

Chương 1

Tổng quan về cấu trúc dữ liệu và giải thuật

1. GIẢI THUẬT VÀ CẤU TRÚC DỮ LIỆU

1.1 Khái niệm giải thuật:

Giải thuật là một dãy các câu lệnh chặt chẽ và rõ ràng xác định một trình tự các thao tác trên một số đối tượng nào đó sao cho sau một số hữu hạn bước thực hiện, ta đạt được kết quả mong muốn.

1.2 Khái niệm cấu trúc dữ liệu:

Dữ liệu trong các bài toán bao gồm một tập các phần tử cụ thể, mà ta gọi là dữ liệu nguyên tử. Nó có thể là một chữ số, một ký tự, . . . , nhưng cũng có thể là một con số, hay một tập . . . đối tượng tùy thuộc vào từng bài toán.

Trên cụ thể những dữ liệu nguyên tử, các cung cách khác nhau theo đó liên kết chúng lại với nhau sẽ cho ta các cấu trúc dữ liệu khác nhau.

Việc lựa chọn một cấu trúc dữ liệu hợp lý để tổ chức dữ liệu vào và trên cụ thể đó xây dựng đối tượng giải thuật hợp lý hữu hiệu để đạt được kết quả mong muốn cho bài toán, đó là một khâu hết sức quan trọng.

1.3 Vai trò của cấu trúc dữ liệu trong một đề án tin học:

Thực hiện một đề án tin học là chuyển bài toán thực tế thành bài toán có thể giải quyết trên máy tính. Mọi 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 đúng các 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**

Tổng hợp yêu cầu xử lý trong thực tế, cần tìm ra các giải pháp tổng hợp để giải quyết yêu cầu, 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 một bài toán trên máy tính, ta thường có khuynh hướng chỉ chú trọng đến việc xây dựng các giải thuật mà quên đi tầm quan trọng của việc tổ chức dữ liệu trong bài toán. Giải thuật phản ánh các phép xử lý, còn dữ liệu tổng hợp xử lý của giải thuật lại là dữ liệu. Chính dữ liệu của dòng các thông tin cần thiết để thực hiện giải thuật. Để xác định đặc điểm giải thuật phù hợp cần ta cần phải biết nó tác động đến loại dữ liệu nào và khi nào cần cấu trúc dữ liệu cũng cần phải hiểu rõ những thao tác nào tác động lên dữ liệu đó. Như vậy trong một đề án tin học, cấu trúc dữ liệu và giải thuật có mối quan hệ chặt chẽ với nhau, được thể hiện qua công thức

$$\text{Cấu trúc dữ liệu} + \text{Giải thuật} = \text{Chương trình}$$

Với một cấu trúc dữ liệu đã chọn, sẽ có những giải thuật tổng hợp hợp, phù hợp. Khi cấu trúc dữ liệu thay đổi thì những giải thuật cũng phải thay đổi theo để tránh việc xử lý gặp những ép, thì ưu tiên 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 được gìn và dữ liệu 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 tổng hợp 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 phần tử như sau:

Phần tử 1: Số điểm môn học chi:

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

```
typedef int Mang[12];
```

```
Mang a;
```

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

7	9	7	5	5	4	2	7	8	9	6	7
SV1				SV2				SV3			

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:

$$\text{Bảng điểm (dòng } i, \text{ cột } j) \Rightarrow a[(i) * \text{số cột} + j + 1]$$

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:

$$a[i] \Rightarrow \text{bảng điểm (dòng}(i / \text{số cột} + 1, \text{ cột}(i \% \text{số cột}))$$

Với phần tử này, thao tác xử lý được cài đặt như sau

```
Void xuất(mang a);
```

```
{
```

```
Int i, mon, so_mon;
```

```
so_mon = 4;
```

```
for ( i = 0; i < 12; i++)
```

```
{
```

```

        sv = i / so_mon + 1;

        mon = i % so_mon + 1;

        printf("Điểm môn: %d", mon, "cà sinh viên %d", sv, " là: %d", a[i]);

    }

}

```

Phân công án 2: sđ dng mng hai chiu

Khai báo mng hai chi u a có kích thc 3 dòng * 4 c t nh sau

```

typedef int mang[3][4];

```

Mang a;

	C t 1	C t 2	C t3	C t 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 sđ môn j của sinh viên i là phần tử t i dòng i c t j trong bng- cũng chính là phần tử i dòng i c t j trong mng.

Bngđm (dòng i, c t j) $\Rightarrow$ a[i][j]

V i phân công án này, thao tác x lý đ c cài đt nh sau:

```

Void xuat(mang a)

```

```

{

    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++)

```

```

        printf("Điểm môn:%d", i+1,"cà sinh viên%", j+1,"
là:%d",
        a[i][j]);

    }

}

```

Nhận xét:

Có thể thấy rõ phôi ng án 2 cung cấp một cấu trúc lưu trữ phù hợp với dữ liệu thực tế hơn phôi ng án 1, và do vậy giải thuật xử lý trên cấu trúc dữ liệu của phôi ng 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 phôi ng án thích hợp cho dữ liệu. Một cấu trúc dữ liệu tốt phải thoả mãn các tiêu chuẩn sau:

*** Phôi n 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 trừu 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 diễn biến thực tế.

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

- Chọn biến số nguyên **“integer”** để lưu trữ tỉn thối ng bán hàng (để tính theo công thức tỉn thối ng bán hàng = trữ giá hàng *5%), do vậy khi làm tròn mỗi giá trữ tỉn thối ng sẽ gây thất hối cho nhân viên bán hàng. Trôi ng hợp này phải sử dụng biến số thực để phản ánh đúng kết quả của công thức tính thực tế.

- Trong trôi ng phôi thông trung hối, mỗi lớp chọn có thể nhận tối đa 25 hối sinh. Lớp hiện có 20 hối sinh, mỗi tháng, mỗi hối sinh phải đóng lệ phí đoàn viên là 1.500 đồng. Chọn một biến số nguyên (khối năng -32768 ÷ 32767) để lưu trữ trữ số tỉn lệ phí đoàn của lớp hối trong tháng, nếu xảy ra trôi ng hợp có thêm 5 hối sinh nộp vào lớp thì giá trữ trữ số tỉn sẽ là

37500 đồng, vượt khả năng lưu trữ của bên đã chọn, gây ra tình trạng tràn số và sai lệch.

*** Phù hợp với các thao tác trên đó:** Tiêu chuẩn này giúp tăng hiệu quả của đề án, việc phát triển các thuật toán đơn giản, tự nhiên hơn và chi phí công trình đạt hiệu quả cao hơn với các đề xuất lý.

*** Tiết kiệm tài nguyên hệ thống:** Cấu trúc dữ liệu chọn nên sử dụng tài nguyên và để để đảm bảo nhiệm vụ của chức năng của nó. Thông thường có hai loại tài nguyên cần lưu ý nhất là bộ nhớ và thời gian. Tiêu chuẩn này nên cân nhắc tùy vào tình huống cụ thể khi thực hiện đề án. Nếu thực sự dùng đề án cần có những xử lý nhanh thì khi chọn cấu trúc dữ liệu yêu cầu tiết kiệm thời gian xử lý phải đạt năng hơn tiêu chuẩn sử dụng tài đa bộ nhớ, và ngược lại.

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

- Sử dụng biến “integer” (2 bytes) để lưu trữ một giá trị cho biến tháng hiện hành. Trong tình huống này ta chỉ cần sử dụng biến kiểu “byte” 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 thực tế 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, chọn học danh sách mở rộng.

3. CÁC CẤU TRÚC DỮ LIỆU CƠ SỞ

Máy tính thực sự chỉ có thể lưu trữ dữ liệu ở dạng nhị phân thông số. Nếu muốn phân ánh đúng các dữ liệu thực tế vào rồi đã dùng và phong phú, cần phải xây dựng những phép ánh xạ, những quy tắc để chuyển các tập hợp lên từng dữ liệu thô, nhằm đưa ra những khái niệm logic và hình thức lưu trữ khác nhau thông qua để 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 x lý có thể thi hành trên đối tượng kiểu T.

Ví dụ:

- Kiểu dữ liệu **Ký tự alphabet** = $\langle V_c, O_c \rangle$ với

$$V_c = \{a - z, A - Z\}$$

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

- Kiểu dữ liệu **Số nguyên** = $\langle V_i, O_i \rangle$

$$V_i = \{-32768 \div 32767\}$$

$$O_i = \{+, -, *, /, \%, \text{các phép so sánh, các phép toán luận lý nh phân}\}$$

Như vậy, mục đích của việc mô tả kiểu dữ liệu của ngôn ngữ lập trình có thể nội dung dữ liệu được phép lưu trữ và các x 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 thống ngôn ngữ lập trình sẽ có một số kiểu dữ liệu được gọi là *kiểu dữ liệu đơn* hay *kiểu dữ liệu phân tử* (atomic).

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

- Kiểu có thể rị rịc : số nguyên, ký tự, logic, liệt kê, miền con
- Kiểu không rị rịc : chuỗi ký tự, mảng,

Tùy từng ngôn ngữ lập trình, các kiểu dữ liệu định nghĩa sẵn này có thể khác nhau đôi chút. Chẳng hạn, với ngôn ngữ C, các kiểu dữ liệu này chỉ gồm

số nguyên, số thực, ký tự. Và theo quan điểm của C, kiểu ký tự thực chất cũng là kiểu số nguyên với một 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 C như là các giá trị nguyên khác 0 và bằng 0. Trong khi đó Pascal định nghĩa tất cả các kiểu dữ liệu đã liệt kê trên và phân biệt chúng một cách chặt chẽ.

Hệ kiểu của C được cho trong bảng sau:

Kiểu DL	Tên kiểu	Phạm vi	Kích thước	Ghi chú
Nguyên	int	-32768->+32767	2 byte	
	Unsigned int	0 -> 65535	2 byte	
	Long	-2147483648->2147483647	4 byte	
	Unsigned long	0 -> 4294967295	4 byte	
Kiểu ký tự	Char(signed char)	-128->+127	1 byte	
	Unsigned char	0 -> 255		
	float	3.4E-38->3.4E+38	4 byte	
Dữ liệu phẩy động	Double	1.7E -308-> 1.7E+308	8 byte	
	Long double	3.4E-4932-> 1.1E+4932	10 byte	

Chú ý: Tất cả các kiểu dữ liệu đều là các kiểu có thể, tức là với hai giá trị bất kỳ a và b thuộc cùng một kiểu, ta luôn có $a \leq b$ hoặc $a \geq b$. Các kiểu còn lại đều là kiểu đếm được, trừ kiểu real.

3.4. Các kiểu dữ liệu có cấu trúc

Khi giải quyết các bài toán phức tạp, ta 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 nào đó các *dữ liệu phân tử*, các thành phần này móc nối với nhau bởi một phương pháp nào đó. Đa số các ngôn ngữ lập trình đều cài đặt

số n mảng số kiểu có cấu trúc có bản nhúng mảng, chuỗi, tệp tin, bản ghi...và cung cấp công cụ cho lập trình viên để định nghĩa kiểu dữ liệu mới.

3.4.1. Mảng một chiều

Trong C và trong nhiều ngôn ngữ thông dụng khác có một cách để n gọn nhất để tạo để móc nối các dữ kiện trong một tệp hồ sơ, đó là cách sắp xếp các dữ kiện đó thành một dãy. Để lưu trữ dãy dữ kiện 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, mảng số thực, mảng các bản ghi, ...).

Trong 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

3.4.2. Chuỗi

Trong 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 chuỗi trong thư viện `stdio.h` như `scanf()`, `gets()`, `printf()`, `puts()`.

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

3.4.3. Mảng nhiều chiều

Mảng nhiều chiều được sử dụng nhiều nhất là mảng 2 chiều, có thể hình dung mảng 2 chiều giống như một bảng gồm các dòng và các cột, chúng hiển, bảng ghi nhật để 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
Hà Nội Phòng	27,5	26,5	26	27,5	27
Hà Long	25	27	26	28	27

Câu 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. Chiều 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, các số thực.

3.4.4. Câu trúc

Câu 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 câu trúc. Các câu trúc có thể được sử dụng để tạo nên các kiểu dữ liệu khác, chẳng hạn như mảng câu trúc.

Giả sử $T_1, T_2, \dots, T_n$ là các kiểu đã cho, và $F_1, F_2, \dots, F_n$ 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 $i = 1, 2, \dots, n$.

struct ST

```
{
    T1    F1 ;
    T2    F2 ;
    ...
    Tn    Fn ;
};
```

struct ST s1, s2, d[20] ;

s1, s2 là các biến cấu trúc, d là mảng 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 $(t_1, t_2, \dots, t_k)$, trong đó $t_i \in T_i$ ($i = 1, 2, \dots, k$).

3.4.5. Kiểu con trỏ

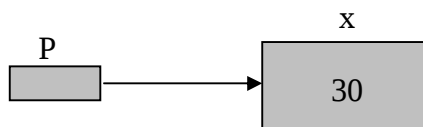
Một phép quan trọng nữa để liên hệ các cấu trúc dữ liệu, đó là số dòng con trỏ.

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 liên 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, thực, mảng, chuỗi... dữ liệu được lưu ở bộ nhớ trong, vì thế khi chương trình kết thúc, dữ liệu cũng bị

xóa. Để khc phc trng hp này, d liu cn đc l u b nh ngoài, đó là các t p tin.

- Theo cách l u tr này, khi thao tác cn m t tên t p tin (g m c đ ng đ n), m t con tr t p (FILE * tên_con_tr).

3.5. Các phép toán trong C

Nh đã nói ở trên, v i m i ki u d li u ta ch có th th c hi n m t s phép toán nh t đ nh trên các d li u c a ki u. Ta không th áp d ng m t s phép toán trên các d li u thu c ki u này cho các d li u thu c ki u khác. Ta có th phân t p hp các phép toán trên các ki u d li u c a Pascal thành hai l p sau:

3.5.1. Các phép toán truy c p

Phép toán này dùng để truy c p đ n các thành ph n c a m t đ i t ng d li u, ch ng h n truy nh p đ n các ph n t c a m t m ng, đ n các thành ph n d li u c a c u trúc.

Ví dụ:

- Gi s A là m t m ng v i t p ch s I. Khi đó A[i] cho phép ta truy c p đ n thành ph n th i+1 c a m ng.

- N u X là m t bi n c u trúc thì vi c truy c p đ n tr ng F c a nó đ c th c hi n b i phép toán X.F.

3.5.2. Các phép toán k t hp d li u

Ngôn ng l p trình C có m t t p hp phong phú các phép toán k t hp m t ho c nhi u d li u đã cho thành d li u m i. sau đây là m t s nhóm các phép toán chính.

1. Các phép toán s h c: Đó là các phép toán +, - , * , / trên t p s th c; các phép toán +, - , * , / , %, trên t p s s nguyên.

2. Các phép toán so sánh: Trên các đ i t ng thu c các ki u có th t , ta có th th c hi n các phép toán so sánh ==, !=, <, >, <=, >=. C n chú ý r ng, k t qu c a các phép toán này là m t giá tr có ki u *boolean*.

3. Các phép toán logic: Đó là các phép toán $\&$, $\|$, $!$, $\neg$ có thể hiển trên hai giá trị khác 0 và bằng 0.

4. Thuật toán, phân tích VÀ đánh giá thuật toán

4.1. Thuật toán

Thuật toán (algorithm) là một trong những khái niệm quan trọng nhất trong tin học. Thuật ngữ thuật toán xuất phát từ nhà toán học Ảrập 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. Thuật toán nội tiêng nhất, có thể hiểu là thuật toán Euclid, thuật toán tìm ước chung lớn nhất của hai số nguyên. Có thể mô tả thuật toán này như sau

Thuật toán Euclid

Vào: m, n nguyên dương

Ra: d , ước 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 $d \leftarrow n$ (gán giá trị của n cho d) và dừng lại

Ngược lại, thì $m \leftarrow n, n \leftarrow r$ và quay lại bước 1

Khái niệm: Thuật toán 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.

Đặc trưng của thuật toán

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

1. Đầu vào: Mỗi thuật toán cần có một số (có thể bằng 0) đầu vào (input). Đó là các giá trị đưa vào khi thuật toán bắt đầu làm việc. Các đầu vào này cần được lấy từ các tệp hay giá trị từ đâu đó. Chẳng hạn, trong

thuật toán Euclid trên, m và n là các đầu liệu vào lấy từ tập các số nguyên dương.

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

3. Tính xác định: Mỗi bước của thuật toán cần phải được mô tả một cách chính xác, chỉ có một cách hiểu duy nhất. Hiện 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 đầu liệu vào, những người khác thực hiện thuật toán khác nhau có thể dẫn đến các kết quả khác nhau. Để đảm bảo được tính xác định thuật toán 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.

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

5. Tính dừng: Với mỗi bộ đầu liệu vào thỏa mãn các điều kiện của đầu liệu vào, thuật toán phải dừng lại sau một số hữu hạn các bước cần thực hiện. Chẳng hạn, thuật toán Euclid thỏa 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 $n > r = n_1 > r_1 = n_2 > r_2 \dots$. Dãy số nguyên dương giảm dần cần phải kết thúc ở 0, do đó sau một số bước nào đó giá trị của r phải bằng 0, thuật toán dừng.

4.2. Biểu diễn thuật toán

Có nhiều phương pháp biểu diễn thuật toán. Có thể biểu diễn thuật toán bằng danh sách các bước, các bước được diễn đạt bằng ngôn ngữ thông thường và các ký hiệu toán học. Có thể biểu diễn bằng sơ đồ khối. Tuy nhiên, như đã trình bày, để đảm bảo tính chính xác của thuật toán nên thuật toán cần được viết trong ngôn ngữ lập trình.

4.3. Phân tích thuật toán

Giả sử để giải một bài toán nào đó chúng ta có một số thuật toán giải. Một câu hỏi đặt ra là, chúng ta cần chọn thuật toán nào trong số thuật toán đã có để giải bài toán một cách hiệu quả nhất. Sau đây ta phân tích thuật toán và đánh giá để lựa chọn tập tính toán của nó.

4.3.1. Tính hiệu quả của thuật toán

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

1. Thuật toán đơn giản, dễ hiểu, dễ cài đặt (dễ viết chương trình)
2. Thuật toán sử dụng tập kiến thức nền tảng tài nguyên của máy tính, và dễ biết, chính xác nhất có thể được.

Khi ta viết một chương trình cho một số dòng mã ít hơn, 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 (thức học hàm) để sử dụng nhiều hơn, 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, bởi rất nhiều người trong các bài toán khác nhau. Trong trường hợp này ta cần dựa trên tiêu chuẩn (2). Ta sẽ cài đặt thuật toán có thể rất phức tạp, miễn là chương trình nhận được chạy nhanh hơn các thuật toán khác.

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

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

- Thời gian cần thiết để thực hiện thuật toán (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...).

Chúng ta sẽ chỉ quan tâm đến thời gian thực hiện thuật toán. Vì vậy khi nói đến đánh giá độ phức tạp của thuật toán, có nghĩa là ta nói đến đánh giá thời gian thực hiện. Một thuật toán có hiệu quả được xem là thuật toán có thời gian chạy ít hơn các thuật toán khác.

4.3.2. Đánh giá thời gian thực hiện của thuật toán

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

Phương pháp thực 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 nhân tố sau đây:

1. Các dữ liệu vào
2. Chương trình dịch để chuyển chương trình 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 nhân tố, nên ta không thể biểu diễn chính xác thời gian chạy là bao nhiêu đơn vị thời gian chuẩn, chúng hơn nó là bao nhiêu giây.

Phương pháp lý thuyết: ta sẽ coi thời gian thực hiện của thuật toán như là một hàm số của kích thước dữ liệu vào. Kích thước của dữ liệu vào là một tham số được truyền cho dữ liệu vào, nó có ảnh hưởng quyết định đến thời gian thực hiện chương trình. Cái mà chúng ta chọn làm kích thước của dữ liệu vào phụ thuộc vào các thuật toán cụ thể. Chẳng hạn, đối với các thuật toán sắp xếp mảng, thì kích thước của dữ liệu vào là số thành phần của mảng, đối với thuật

toán gì? Nếu bạn phân tích kỹ thuật tính với n lớn, ta nhận thấy kích thước của thông tin đầu vào là một số nguyên dương n . Ta sẽ sử dụng hàm số $T(n)$, trong đó n là kích thước của dữ liệu đầu vào, để biểu diễn thời gian thực hiện của một thuật toán.

Ta có thể xác định thời gian thực hiện $T(n)$ là số phép toán số cần phải tiến hành khi thực hiện thuật toán. Các phép toán số cần là các phép toán mà thời gian thực hiện bị chặn trên bởi một hằng số không phụ thuộc vào cách cài đặt để thực hiện. Chẳng hạn các phép toán số học $+$, $-$, $*$, $/$, các phép toán so sánh $==$, $!=$... là các phép toán số cần.

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 bắt đầu qua nhận xét phụ thuộc vào cách cài đặt, chủ yếu tập trung vào xác định độ lớn của thời gian thực hiện $T(n)$. Ký hiệu toán học O (đọc là ô-lô-n) để chỉ số độ phức tạp mô tả độ lớn của hàm $T(n)$.

Gọi s 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-lô-n của $f(n)$), nếu và chỉ nếu tồn tại các hằng số dương c và n_0 sao cho $T(n) \leq c.f(n)$, với $\forall n > n_0$.

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

Ví dụ: Gọi s $T(n) = 10n^2 + 4n + 4$

Ta có: $T(n) \leq 10n^2 + 4n^2 + 4n^2 = 12n^2$, với $\forall n \geq 1$

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

Bảng sau đây cho ta các cấp thời gian thực hiện thuật toán để chỉ số độ phức tạp rõ ràng nhất và tên gọi thông thường của chúng.

Ký hiệu ô-lô-n (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 tính.

Các hàm như $\log_2 n$, n , $n \log_2 n$, n^2 , n^3 được gọi là các hàm đa thức. Thuật toán có thời gian tính là các hàm đa thức thì thường chấp nhận được.

Các hàm như 2^n , $n!$, n^n được gọi là hàm lũy thừa. Một thuật toán mà thời gian tính của nó là các hàm lũy thừa thì được coi là không chấp nhận được.

4.3.4. Xác định độ phức tạp tính toán

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

Quy tắc cộng:

Giả sử $T_1(n)$ và $T_2(n)$ là thời gian tính 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 tính của P_1 và 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 tính toán mà thời gian tính của từng bước lần lượt là $O(n^2)$, $O(n^3)$ và $O(n \log_2 n)$ thì thời gian tính của cả ba bước đầu là $O(\max(n^2, n^3)) = O(n^3)$. Khi đó thời gian tính của chương trình sẽ là $O(\max(n^3, n \log_2 n)) = O(n^3)$.

Quy tắc nhân:

Nếu thời gian chạy với P1 và P2 là $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$ thì thời gian thực hiện P1 và P2 liên nhau sẽ là: $T_1(n)T_2(n) = O(f(n)g(n))$

Trong sách báo quy tắc các thuật toán thông thường được trình bày dưới dạng các thủ tục hoặc hàm trong ngôn ngữ Pascal. Để đánh giá thời gian thực hiện thuật toán, ta cần biết cách đánh giá thời gian thực hiện các câu lệnh của C. Các câu lệnh trong C được định nghĩa như sau:

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

2. Nếu $S_1, S_2, \dots, S_n$ là các câu lệnh thì

$$\{ S_1, S_2, \dots, S_n \}$$

là câu lệnh và được gọi là 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**.

4. Nếu $S_1, S_2, \dots, S_{n+1}$ là các câu lệnh, E là biểu thức có kiểu thực thể được định nghĩa, và $v_1, v_2, \dots, v_n$ là các giá trị cùng kiểu với E thì

Switch (E)

{

Case v_1 : S_1 ;

Case v_2 : S_2 ;

.....

Case v_n : S_n ;

[Default : S_{n+1} ;]

}

end;

là câu lệnh và khối code là lệnh **Switch**.

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

while (E) S;

là câu lệnh và khối code là lệnh **while**.

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

DO { $S_1, S_2, \dots, S_n$ } **while** (E)

là câu lệnh và khối code là lệnh **Do ...while**

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

for ($i = E_1; i \leq E_2; i++$) S ;

là câu lệnh, và

for ($i = E_2; i \geq E_1; i--$) S ;

là câu lệnh. Lệnh này khối code là lệnh **for**.

Giống rằng, các lệnh gán không chứa các biến hàm. Khi đó để đánh giá thời gian thực hiện một chương trình, ta có thể áp dụng phương pháp dưới đây

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

2. Lệnh lặp thành: thời gian thực hiện lệnh lặp thành được xác định bởi lượt lặp.

3. Lệnh **if**: Giống 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**: Giống thời gian thực hiện lệnh S (thân của while) là $O(f(n))$ và $g(n)$ là số lần lặp 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 $\{ S_1, S_2, \dots, S_n \}$ 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 đương với lệnh **Do .. While** và **While**.

Đánh giá thời gian thực hiện (học hàm) đệ quy:

Trong học hàm chúng ta xét một ví dụ cụ thể, ta sẽ đánh giá thời gian thực hiện của hàm đệ quy sau:

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

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

- Với $n=1$, chức năng thực hiện lệnh gán $fact = 1$, do đó $T(1) = O(1)$.

- Với $n > 1$, chức năng 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 lệnh gọi đệ quy $fact(n - 1)$). Tóm lại, ta có quan hệ đệ quy 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 quy, tìm $T(n)$, chúng ta áp dụng phương pháp thử nghiệm. Ta có phương trình đệ quy

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

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 toán 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, $T(n) = O(n)$.

Tại ví dụ trên, ta suy ra phương pháp tổng quát sau đây để đánh giá thời gian thực hiện một thao tác (hàm) đệ quy. Để đơn giản, ta giả thiết rằng các thao tác (hàm) là đệ quy trực tiếp. Điều đó có nghĩa là các thao tác (hàm) chỉ chứa các lời gọi đệ quy đến chính nó. Giả sử thời gian thực hiện thao tác là $T(n)$, với n là kích thước dữ liệu đưa vào. Khi đó thời gian thực hiện các lời gọi đệ quy 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 thao tác theo quy tắc 1-7 (quy tắc đánh giá thời gian thực hiện các câu lệnh) ta sẽ nhận được quan hệ đệ quy sau :

$$T(n) = F(T(m_1), T(m_2), \dots, T(m_k))$$

Trong đó $m_1, m_2, \dots, m_k < n$. Giả sử phương trình đệ quy này, ta sẽ nhận được số đánh giá của $T(n)$.

4.4. Phân tích thuật toán

Ví dụ 1: Phân tích thuật toán 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 thuật toán phụ thuộc vào số lần nhúng trong hai số m và n . Giả sử $m \geq n > 0$, khi đó cần đưa 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 thuật toán là thời gian thực hiện các 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 lần nhúng các lệnh thực hiện lặp lại.

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 có hai lần thực hiện khối lệnh thì phần dư 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 của while là $O(\log_2 n)$. Đó cũng là thời gian thực hiện của thuật toán.

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

$$e^x = 1 + x/1! + x^2/2! + \dots + x^n/n!, \text{ với } x \text{ và } n \text{ cho trước}$$

Float *Exp1* (*int* *n*, *float* *x*)

{Tính tổng 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 giả 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

Vì $i = 2$ thì câu lệnh này được thực hiện 2 lần

.....

Vì $i = n$ thì câu lệnh này được thực hiện n lần

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

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

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 : Đưa vào số hạng trước để tính số hạng sau

$$\frac{x^2}{2!} = \frac{x}{1!} \cdot \frac{x}{2}, \dots, \frac{x^n}{n!} = \frac{x^{n-1}}{(n-1)!} \cdot \frac{x}{n}$$

```
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)
```

```
}
```

Tổng cộng thì hai giai đoạn trên ta có thể nói rằng giai đoạn tìm kiếm hai phần tử trong mảng có độ phức tạp là $O(1)$. Do đó thì độ phức tạp của hai giai đoạn trên là $O(n)$ nên độ phức tạp của hai giai đoạn trên là $O(n)$.

Như vậy thì hai giai đoạn trên ta có thể nói rằng giai đoạn tìm kiếm hai phần tử trong mảng có độ phức tạp là $O(n)$ (vì độ phức tạp của hai giai đoạn trên là $O(n)$ nên độ phức tạp của hai giai đoạn trên là $O(n)$).

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

Vào: Dãy $s_1, s_2, \dots, s_n$ và khóa cần tìm x

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

Int *linear_search*(*day s*, *int n*, *kdl x*);

{trong đó **day**, **kdl** là dãy các phần tử và kiểu dữ liệu đã định nghĩa tại trước}

{

Int *i*;

i = 0;

do

i = *i* + 1;

while (*i* > *n*) || (*s*[*i*] = *x*);

return(*i*);

}

Ví dụ này ta không thể đánh giá hai ví dụ trên. Do quá trình tìm kiếm không nhất thiết phải dựa vào kích thước của dữ liệu vào, mà còn phụ thuộc vào tình trạng của dữ liệu. Tức là độ phức tạp của hai giai đoạn trên 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

trình hợp nh trên ta cần phải đánh giá thời gian tính tất nhất, tất nhất và trung bình của thuật toán với độ dài đầu vào n . Rõ ràng thời gian tính của thuật toán có thể được đánh giá bởi số lần thực hiện câu lệnh $i := i + 1$.

Nếu $s[1] = x$ thì câu lệnh $i := i + 1$ trong thân vòng lặp repeat thực hiện 1 lần. Do đó thời gian tính tất nhất của thuật toán 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 thời gian tính tất nhất là $O(n)$.

Cùng ta tính thời gian tính trung bình của thuật toán. Nếu x được tìm thấy ở vị trí thứ i của dãy thì câu lệnh $i := i + 1$ phải thực hiện i lần ($i = 1, 2, \dots, n$), 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

$$[(1 + 2 + \dots + n) + n] / (n + 1)$$

Ta có

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

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

Nhận xét:

Việc xác định $T(n)$ trong trình hợp trung bình thường gặp nhiều khó khăn vì số phải dùng tới công cụ toán để biết, hơn nữa tính trung bình có nhiều cách quan niệm. Trong các trình hợp mà $T(n)$ trung bình thường khó xác định nên ta thường đánh giá giới thuật qua giá trị xấu nhất của $T(n)$. Hơn nữa, trong một số thuật toán, việc xác định trình hợp xấu nhất là rất quan trọng.