vun lại thành đống. Nó có dạng như hình 3.4(a).



Hình 3.4(a): Cây sau khi loại nút cuối

Nhận thấy rằng, cây này chưa phải là một đống, nhưng tất cả các cây con của cây này đều là đống. Vì vậy, ở bước này và tất cả các bước còn lại ta chỉ cần xét nút gốc (nút thứ 1) của cây.

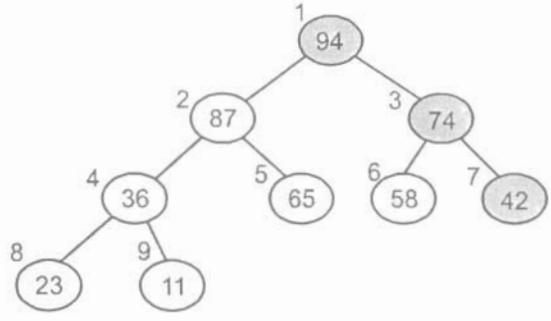
Ta có đống thứ 2 như hình 3.4(b).



Hình 3.4(b): Đống thứ 2

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
94	87	74	36	65	58	42	23	11	**99**

Lại đổi vị trí nút đầu tiên và nút cuối cùng - hình 3.4(c).



Hình 3.4(c): Đổi vị trí nút đầu và nút cuối

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
11	87	74	36	65	58	42	23	**94**	**99**

Sau đó một lượt xử lý tiếp theo được thực hiện tương tự và cứ như thế cho đến khi toàn bộ dãy khóa được sắp xếp. Bạn đọc có thể tự mình mô tả quá trình sắp xếp trong các bước tiếp theo và mô tả việc sắp xếp dãy khóa theo thứ tự giảm dần.

1.4.3. Giải thuật

Nhận thấy, có thể coi một nút lá là một cây con đã thoả mãn tính chất của đống, có nghĩa một nút lá được coi là một đống. Như vậy,

bước tạo đống hay vun đống được quy về một phép xử lý chung là: chuyển một cây thành đống mà cây con trái và cây con phải của gốc đã là đống rồi. Ta xây dựng riêng một thủ tục vun đống là thủ tục Hoan_vi.

Mặt khác, đối với cây nhị phân hoàn chỉnh có n nút, chỉ các nút ứng với các vị trí từ 1 đến (n/2) mới có thể là nút cha của các nút khác. Nên khi tạo đống thủ tục Hoan_vi chỉ cần áp dụng với các cây con có gốc ở vị trí (n/2), (n/2) - 1,..., 1. Còn khi vun đống thì luôn áp dụng với cây có gốc ở vị trí 1.

Giải thuật sắp xếp dãy n khoá X_1, X_2,..., X_n được biểu diễn bởi 3 thủ tục tựa ngôn ngữ lập trình C sau đây.

```
void Hoan_vi(X[ ],k,r)
{
  if (X[k] khác lá và có giá trị nhỏ hơn 2 con)
  {       + Chọn con có giá trị lớn hơn, giả sử là X[j];
          + Đổi chỗ X[k], và X[j] ;
          + call Hoan_vi(X,j,r);
  }
}
```

Thủ tục này thực hiện biến đổi cây có gốc là X[k] thành đống, với r là vị trí của khoá cuối cùng trong dãy được xét. Thủ tục này cũng có thể viết dưới dạng một giải thuật lặp để cải thiện thời gian sắp xếp. Bạn đọc dựa vào minh hoạ trên để "làm mịn" thủ tục này.

```
void Tao_dong_dau_tien(X[], n)
{
  for (i = n/2; i >= 1; i--)
          Hoan_vi(X, i, n);
}
```

Thủ tục này thực hiện biến đổi dãy ban đầu (n khoá) thành dãy biểu diễn đống đầu tiên. Quá trình biến đổi đống đầu tiên được thực

hiện từ dưới lên, các đống tiếp theo sẽ được thực hiện từ trên xuống (bạn đọc có thể xem lại phần minh họa).

```
void HEAP_SORT(X[ ], n)
{
Tao_dong_dau_tien(X,n);
  for (i = n; i >= 2; i--)
  {
        Đổi chỗ (X[1], X[i]);
        Hoan_vi(X, 1, i-1);
  }
}
```

Thủ tục này thực hiện việc sắp dãy khoá theo chiều tăng dần, sử dụng hai thủ tục bên trên. Tuy nhiên, thủ tục Hoan_vi được viết dưới dạng đệ qui, vì thế làm chậm quá trình sắp xếp và tốn bộ nhớ. Trường hợp này ta nên khử đệ quy, nghĩa là sử dụng giải thuật lặp, việc này khá đơn giản, bạn đọc tự nghiên cứu, xem như một bài tập.

1.4.4. Phân tích đánh giá

Đối với HEAP_SORT ta chú ý tới việc đánh giá trong trường hợp xấu nhất. Như ta đã biết: một cây nhị phân hoàn chỉnh có n nút thì chiều cao của cây đó là $[\log_2(n+1)]$. Khi tạo đống cũng như khi vun lại đống trong giai đoạn sắp xếp, trường hợp xấu nhất thì số lượng phép so sánh cũng chỉ tỷ lệ với chiều cao của cây. Do đó có thể suy ra, trong trường hợp xấu nhất, cấp độ lớn của thời gian thực hiện HEAP_SORT chỉ là $O(n\log_2 n)$. Việc đánh giá thời gian thực hiện trung bình phức tạp hơn, ta không xét tới mà chỉ ghi nhận một kết quả đã được chứng minh là: cấp độ lớn của thời gian thực hiện trung bình giải thuật HEAP_SORT cũng là $O(n\log_2 n)$.

Có thể nhận xét thêm là: QUICK_SORT còn phải dùng thêm không gian nhớ cho ngăm xếp (stack), để bảo lưu thông tin về các phân đoạn sẽ được xử lý tiếp theo (vì thực hiện đệ qui). Còn

HEAP_SORT thì ngoài một nút nhớ phụ, để thực hiện đổi chỗ, nó không cần thêm gì nữa.

1.5. Sắp xếp kiểu trộn

Bây giờ ta xét tới một phương pháp sắp xếp mới, khá đặc biệt ở chỗ nó dựa trên một phép toán rất đơn giản là phép trộn.

1.5.1. Phép trộn hai đường

Một tư tưởng hết sức đơn giản là trộn hai dãy đã được sắp xếp thành một dãy được sắp xếp.

Giả sử cho hai dãy được sắp xếp:

X: 12 25 28

và Y: 3 9 15 32 39

Khi đó ta sẽ trộn hai dãy X và Y thành dãy Z cũng được sắp xếp tăng như sau:

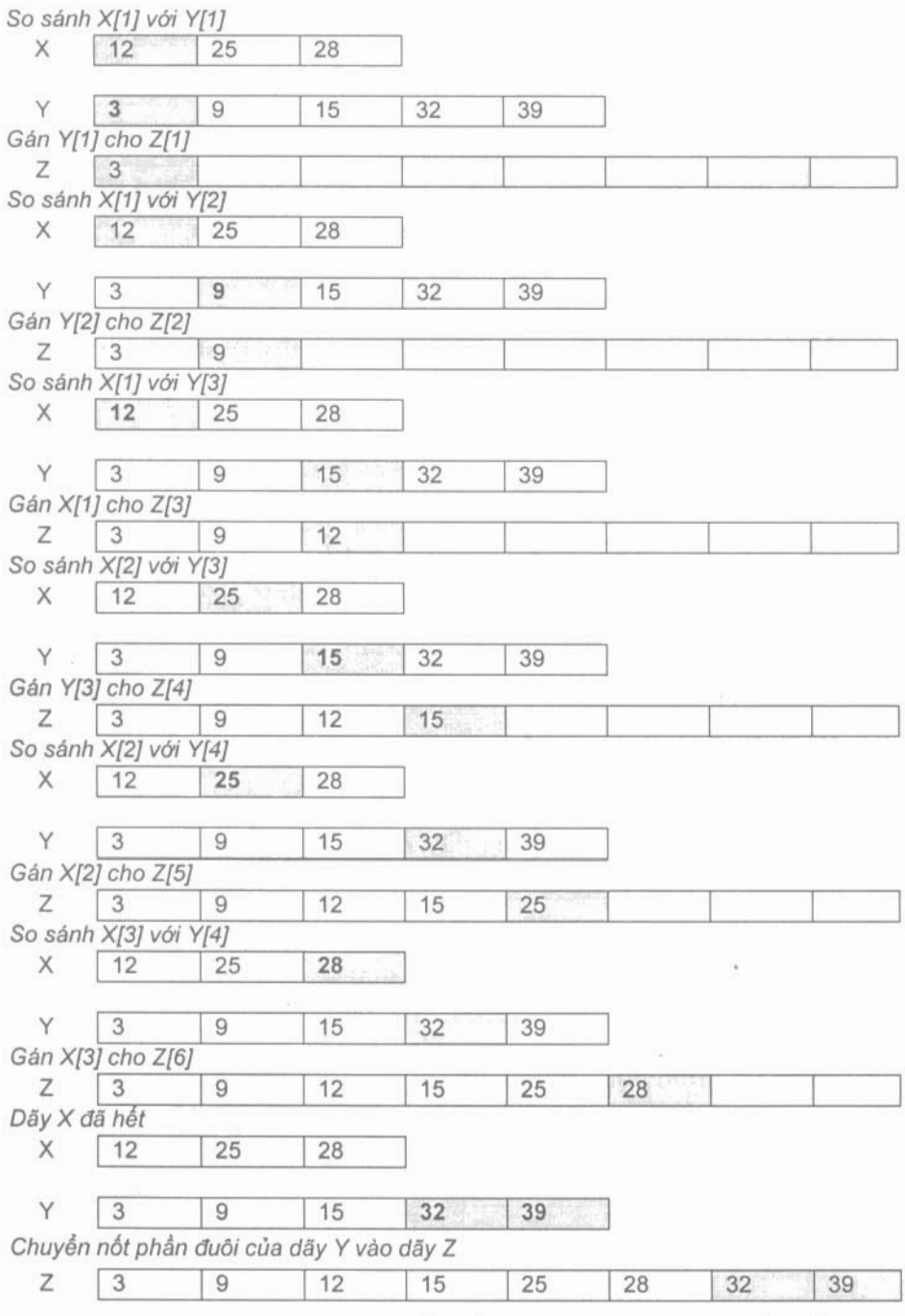
Z: 3 9 12 15 25 28 32 39

Có thể mô tả cách trộn đơn giản như sau:

+ So sánh hai khóa nhỏ nhất của hai dãy X và Y, chọn khóa nhỏ hơn đặt vào vị trí thích hợp trong dãy Z, rồi loại khóa được chọn khỏi dãy chứa nó.

+ Quá trình như vậy cứ tiếp tục cho đến khi một trong hai dãy X hoặc Y đã hết, khi đó chuyển nốt phần đuôi của dãy còn lại vào dãy Z là xong.

Hình 3.5 minh họa cách làm trên.



Hình 3.5: Trộn dãy X và dãy Y sắp xếp tăng, thành dãy Z sắp xếp tăng

Bây giờ ta xét tới giải thuật MERGING, các dãy X, Y, Z được lưu trữ bởi các mảng, với m, n lần lượt là độ dài của X và Y. Việc so sánh X[i] và Y[j] sẽ xác định phần tử của dãy nào được gán cho Z[k].

```
void MERGING(X[ ], m, Y[ ], n, Z)
{
  //1. Khởi tạo các chỉ số
        i = 1; j = 1; k = 1;
  //2. Chuyển các phần phần tử từ dãy X, Y vào dãy Z
        while (i <= m && j <= n)
        {
              if (X[i] < Y[j])
              {
                    Z[k] = X[i]; i++; k++;
              }
              else
              {
                    Z[k] = Y[j]; j++; k++;
              }
        }
  //3. Một trong hai mạch đã hết, chuyển đuôi của dãy còn lại vào Z
        while (i <= m)
        {
              Z[k] = X[i]; i++; k++;
        }
        while (j <= n)
        {
              Z[k] = Y[j]; j++; k++;
        }
}
```

1.5.2. Sắp xếp kiểu trộn hai đường trực tiếp

a. Giới thiệu phương pháp

Với ý tưởng trộn hai dãy đã sắp xếp thành một dãy đã sắp xếp người ta đã phát triển thành một phương pháp sắp xếp mới mà ta quen gọi là phương pháp trộn.

Giả sử cần sắp xếp dãy khóa X với độ dài n: X_1, X_2, ..., X_n. Có thể thấy rằng mỗi khóa X_i trong dãy có thể xem như một dãy con được sắp với độ dài là 1 (sau đây ta gọi là một vệt).

Nếu trộn hai vệt như vậy ta sẽ được một vệt mới với độ dài là 2. Lại trộn hai vệt độ dài 2 ta được một vệt độ dài 4, tiếp tục trộn như thế, cuối cùng ta sẽ được một vệt có chiều dài n, như thế dãy ban đầu được sắp xếp hoàn toàn.

**** Ta mô tả phương pháp vừa nêu trong ví dụ sau:***

Cho dãy khóa: 42 23 74 11 65 58 94 36 99 87

Coi dãy khóa gồm 10 mạch có độ dài 1

[42] [23] [74] [11] [65] [58] [94] [36] [99] [87]

Trộn các cặp vệt độ dài 1 kề nhau ta được:

[23 42] [11 74] [58 65] [36 94] [87 99]

Trộn các cặp vệt độ dài 2 kề nhau ta được:

[11 23 42 74] [36 58 65 94] [87 99]

Ta giữ nguyên vệt lẻ cặp: [87 99]

Trộn các cặp vệt độ dài 4 kề nhau ta được:

[11 23 36 42 58 65 74 94] [87 99]

Ta giữ nguyên vệt lẻ cặp: [87 99]

Cuối cùng, trộn hai vệt ta có dãy được sắp xếp:

[11 23 36 42 58 65 74 87 94 99]

Qua ví dụ trên bạn đọc đã hiểu được phương pháp sắp trộn, sau đây ta đi thiết kế giải thuật cho phương pháp này.

b. Giải thuật

Giải thuật gồm ba công đoạn:

+ Thủ tục trộn hai vệt thành một vệt

+ Thủ tục trộn một lượt các cặp vệt độ dài *l* (có thể một vệt có chiều dài nhỏ hơn *l*).

+ Thủ tục sắp xếp trộn

Ta cải tiến thủ tục MERGING thành thủ tục trộn hai vệt kề nhau trên dãy khóa X, vệt được trộn nằm trên dãy Z.

```
void MERGE (X[ ], bt1, w1, bt2, w2, Z[ ])
{
//bt1, bt2: là vị trí biên trái của hai vệt, w1, w2 là độ dài của hai vệt
      i = bt1; j = bt2; bp1 = bt1+w1-1; bp2 = bt2+w2-1; k = bt1;
//bp1, bp2 là biên phải của hai vệt, k là biên trái của vệt mới trên dãy Z
  while (i <= bp1 && j <= bp2)
  {
        if (X[i] < X[j])
        {
              Z[k] = X[i]; i++; k++;
        }
        else
        {
              Z[k] = X[j]; j++; k++;
        }
  }
  while (i <= bp1)
  {
        Z[k] = X[i]; i++; k++;
  }
  while (j <= bp2)
  {
        Z[k] = X[j]; j++; k++;
  }
}
```

**** Thủ tục trộn một lần các cặp vệt***

Với dãy khóa X độ dài n: X_1, X_2,..., X_n, các dãy con trong X là các vệt có độ dài *l*, trừ dãy con cuối cùng có thể có độ dài nhỏ hơn *l*.

Nếu *cv* là số cặp vệt có độ dài *l* thì *cv* = *n*/(2**l*)

Đặt s = 2**l***cv* thì s là số các phần tử thuộc các cặp vệt, và *r* = *n*-*s* là số các phần tử còn lại.

```
void MERGE_PASS (X[ ], n, l, Z[ ])
{
  //Z là dãy có độ dài n, chứa các phần tử của X sau khi trộn
  //1. Khởi tạo các giá trị ban đầu
        cv = n/(2*l);
        s = 2*l*cv;
        r = n – s;
  //2. Trộn từng cặp mạch
        for (j = 1; j <= cv; j++)
        {
              b1 = l + (2*j -2)*l ; //biên trái của mạch thứ nhất
              MERGE(X, b1, l, b1+l, Z);
        }
  //3. Chỉ còn một vệt
        if (r<=l)
        for (j=1; j<=r; j++)
              Z[s+j] = X[s+j];
//4. Còn hai vệt nhưng một vệt có độ dài nhỏ hơn l
        else MERGE(X, s+1, l, s+l+1, r-l, Z);
}
```

Thủ tục MERGE_PASS được gọi trong thủ tục thực hiện sắp xếp kiểu trộn hai đường trực tiếp sau đây:

```
void MERGE_SORT (X[ ], n)
{
  //1. Khởi tạo số phần từ trong một vệt
        l =1;
  //2. Sắp xếp trộn
```

```
        while (l<n)
            {
            //trộn và chuyển các phần tử vào dãy Z
                MERGE_PASS(X, n, l, Z);
            //trộn và chuyển các phần tử trở lại dãy X
                MERGE_PASS(X, n, 2*l, Z);
                l = l*4;
            }
}
```

Sau khi sắp xếp xong các phần tử của dãy khóa X vẫn ở chỗ cũ.

c. Phân tích đánh giá

Trong phương pháp này ta thấy, số lượng phép toán chuyển chỗ thường nhiều hơn số lượng phép toán so sánh. Chẳng hạn, với thủ tục *MERGE*, trong câu lệnh *while*, ứng với một phép so sánh có một phép đổi chỗ. Nhưng nếu một vệt nào đó hết trước, thì phần đuôi của vệt còn lại được chuyển chỗ mà không tương ứng với một phép so sánh nào. Vì vậy, với phương pháp này ta chọn phép toán tích cực là phép toán chuyển chỗ để đánh giá thời gian thực hiện của giải thuật.

Nhận thấy rằng, ở bất kỳ lượt trộn nào (MERGE_PASS) thì toàn bộ các khóa cũng được chuyển sang dãy mới (từ X sang Z, hoặc từ Z sang X). Như vậy chi phí thời gian cho một lượt trộn là O(n). Ngoài ra số lượt gọi thủ tục MERGE_PASS trong thủ tục MERGE_SORT là $[\log_2 n]$, vì ở lượt 1 kích thước của vệt là $l=1=2^0$. Ở lượt i kích thước của vệt sẽ là 2^{i-1}, mà sau lượt cuối cùng thì vệt đã có kích thước là n.

Vậy sắp xếp kiểu trộn hai đường trực tiếp có thời gian thực hiện là $O(n\log_2 n)$. Tuy nhiên, chi phí về không gian nhớ khá lớn, nó đòi hỏi 2n phần tử nhớ, gấp đôi so với các phương pháp khác. Do đó người ta thường sử dụng phương pháp này khi sắp xếp ngoài, sắp xếp dữ liệu trong tệp tin.

Kết luận chung:

Ta nhận thấy, với cùng một mục đích sắp xếp mà có rất nhiều phương pháp và kỹ thuật giải quyết khác nhau. Cấu trúc dữ liệu được

lựa chọn để mô tả đối tượng sắp xếp đã ảnh hưởng rất lớn tới việc lựa chọn giải thuật sắp xếp thích hợp. Các giải thuật sắp xếp đơn giản thể hiện ba kỹ thuật cơ sở của sắp xếp (dựa vào phép so sánh các giá trị khoá). Tuy nhiên, cấp độ lớn của thời gian thực hiện chúng là (n^2), do đó chỉ nên sử dụng chúng khi n nhỏ. Các giải thuật cải tiến như QUICK_SORT, HEAP_SORT đã đạt được thời gian thực hiện tốt hơn là $O(n\log_2 n)$, thường được sử dụng khi n lớn. Nếu dãy khoá cần sắp xếp vốn có khuynh hướng hầu như đã được sắp xếp sẵn rồi thì QUICK_SORT lại không nên dùng. Nhưng nếu ban đầu dãy có khuynh hướng ít nhiều có thứ tự ngược với thứ tự sắp xếp thì HEAP_SORT lại tỏ ra thuận lợi.

Việc khẳng định phương pháp sắp xếp này tốt hơn các phương pháp khác chỉ là tương đối, vì thế việc chọn một phương pháp sắp xếp thích hợp thường tuỳ thuộc vào từng yêu cầu, từng điều kiện cụ thể của bài toán.

2. TÌM KIẾM

2.1. Bài toán tìm kiếm

Tìm kiếm là một đòi hỏi thường xuyên trong việc xử lý các bài toán tin học, nhất là đối với các bài toán quản lý. Tuy nhiên, ở đây ta chỉ xét các giải thuật tìm kiếm một cách tổng quát, không liên quan đến mục đích xử lý cụ thể nào.

Ta mô tả bài toán tìm kiếm như sau:

Cho một bảng gồm n bản ghi R_1, R_2,..., R_n. Mỗi bản ghi R_i có một khoá tìm kiếm là R_i.key ($1 \le i \le n$). Hãy tìm bản ghi có giá trị khoá tương ứng là KH cho trước.

KH được gọi là khoá tìm kiếm.

Việc tìm kiếm sẽ hoàn thành khi có một trong hai tình huống sau đây xảy ra:

1. Tìm được bản ghi có giá trị khoá tương ứng bằng KH, lúc đó việc tìm kiếm là thành công.

2. Không tìm được bản ghi nào có khoá tương ứng bằng KH, lúc đó việc tìm kiếm là không thành công.

Cũng giống như việc sắp xếp, khoá của mỗi bản ghi chính là đặc điểm nhận biết của bản ghi đó trong tìm kiếm. Để đơn giản, trong các giải thuật ta coi nó là đại diện cho bản ghi đó và trong các ví dụ ta cũng chỉ nói tới khoá. Ta coi các khoá R_i.key ($1 \le i \le n$) là các số khác nhau. Trong chương này ta cũng chỉ xét các phương pháp tìm kiếm cơ bản, phổ dụng, đối với dữ liệu được lưu trữ ở bộ nhớ trong và được gọi là tìm kiếm trong. Có hai phương pháp tìm kiếm trong là tìm kiếm tuần tự và tìm kiếm nhị phân mà ta sẽ xét trong các phần sau đây.

2.2. Tìm kiếm tuần tự

2.2.1. Nguyên tắc tìm kiếm

Tìm kiếm tuần tự là phương pháp tìm kiếm rất đơn giản. Nguyên tắc tìm kiếm theo phương pháp này có thể tóm tắt như sau:

Bắt đầu từ bản ghi thứ nhất, lần lượt so sánh khoá tìm kiếm KH với các khoá tương ứng của các bản ghi trong bảng, cho đến khi tìm được một bản ghi mong muốn trong bảng (trường hợp tìm kiếm thành công, trả về vị trí của bản ghi tìm được) hoặc đã hết bảng mà không thấy (trường hợp tìm kiếm không thành công).

Ví dụ minh hoạ

Giả sử các khoá tương ứng của các bản ghi trong bảng có 6 bản ghi là dãy số:

38 12 -56 95 23 11 và khoá cần tìm là KH = 95, khi đó việc tìm kiếm được thực hiện như mô tả dưới đây:

Với i = 1 < n	R_1.key = 38 ≠ KH, chuyển sang so sánh bản ghi tiếp theo.
Với i = 2 < n	R_2.key = 12 ≠ KH, chuyển sang so sánh bản ghi tiếp theo.
Với i = 3 < n	R_3.key = -56 ≠ KH, chuyển sang so sánh bản ghi tiếp theo.
Với i = 4 < n	R_4.key = 95 = KH, kết thúc tìm kiếm và trả về vị trí tìm được là i = 4

Ví dụ trên đây minh hoạ cho một phép tìm kiếm thành công, bạn đọc tự lấy ví dụ minh hoạ cho phép tìm kiếm không thành công.

2.2.2. Giải thuật

Thủ tục dưới đây thể hiện giải thuật tìm kiếm tuần tự, tìm kiếm khoá KH trên một dãy khoá X, có n phần tử. Nếu có nó sẽ đưa ra chỉ số của khoá ấy, nếu không có nó đưa ra giá trị 0.

```
int Sequence_Search(X[ ], n, KH)
{
  //1. Khởi tạo
        i=1;
  //2  Tìm khoá trong dãy
        while (X[i] != KH && i <= n) i = i+1;
  //3. Kiểm tra và trả về kết quả tìm kiếm
        if (i <= n) return (i);
        else return (0);
}
```

Để "tiết kiệm" thời gian ta có thể cải tiến giải thuật trên bằng cách thêm vào một khoá phụ X_{n+1}, có giá trị bằng KH như sau:

```
int Improvement_Sequence_Search(X[ ], n, KH)
{
  //1. Khởi tạo
        i = 1; X[n+1] = KH;
  //2. Tìm khoá trong dãy
        while (X[i] != KH)
              i = i+1;
  //3. Kiểm tra và trả về kết quả tìm kiếm
        if (i == n+1) return (0);
        else return (i);
}
```

Với giải thuật cải tiến, ta giảm được một nửa số phép toán kiểm tra trong vòng lặp *while*, vì thế tiết kiệm được khá nhiều thời gian khi số lượng bản ghi trong dãy X thật sự lớn.

2.2.3. Đánh giá hiệu lực của giải thuật

Để đánh giá hiệu lực của phép tìm kiếm ta có thể dựa vào số lượng các phép so sánh. Ta thấy với giải thuật trên (cải tiến). trong trường hợp tốt nhất, chỉ cần 1 phép so sánh, $C_{min} = 1$, còn trong trường hợp xấu nhất số phép toán so sánh là $C_{max} = n+1$. Nếu hiện tượng khoá tìm kiếm trùng với một khoá nào đó của bảng là đồng khả năng thì $C_{tb} = (n+1)/2$. Tóm lại, cả hai trường hợp xấu nhất cũng như trung bình, thời gian thực hiện tìm kiếm theo giải thuật trên là O(n).

Trong trường hợp dãy khoá đã được sắp xếp theo chiều tăng dần (hoặc giảm dần), thời gian trung bình thực hiện giải thuật sẽ nhỏ hơn. Tuy nhiên, trong trường hợp này ta có thể áp dụng phương pháp tìm kiếm khác, với thời gian nhanh hơn nhiều, đó là phương pháp tìm kiếm nhị phân.

2.3. Phương pháp tìm kiếm nhị phân

2.3.1. Nguyên tắc

Phương pháp tìm kiếm nhị phân là phương pháp tìm kiếm khá thông dụng. Trong phương pháp này ta áp dụng chiến lược "chia để trị" để giải quyết bài toán tìm kiếm. Việc này cũng giống như việc ta tìm một từ trong quyển từ điển. Ta có thể hiểu nôm na thế này: để tìm từ, ta mở từ điển vào trang "giữa", tìm từ trong trang này, nếu có, việc tìm kiếm là thành công, còn nếu không có ta tìm ở "nửa trước" hoặc nửa sau của từ điển. Việc tìm ở "nửa trước" hay "nửa sau" cũng được thực hiện giống như trên (nghĩa là ta cũng mở vào trang giữa...). Việc tìm kiếm dừng lại khi tìm được từ trong một trang "giữa" nào đó (tìm kiếm thành công), hoặc khi từ điển chỉ còn một trang, mà không có từ cần tìm (tìm kiếm không thành công).

Như vậy, trong phương pháp tìm kiếm nhị phân, giả sử với dãy khoá là $X_l, X_{l+1},..., X_r$ nó luôn chọn khoá ở giữa là X_j, với $j = [(l+r)/2]$ để so sánh với khoá tìm kiếm KH. Tìm kiếm sẽ kết thúc nếu $KH = X_j$. Nếu $KH < X_j$ tìm kiếm được thực hiện tiếp với $X_{l+1}, X_{l+2},..., X_{j-1}$, còn $KH > X_j$, tìm kiếm được thực hiện tiếp với $X_{j+1}, X_{j+2},..., X_r$. Với dãy

khoá sau, một kỹ thuật tương tự lại được sử dụng. Quá trình tìm kiếm được tiếp tục khi tìm thấy khoá mong muốn hoặc dãy khoá xét đó là rỗng (không thấy).

Ví dụ minh hoạ

Giả sử các khoá tương ứng của các bản ghi trong bảng có 6 bản ghi là dãy số tăng dần:

-56 11 12 23 38 95 và khoá cần tìm là KH = 95, khi đó việc tìm kiếm được thực hiện như mô tả dưới đây.

X_1 X_2 X_3 X_4 X_5 X_6

[-56 11 **12** 23 38 95]

Bước 1: j = 3, khoá ở giữa là $X_3 = 12$, $KH > X_3$

Tìm kiếm tiếp tục với dãy khoá

[23 **38** 95]

Bước 2: j = 5, khoá ở giữa là $X_5 = 38$, $KH > X_5$

Tìm kiếm tiếp tục với dãy khoá

[95]

Bước 3: j = 6, khoá ở giữa là $X_6 = 95$, $KH = X_5 \Rightarrow$ Tìm kiếm thành công!

Ví dụ minh hoạ tìm kiếm không thành công, bạn đọc xem như là một bài tập.

Sau đây là giải thuật tìm kiếm nhị phân.

2.3.2. Giải thuật

Giải thuật được viết dưới dạng một thủ tục dạng đệ quy, tựa ngôn ngữ C. Nếu tìm thấy, giải thuật trả về vị trí của khoá được tìm thấy (giá trị j). Nếu không tìm thấy, giải thuật trả về giá trị 0.

```
int BINARY_SEARCH(X[ ], l, r, KH)
{
//1. Trường hợp dãy được xét rỗng, tìm kiếm không thành công
  if (l > r)
        return 0;
```

```
//2. Tìm kiếm trong dãy được xét
  else {
        j = (l+r) % 2;
  //2.1. Tìm kiếm thành công
        if (KH == X[j])  return j;
  //2.2. Tìm ở nửa dãy trái
        else
              if (KH < X[j]) return BINARY_SEARCH(X,l,j-1,KH);
  //2.3. Tìm ở nửa dãy phải
              else    return BINARY_SEARCH(X,j+1,r,KH);
  }
}
```

Trong giải thuật trên *l*, r tương ứng là chỉ số của khoá đầu tiên và chỉ số của khoá cuối cùng của dãy đang xét.

Giải thuật tìm kiếm nhị phân cũng có thể được viết dưới dạng lặp (khử đệ quy), việc này bạn đọc có thể tự làm.

2.3.3. Đánh giá giải thuật

Trong giải thuật trên ta thấy số lượng phép so sánh phụ thuộc vào giá trị khoá KH. Trường hợp thuận lợi nhất đối với dãy khoá X_1, X_2,..., X_n, mà lời gọi sẽ là BINARY_SEARCH(X,1,n,KH) là KH = X[(n+1)/2], nghĩa là chỉ cần một phép so sánh, lúc đó $T_1 = O(1)$. Trường hợp xấu nhất có phức tạp hơn. Ta gọi w(r-*l*+1) là hàm biểu thị số lượng phép so sánh trong trường hợp xấu nhất ứng với một lời gọi BINARY_SEARCH(X,*l*,r,KH) và đặt n = r-*l*+1 (ứng với dãy khoá mà *l* = 1, r = n) thì trong trường hợp xấu nhất ta sẽ phải gọi đệ quy, vậy ta có:

$$w(n) = 1+w(n \text{ div } 2)$$

Với phương pháp truy hồi có thể viết:

$$w(n) = 1+1+w(n \text{ div } 2^2) \Rightarrow w(n) = 1+1+1+w(n \text{ div } 2^3)$$

Như vậy w(n) có dạng $w(n) = k+w(n \text{ div } 2^k)$

Khi $(n \text{ div } 2^k) = 1$ ta có $w(n \text{ div } 2^k) = w(1)$ và khi đó tìm kiếm phải kết thúc. Song $(n \text{ div } 2^k) = 1$ thì suy ra $2^k \le n \le 2^{k+1}$, do đó

$k \le \log_2 n < k+1$, nghĩa là có thể viết $k = \log_2 n$. Vì vậy, cuối cùng ta có $w(n) = \log_2 n + 1$ hay $T_X(n) = O(\log_2 n)$.

Người ta cũng chứng minh được $T_{tb}(n) = O(\log_2 n)$.

Rõ ràng so với tìm kiếm tuần tự (có thời gian trung bình là O(n)), chi phí tìm kiếm nhị phân ít hơn nhiều. Hiện tại không có phương pháp tìm kiếm nào, dựa trên việc so sánh giá trị khoá lại có thể đạt được kết quả tốt hơn.

Tuy nhiên, nên nhớ rằng tìm kiếm nhị phân chỉ thực hiện trên dãy khoá đã được sắp xếp, nên trong tìm kiếm cũng phải tính đến chi phí cho việc sắp xếp. Nếu dãy khoá luôn biến động, thì chi phí cho việc sắp xếp sẽ lớn, và đó là một trở ngại lớn với phương pháp tìm kiếm này.

Chương này đã giới thiệu với bạn đọc một số phương pháp sắp xếp và tìm kiếm khá thông dụng, cùng với ưu và nhược điểm của mỗi phương pháp. Mong rằng bạn đọc sẽ tìm hiểu thật kỹ và thành công trong việc ứng dụng chúng trong các bài toán tin học mà mình sẽ triển khai.

BÀI TẬP CHƯƠNG 3

1. Cho dãy khóa X: 50 8 34 6 98 17 83 25 66 42 21 59 63 71 85

   a. Hãy minh họa ba phương pháp sắp xếp đơn giản đã nêu, qua dãy khóa X theo thứ tự tăng dần, giảm dần.

   b. Tính số lượng phép toán đổi chỗ trong mỗi trường hợp, đưa ra nhận xét.

   c. Minh họa các phương pháp sắp xếp phân đoạn, vun đống và trộn qua dãy khóa X theo thứ tự tăng dần, giảm dần.

d. Minh họa phương pháp tìm kiếm tuần tự khóa k1 = 98, k2 = 63, k3 = 44 trên dãy khóa X và tính số phép toán so sánh trong mỗi trường hợp.

e. Với dãy khóa X được sắp xếp theo chiều tăng dần, hãy minh họa việc tìm kiếm các khóa k1, k2, k3 trên dãy X theo phương pháp tìm kiếm nhị phân.

f. Với dãy khóa X được sắp xếp theo chiều giảm dần, hãy minh họa việc tìm kiếm các khóa k1, k2, k3 trên dãy X theo phương pháp tìm kiếm nhị phân.

2. Hãy viết lại giải thuật của ba phương pháp sắp xếp đơn giản và "chạy chậm" các giải thuật này khi chúng được sử dụng để sắp xếp dãy khóa X (bạn đọc cũng nên tự lấy thêm các dãy khóa khác để minh họa).

3. Viết lại các giải thuật của các phương pháp sắp xếp phân đoạn, vun đống và trộn. "Chạy chậm" các giải thuật này trên dãy khóa X.

4. Viết chương trình ứng dụng thực hiện các yêu cầu sau:

- Nhập một dãy X, với n số từng đôi một khác nhau.
- Nhập số k, bằng phương pháp tìm kiếm tuần tự, hãy cho biết số k có trong dãy X hay không, nếu có cho biết vị trí của nó.
- Sắp xếp dãy theo chiều tăng dần bằng phương pháp lựa chọn/nổi bọt/thêm dần/phân đoạn/vun đống/trộn.
- Hiển thị dãy vừa sắp xếp ra màn hình.
- Nhập số m, bằng phương pháp tìm kiếm nhị phân, hãy cho biết số m có trong dãy X (vừa sắp) hay không, nếu có cho biết vị trí của nó.
- Sắp xếp dãy theo chiều giảm dần bằng phương pháp lựa chọn/nổi bọt/thêm dần/phân đoạn/vun đống/trộn.
- Hiển thị dãy vừa sắp xếp ra màn hình.

- Nhập số t, bằng phương pháp tìm kiếm nhị phân, hãy cho biết số t có trong dãy X (vừa sắp) hay không, nếu có cho biết vị trí của nó.

5. Viết chương trình

- Tạo một danh sách sinh viên, mỗi sinh viên gồm các thông tin: Mã sinh viên, họ và tên, năm sinh, giới tính và điểm tổng kết.
- Hiển thị danh sách ra màn hình sao cho điểm trung bình của sinh viên theo thứ tự giảm dần (sử dụng một trong các phương pháp sắp xếp phân đoạn/vun đống/trộn).
- Hiển thị danh sách ra màn hình sao cho tên của sinh viên theo thứ tự trong bảng chữ cái (sử dụng một trong ba phương pháp sắp xếp đơn giản).
- Nhập vào mã của một sinh viên, bằng phương pháp tìm kiếm tuần tự/nhị phân hãy cho biết sinh viên có mã vừa nhập có trong danh sách không, nếu có hãy cho biết vị trí của nó trong danh sách.