CHƯƠNG 1
MỘT SỐ KHÁI NIỆM CƠ BẢN

Nội dung chính

- 1.1. Thuật toán (Algorithm)
- 1.2. Biểu diễn thuật toán
- 1.3. Độ phức tạp của thuật toán (Algorithm Complexity)
- 1.4. Cấu trúc dữ liệu (Data structure)
- 1.5. Các chiến lược thiết kế thuật toán

1.1 Thuật toán (Algorithm)

1.1.1. Khái niệm

Là một hệ thống chặt chẽ và rõ ràng các quy tắc nhằm xác định một dãy thao tác sao cho: với một bộ dữ liệu vào, sau một số hữu hạn bước thực hiện các thao tác đã chỉ ra, ta đạt được mục tiêu đã định.

1.1 Thuật toán (Algorithm)

1.1.2. Các đặc trưng của thuật toán

a. Tính đúng đắn :

- Cho kết quả đúng đối với các bộ dữ liệu đầu vào.
- Đặc trưng quan trọng nhất đối với một thuật toán.

b. Tính dừng:

- Đảm bảo sẽ dừng sau một số hữu hạn các bước.

1.1 Thuật toán (Algorithm)

c. Tính xác định:

- Các bước phải rõ ràng, cụ thể.
- Tránh nhập nhằng, nhầm lẫn đối với người đọc và hiểu, cài đặt.

d. Tính hiệu quả:

- Có khả năng giải quyết hiệu quả bài toán đặt ra trong thời gian hoặc các điều kiện cho phép trên thực tế đáp ứng được yêu cầu người dùng.

1.1 Thuật toán (Algorithm)

e. Tính phổ quát (phổ biến):

- Có thể giải quyết được một lớp các bài toán tương tự.

Các giá trị đầu vào được gọi là các giá trị dữ liệu Input.

Kết quả của thuật toán gọi là Output.

1.2 Biểu diễn thuật toán

1.2.1. Ngôn ngữ tự nhiên

Sử dụng một loại NNTN để liệt kê các bước của thuật toán

Ví dụ: Thuật toán tính tổng hai số a và b:

Bước 1 : Nhập vào các số a và b;

Bước 2 : Tính tổng a+b;

Bước 3 : Xuất kết quả của tổng a+b

1.2 Biểu diễn thuật toán

❖ **Ưu điểm:**

- Đơn giản.
- Không yêu cầu người viết, người đọc có kiến thức nền tảng.

❖ **Nhược điểm:**

- Dài dòng.
- Không làm nổi bật cấu trúc của thuật toán.
- Khó biểu diễn với những bài toán phức tạp.

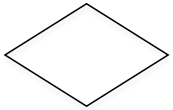
1.2 Biểu diễn thuật toán

1.2.2. Lưu đồ

Bắt đầu hoặc kết thúc

Nhập hoặc xuất dữ liệu

Xử lí, tính toán...

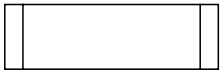


Chứa các biểu thức kiểm tra điều kiện và rẽ nhánh chương trình

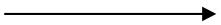
Điểm nối (Sử dụng khi vượt quá trang)

Chuẩn bị

Tập tin dữ liệu



Khối chương trình con



Hướng đi của thuật toán và kết nối các hình

Các chú thích, giải thích

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Ưu điểm: rõ ràng, tường minh.

Nhược điểm: cồng kềnh.

1.2 Biểu diễn thuật toán

1.2.3. Mã giả (pseudocode)

Gần giống với NNLT. Gồm:

- Ngôn ngữ tự nhiên, các ký hiệu toán học.
- Vay mượn một số cấu trúc của NNLT nào đó.

Không thể thực thi trên máy tính.

Mỗi tác giả viết có mỗi phong cách khác nhau, miễn là trình bày rõ ràng và thể hiện được cách giải quyết bài toán.

1.2 Biểu diễn thuật toán

Ví dụ: Tìm số lớn nhất trong hai số a và b:

1. Nhập giá trị a,b;
2. if (a ≥ b) then
   Xuất kết quả: Số lớn nhất là a;
   else
   Xuất kết quả: Số lớn nhất là b

Ưu điểm:

Tiện lợi, đơn giản

Dễ hiểu, dễ diễn đạt

1.2.4 Ngôn ngữ lập trình

C/C++, Java, ….

1. Sự cần thiết

- Lựa chọn thuật toán tốt nhất trong các thuật toán để dễ cài đặt chương trình.

- Cải tiến thuật toán hiện có để được một thuật toán tốt hơn.

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

- Một thuật toán được xem là tốt nếu nó đạt các yếu tố sau:
  - Thực hiện đúng.
  - Tốn ít bộ nhớ (chi phí không gian bộ nhớ).
  - Thực hiện nhanh (chi phí thời gian thực thi thuật toán).

2. Thời gian thực hiện chương trình

Là một **hàm** của kích thước dữ liệu vào, ký hiệu **T(n)** trong đó n là kích thước (độ lớn) của dữ liệu đầu vào.

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

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Thời gian thực hiện là một hàm không âm, tức là:

$$T(n) \geq 0 \ \forall \ n \geq 0.$$

Đơn vị đo của T(n) không phải là giờ, phút, giây...

Là một lệnh được thực hiện trong một máy tính lý tưởng.

Ví dụ: T(n) = Cn thì có nghĩa là chương trình ấy cần Cn chỉ thị thực thi.

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Trên mọi bộ dữ liệu vào có kích thước n thì:

- Trường hợp **xấu nhất** (Worst-case) : tìm T(n) là **thời gian lớn nhất** khi thực hiện thuật toán
- Trường hợp **tốt nhất** (Best-case) : tìm T(n) là **thời gian ít nhất** khi thực hiện thuật toán

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

- Thời gian **trung bình** (average-case) : Giả sử rằng dữ liệu vào tuân theo một **phân phối xác suất** nào đó và tính toán giá trị **kỳ vọng** (trung bình) của thời gian chạy cho mỗi kích thước dữ liệu n (T(n)), sau đó phân tích thời gian thực hiện thuật toán dựa trên hàm T(n).

Để tính thời gian **trung bình,** thực hiện các bước sau:

1. Quyết định một **không gian lấy mẫu** (sampling space) để diễn tả những dữ liệu đầu vào (kích thước n) có thể có. Giả sử không gian lấy mẫu S= $\{I_1, I_2, \ldots, I_n\}$

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

2. Định nghĩa một **phân bố xác suất** P trên S mà biểu diễn mức độ chắc chắn mà dữ liệu đầu vào có thể xảy ra.

3. Tính **tổng số tác vụ** căn bản được giải thuật thực hiện để xử lý một trường hợp mẫu. Dùng $v(I_k)$ ký hiệu tổng số tác vụ được thực hiện bởi giải thuật khi dữ liệu đầu vào thuộc trường hợp I_k.

4. Tính **giá trị trung bình** của số tác vụ căn bản bằng cách tính kỳ vọng sau:

$$T_{avg}(n)= v(I_1).p(I_1)+ v(I_2).p(I_2)+ \ldots + v(I_k).p(I_k)$$

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Ví dụ

```
int search(int a[], int n, int X)
{
    int i=0;
    while (i<n && a[i] != X)
        i++;
    return i;
}
```

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Giả sử X có xuất hiện trong mảng và giả định rằng xác suất để nó xuất hiện tại 1 vị trí bất kì trong mảng là đều nhau và xác suất để mỗi trường hợp xảy ra là **p = 1/n.**

Số lần so sánh để tìm thấy X nếu nó xuất hiện tại vị trí 1 là 1

Số lần so sánh để tìm thấy X nếu nó xuất hiện tại vị trí 2 là 2

…

Số lần so sánh để tìm thấy X nếu nó xuất hiện tại vị trí *n* là *n*

Tổng số tác vụ so sánh trung bình là:

$$
\begin{aligned}
T(n) &= 1.(1/n) + 2.(1/n) + \ldots + n.(1/n) \\
&= (1 + 2 + \ldots + n).(1/n) \\
&= (1 + 2 + \ldots + n)/n \\
&= n.(n+1)/2.(1/n) \\
&= (n+1)/2.
\end{aligned}
$$

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Ví dụ: Search (a,n,x)

0	1	2	3	4	5	6	7
5	6	3	10	1	4	7	9

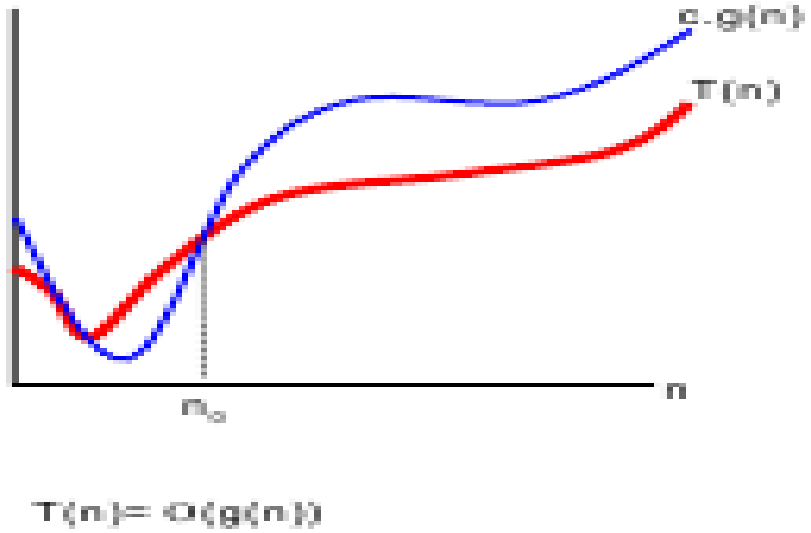
Search(a,n,5): tốt nhất.

Search(a,n,15): xấu nhất.

Search(a,n,10): trung bình.

3. Khái niệm

Big – O (O lớn): tồn tại các hằng số dương c và n_0 sao cho $T(n) \leq c.g(n)$ với mọi $n \geq n_0$. Viết là O(g(n)).



1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Big–O được sử dụng để chỉ định các chặn trên của hàm tăng.

Độ phức tạp thời gian của thuật toán:

Ký hiệu O(g(n)). Viết là: T(n)=O(g(n)).

Ví dụ: O(n), O(nlogn), $O(n^2)$,…

Ví dụ: $T(n) = (n + 1)^2 = n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 = 4n^2$

với $\forall n \geq 1 \Rightarrow T(n) = O(n^2)$.

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Ví dụ: 1 bài toán có 2 thuật toán P_1, P_2

– P_1 có $T_1(n) = 100n^2$

– P_2 có $T_2(n) = 5n^3$

Chọn P_1 hay P_2 ?

Nếu $n \leq 20$ thì $T_1(n) \geq T_2(n)$

Nếu $n > 20$ thì $T_1(n) < T_2(n)$

Các độ phức tạp thường gặp:

1, logn, n, nlogn, n^2, n^3, 2^n, n!, n^n

ĐPT đa thức => chấp nhận được.

ĐPT hàm mũ => cải tiến/ tìm thuật toán khác.

Có những ký hiệu khác ω và Θ nhưng thường sử dụng nhất là O.

4. Quy tắc xác định độ phức tạp

- **T(n) = C** (hằng số dương): **O(1).**

- **Bỏ hằng**: T(n) = O(C(f(n)) = O(f(n)).

  Ví dụ: O(5n) = O(n)

- **Đa thức**: lấy đơn thức có mũ cao nhất.

  $T(n)= 5n^3 +2n^2 + 6n$ thì $T(n) = O(n^3)$.

- **Quy tắc lấy max**:

  T(n) = O(f(n)+ g(n)) = O(max(f(n), g(n)))

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

$T_1(n)$ và $T_2(n)$ là thời gian thực hiện của hai đoạn chương trình P_1 và P_2.

$$T_1(n) = O(f(n)),\ T_2(n) = O(g(n))$$

- **Quy tắc cộng**: P_1 và P_2 nối tiếp nhau (tuần tự) thì:

$$T(n) = T_1(n) + T_2(n) = O(f(n) + g(n)) =$$

$$O(\max(f(n),g(n))).$$

- **Quy tắc nhân**: P_1 và P_2 lồng nhau thì:

$$T(n) = O(f(n).g(n)).$$

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Quy tắc chung không có CT con

- **Các lệnh**: gán, nhập, xuất, so sánh, return: **O(1)**
- Một chuỗi **tuần tự** các lệnh: **Quy tắc cộng**
- Lệnh **if**: if (điều kiện)

lệnh 1

else

lệnh 2

lấy **max** (lệnh 1, lệnh 2) + điều kiện

- **Vòng lặp**: tổng thời gian thực hiện thân vòng lặp, trong trường hợp không xác định được số lần lặp ta phải lấy số lần lặp trong trường hợp xấu nhất.

Trình tự đánh giá:

Nối tiếp: Từ trên xuống.

Lồng nhau: Từ trong ra.

for(i=a; i<=b; i++): số lần lặp= b-a+1

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

for(i=1; i<=n; i= 2*i)

Sau lần lặp thứ 1: $i = 2^1$

Sau lần lặp thứ 2: $i = 4 = 2^2$

…..

Sau lần lặp thứ k: $i = 2^k$

Lặp kết thúc khi $i = n = 2^k$

=> $k = \log_2 n$ =>số lần lặp = $\log_2 n$

- **while, do…while** : xác định số lần lặp trường hợp xấu nhất.

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Phương pháp thực hiện: Xác định đầu vào thường ký hiệu là n.

- **Cách 1**: dùng cho tất cả các loại chương trình.
  - Tính thời gian thực hiện T(n) cho toàn bộ chương trình => O(f(n)) từ T(n).
- **Cách 2**: không áp dụng cho chương trình đệ quy
  - Chia chương trình nhiều đoạn nhỏ.
  - Tính $T_i(n)$ và $O(f_i(n))$ cho từng đoạn.
  - Áp dụng quy tắc cộng, quy tắc nhân để có O(f(n)) cho cả chương trình

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Ví dụ 1:

1	sum=0;	O(1)
2	for(i=1;i<=n;i=i*2) {	O(logn)
3	cin>>x;	O(1)
4	sum=sum+x ;}	O(1)

T(n) = O(logn)

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Ví dụ 2: sắp xếp nổi bọt

	void Sort(int a[],int n){	4,5,6: O(1)
	int i,j,temp;	3: O(1)
1	for(i= 0; i<=n-2; i++)	2: lặp (n-1)-(i+1)+1= n-i-1 lần
2	for(j=n-1; j>=i+1;j--)	2: O(n-i-1)
3	if (a[j] < a[j-1]) {	1: chính là toàn chương trình
4	temp=a[j-1];	
5	a[j-1]= a[j];	
6	a[j]= temp;}}	

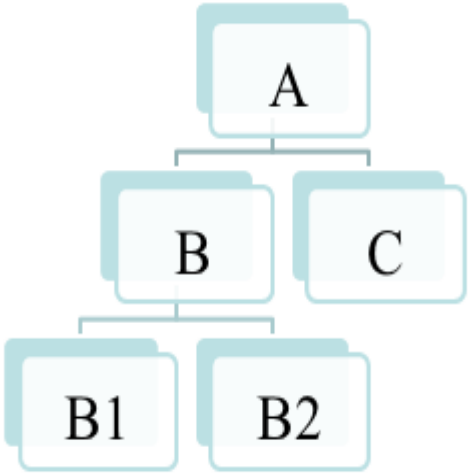
$$T(n) = \sum_{i=0}^{n-2}(n-i-1) = \frac{n(n-1)}{2} = O(n^2)$$

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Ví dụ 3: Tìm kiếm

	int search (int x, int a[], int n){ int i, found;	• 1, 2 và 7: O(1)
1	i=0;	• 5 và 6: O(1) => 4: O(1)
2	found = 0;	• Trong trường hợp xấu nhất 3 thực hiện 2n lần => 3: O(n)
3	while (i <= n-1 && ! found)	
4	if (x == a[i])	
5	found = 1;	T(n) = O(n)
6	else i++;	
7	return found;	
	}	

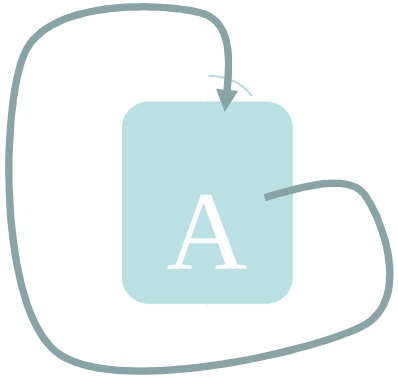
Đánh giá CT có gọi CTC không đệ quy



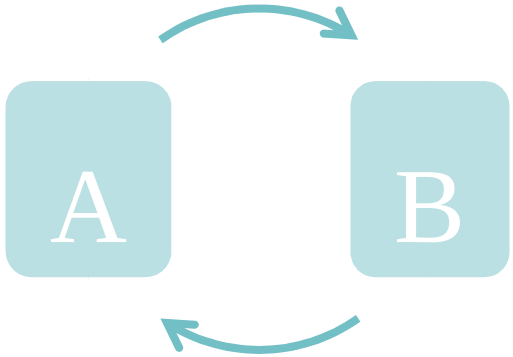
- Đánh giá: C, B1, B2
- Đánh giá B
- Đánh giá A

- Quy trình: "Trong ra"/ "Dưới lên"
- Đánh giá CTC không gọi CTC
- Đánh giá CTC gọi CTC đã được đánh giá

Đánh giá các CT đệ quy



ĐỆ QUY TRỰC TIẾP



ĐỆ QUY GIÁN TIẾP

- Không thể áp dụng "trong ra"
- Giải pháp:
  - Thành lập phương trình đệ quy.
  - Giải phương trình đệ quy

Thành lập phương trình đệ quy

Ví dụ n!= n*(n-1)*(n-2)*….*2*1

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

```
int  Giai_thua(int  n){
   if  (n==0)  return  1;
   return  n*Giai_thua(n-1);}
```

- Gọi T(n) là thời gian tính n!.
- Thì T(n-1) là thời gian tính (n-1)!.
- Khi n = 0 thì CT return 1, tốn O(1), do đó ta có T(0) = 1.
- Khi n > 0 thì CT phải:
  - Tính (n-1)!, tốn thời gian T(n-1)
  - Tính n *(n-1)! và return kết quả tốn hằng thời gian, cho là 1

$$\mathbf{T(n) = \begin{cases} 1 & \text{nếu } n = 0 \\ T(n-1) + 1 & \text{nếu } n > 0 \end{cases}}$$

Tìm giá trị lớn nhất của mảng có n phần tử

$$A\_Max(a, L, R) = \begin{cases} a[L] & \text{nếu } L = R \\ Max(A\_Max(a, L, M), A\_Max(a, M+1, R)) \end{cases}$$

- Gọi T(n) là thời gian tìm GTLN của mảng có n phần tử
- Thì $T(\frac{n}{2})$ là thời gian tìm GTLN của mảng có n/2 phần tử
- Khi n=1 (L=R) thì CT return a[L], tốn O(1) => T(1) = 1
- Khi n>1 thì CT phải:
  - Tìm GTLN của 2 mảng có n/2 phần tử, tốn thời gian 2 $T(\frac{n}{2})$
  - Thực hiện Max và return kết quả tốn hằng thời gian, cho là 1

$$\mathbf{T(n)} = \begin{cases} \mathbf{1} & \mathbf{nếu\ n = 1} \\ \mathbf{2T(\frac{n}{2}) + 1\ nếu\ n > 1} \end{cases}$$

Giải phương trình đệ quy

- **Phương pháp truy hồi**
  - Triển khai T(n) theo T(n - 1) rồi T(n - 2) … cho đến T(1) hoặc T(0).
  - Suy ra nghiệm.
- **Lời giải tổng quát của một lớp các phương trình đệ quy**

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Ví dụ 1

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

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

$T(n) = [T(n-2) + 1] + 1 = T(n-2) + 2$

$T(n) = [T(n-3) + 1] + 2 = T(n-3) + 3$

……

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

- Quá trình trên kết thúc khi $n - i = 0 \Leftrightarrow i = n$
- Khi đó ta có $T(n) = T(0) + n = 1 + n = O(n)$

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Ví dụ 2 $T(n)=\begin{cases}1 & \text{nếu } n=1\\ 2T\left(\frac{n}{2}\right)+n & \text{nếu } n>1\end{cases}$

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

$$T(n)=2\left[2T\left(\frac{n}{4}\right)+\frac{n}{2}\right]+n=4T\left(\frac{n}{4}\right)+2n=2^2T\left(\frac{n}{2^2}\right)+2n$$

$$T(n)=4\left[2T\left(\frac{n}{8}\right)+\frac{n}{4}\right]+2n=8T\left(\frac{n}{8}\right)+3n=2^3T\left(\frac{n}{2^3}\right)+3n$$

.....

$$T(n)=2^iT\left(\frac{n}{2^i}\right)+in$$

Kết thúc khi $\frac{n}{2^i}=1\Leftrightarrow 2^i=n\Rightarrow i=\log n$

Ta có $T(n)=nT(1)+n\log n=n+n\log n=O(n\log n)$

Ví dụ 3

Bài toán: Một chương trình đệ quy mà lặp qua bộ dữ liệu nhập để loại đi một phần tử. Hệ thức truy hồi của nó như sau:

$$T(1) = 1$$

$$T(n) = T(n-1) + n \text{ với } n \geq 2$$

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

$$T(n) = T(n-1) + n$$
$$= T(n-2) + (n-1) + n$$
$$= T(n-3) + (n-2) + (n-1) + n$$
$$\ldots.$$
$$= T(1) + 2 + \ldots + (n-2) + (n-1) + n$$
$$= 1 + 2 + \ldots + (n-1) + n$$
$$= n(n+1)/2$$

Vậy $T(n) = O(n^2)$

Ví dụ 4

Bài toán: Một chương trình đệ quy giảm một nửa bộ dữ liệu nhập trong một bước làm việc. Hệ thức truy hồi là:

$$T(1) = 0$$

$$T(n) = T(n/2) + 1 \text{ với } n \geq 2$$

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Ta có: $T(n) = T(n/2) + 1$

Đặt $n = 2^t \Rightarrow t = \log n$

$T(2^t) = T(2^{t-1}) + 1$

$= T(2^{t-2}) + 1 + 1 = T(2^{t-2}) + 2$

$= T(2^{t-3}) + 3$

…

$= T(2^0) + t$

$= T(1) + t$

$= t.$ Vậy $T(n) = O(\log n)$

Ví dụ 5

Bài toán: Một chương trình đệ quy mà chia đôi bộ dữ liệu nhập trong một bước làm việc nhưng phải xem xét từng phần tử trong dữ liệu nhập. Hệ thức truy hồi là:

$$T(1) = 0$$

$$T(n) = 2T(n/2) + n \text{ với } n \geq 2$$

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Ta có: $T(n) = 2T(n/2) + n$

Đặt $n = 2^t \Rightarrow t = \log n$

$T(2^t) = 2T(2^{t-1}) + 2^t$

$T(2^t)/ 2^t = T(2^{t-1})/ 2^{t-1} + 1$

$= T(2^{t-2})/ 2^{t-2} + 1 + 1 = T(2^{t-2})/ 2^{t-2} + 2$

$= T(2^{t-3})/ 2^{t-3} + 3$

…

$= T(2^{t-t})/ 2^{t-t} + t$

$= t$

$T(2^t)/ 2^t = t \Rightarrow T(2^t) = t.2^t$ hay $T(n) = n.\log n$

Vậy $T(n) = O(n.\log n)$

BÀI TOÁN ĐỆ QUY TỔNG QUÁT

- Bài toán kích thước n:
  - Chia bài toán thành a bài toán con, kích thước n/b.
  - Giải các bài toán con và tổng hợp kết quả.
- Với bài toán con: tiếp tục chia cho đến các bài toán kích thước 1.

=>Thuật toán đệ quy.

- Giả thiết T(1) =1
- Giả thiết thời gian để chia bài toán và tổng hợp kết quả là d(n).

LỚP PHƯƠNG TRÌNH ĐỆ QUY TỔNG QUÁT

- Gọi T(n) thời gian giải bài toán kích thước n
- $T(\frac{n}{b})$ thời gian giải bài toán kích thước n/b
- Khi n=1, ta có T(1)=1
- Khi n>1, giải a bài toán kích thước n/b, tốn $aT\left(\frac{n}{b}\right)$ + thời gian phân chia, tổng hợp d(n)
- $$T(n) = \begin{cases} 1 & \text{nếu } n = 1 \\ aT(\frac{n}{b}) + d(n) \end{cases}$$

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Giả thiết: $n=b^k$

$$T(n) = a^k + \sum_{j=0}^{k-1} a^j d\left(b^{k-j}\right)$$

Nghiệm thuần nhất
$a^k = n^{\log_b a}$

Nghiệm riêng

Nghiệm của phương trình là: MAX(NTN,NR).

HÀM NHÂN

- Hàm f(n) là hàm nhân nếu:

  $f(m.n) = f(m).f(n)$ với mọi m và n.

- Ví dụ:
  - Hàm $f(n) = n^k$ là một hàm nhân, vì $f(m.n) = (m.n)^k = m^k.n^k = f(m).f(n)$
  - Hàm f(n) = logn không phải là một hàm nhân,

    vì $f(n.m) = \log(n.m) = \log n + \log m \neq \log n.\log m = f(n).f(m)$

Nghiệm PT khi d(n) là hàm nhân

- Trường hợp 1: a > d(b)

  $T(n) = O(n^{\log_b a})$

  Cải tiến thuật toán: giảm a (giảm số bài toán con)

- Trường hợp 2: a < d(b)

  $T(n) = O(n^{\log_b d(b)})$

  Cải tiến thuật toán: Cải tiến phân chia và tổng hợp kết quả

- Trường hợp 3: a = d(b)

  $T(n) = O(n^{\log_b a} \log_b n)$

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

Ví dụ 1

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

- Phương trình thuộc lớp PT tổng quát
- d(n) = 1 là hàm nhân, a=b=2
- d(b)=1 <a => Trường hợp 1
- $T(n)= O(n^{\log_b a}) = (n^{\log_2 2}) = O(n)$

Ví dụ 2

- $$T(n) = \begin{cases} 1 & \text{nếu } n = 1 \\ 3T(\frac{n}{2}) + n^2 & \text{nếu } n > 1 \end{cases}$$
- Phương trình thuộc lớp PT tổng quát
- $d(n) = n^2$ là hàm nhân, a=3, b=2
- $d(b) = b^2 = 4 > a$ => Trường hợp 2
- $T(n) = O(n^{\log_b d(b)}) = (n^{\log_2 4}) = O(n^2)$

d(n) không phải hàm nhân

- Giả thiết $n = b^k$
- Nghiệm thuần nhất = $n^{\log_b a}$
- Tính nghiệm riêng theo công thức:

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

- Nghiệm PT = MAX(NR,NTN)

Ví dụ

$$T(n)=\begin{cases}1 & \text{nếu } n=1\\ 2T\left(\frac{n}{2}\right)+n\log n\end{cases}$$

- PT thuộc lớp PTTQ nhưng d(n) = nlogn không phải hàm nhân.
- NTN = $n^{\log_b a} = n^{\log 2} = n$
- Phải tính nghiệm riêng

1.3. Độ phức tạp của thuật toán (Algorithm Complexity)

$$NR = \sum_{j=0}^{k-1} a^j d\left(b^{k-j}\right) = \sum_{j=0}^{k-1} 2^j 2^{k-j} \log 2^{k-j}$$

a=b=2
d(n)=nlogn

$$NR = 2^k \sum_{j=0}^{k-1} (k-j) = 2^k \frac{k(k+1)}{2} = O(2^k k^2)$$

- Do $n = b^k$ nên $k = \log_b n$, vì $b = 2 \Rightarrow 2^k = n \Leftrightarrow k = \log n$
- $NR = O(n\log^2 n) > O(n) = NTN$
- $T(n) = O(n\log^2 n)$.

1.4. Cấu trúc dữ liệu (Data structure)

- Việc lập trình là đi tìm một CTDL phù hợp để biểu diễn dữ liệu của bài toán và từ đó xây dựng thuật toán phù hợp CTDL đã chọn.
- CTDL được sử dụng để biểu diễn dữ liệu còn các thuật toán được sử dụng để thực hiện các thao tác trên các dữ liệu của bài toán nhằm hoàn thành các chức năng của chương trình.

1.5. Các chiến lược thiết kế thuật toán

1.5.1. Duyệt toàn bộ (Exhausted search)

- Xét tất cả các ứng cử viên thuộc một không gian có thể có của bài toán để xem đó có phải là nghiệm của bài toán hay không.
- Yêu cầu có một hàm kiểm tra xem một ứng cử viên nào đó có phải là nghiệm của bài toán hay không.
- Mặc dù dễ hiểu nhưng không dễ thực hiện, đặc biệt là không hiệu quả với các bài toán mà kích thước input lớn.

1.5. Các chiến lược thiết kế thuật toán

1.5.2. Đệ qui quay lui – Backtracking

Xây dựng thuật toán dựa trên quan hệ đệ qui.

Nghiệm của bài toán được mô hình hóa dưới dạng một vecto, mỗi thành phần của vecto nghiệm sẽ có một tập giá trị có thể nhận và thuật toán sẽ tiến hành các bước gán các giá trị có thể cho các thành phần của nghiệm để xác định đúng nghiệm của bài toán.

Mặc dù không phải bài toán nào cũng có thể áp dụng song các thuật giải dựa trên phương pháp đệ qui quay lui luôn có vẻ đẹp từ sự ngắn gọn, súc tích mà nó mang lại.

1.5. Các chiến lược thiết kế thuật toán

1.5.3. Chia để trị (Divide and Conquer)

- Chiến lược quan trọng trong việc thiết kế các thuật toán.
- Chia bài toán ban đầu thành các bài toán nhỏ hơn, giải các bài toán nhỏ hơn đó, sau đó kết hợp nghiệm của các bài toán nhỏ hơn đó lại thành nghiệm của bài toán ban đầu.

1.5. Các chiến lược thiết kế thuật toán

1.5.3. Chia để trị (Divide and Conquer)

Khó khăn:

Làm thế nào để chia tách bài toán một cách hợp lý thành các bài toán con, vì nếu các bài toán con lại được giải quyết bằng các thuật toán khác nhau thì sẽ rất phức tạp.

Việc kết hợp lời giải của các bài toán con sẽ được thực hiện thế nào?

1.5. Các chiến lược thiết kế thuật toán

1.5.4. Chiến lược tham lam (Greedy)

- Xây dựng thuật toán tìm nghiệm tối ưu cục bộ cho các bài toán tối ưu nhằm đạt được nghiệm tối ưu toàn cục cho cả bài toán (trong trường hợp tổng quát).
- Trong trường hợp cho nghiệm đúng, lời giải của chiến lược tham lam thường rất dễ cài đặt và có hiệu năng cao (độ phức tạp thuật toán thấp).
- Chú ý: Trong một số bài toán nếu xây dựng được hàm chọn thích hợp có thể cho nghiệm tối ưu. Trong nhiều bài toán, thuật toán tham ăn chỉ cho nghiệm gần đúng với nghiệm tối ưu.

1.5. Các chiến lược thiết kế thuật toán

1.5.5. Qui hoạch động (Dynamic Programming)

- Xây dựng thuật toán để giải quyết các bài toán tối ưu.
- Xây dựng các quan hệ đệ qui của bài toán, bài toán gốc sẽ có lời giải dựa trên các bài toán con (sub problems) dựa trên quan hệ đệ qui.
- Thường sử dụng các mảng để lưu lại giá trị nghiệm của các bài toán con.
- Có hai cách tiếp cận: bottom up và top down.

Một số công thức tính tổng thường gặp

- $1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$
- $1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$
- $1 + a^1 + a^2 + \cdots + a^n = \frac{a^{n+1}-1}{a-1}$
- $\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^n} = 1 - \frac{1}{2^n}$
- $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \ln(n) + \gamma$

$\gamma = 0{,}577215665$: *hằng số Euler*

Bài tập

1. Giải các phương trình đệ quy

a. $T(1) = C_1$, $T(n) = 2T(n-1) + C_2$

b. $T(1) = 1$, $T(n) = T(n-1)+n \quad \forall n \geq 2$

c. $T(1) = 0$, $T(n) = T\left(\frac{n}{2}\right) + 1 \; \forall n \geq 2$

d. $T(1) = 0$, $T(n) = 2T\left(\frac{n}{2}\right) + n \; \forall n \geq 2$

2. Giải các phương trình đệ quy sau với $T(1) = 1$

a. $T(n) = 4T(\frac{n}{2}) + n$

b. $T(n) = 4T(\frac{n}{2}) + n^2$

c. $T(n) = 4T(\frac{n}{2}) + n^3$

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

Bài tập

3. Xây dựng sơ đồ thuật toán cho bài toán tính số Fibonaci thứ N, biết rằng dãy số Fibonaci được định nghĩa như sau:

F[0] = F[1] = 1, F[N] = F[N-1] + F[N-2] với $N \geq 2$.

4. Hãy viết chương trình nhanh nhất có thể được để in ra tất cả các số nguyên số có hai chữ số.

5. Áp dụng thuật toán sàng để in ra tất cả các số nguyên tố nhỏ hơn N.


Thank You!