Môn học:

Phân tích và Thiết kế Giải thuật

Số tín chỉ: 3

BÀI GIẢNG ĐIỆN TỬ

Biên soạn bởi: PGS.TS. Dương Tuấn Anh

Khoa Khoa Học và Kỹ Thuật Máy Tính

Trường Đ.H. Bách Khoa

Đại học Quốc Gia Tp Hồ Chí Minh

Tài liệu tham khảo

[1] Cormen, T. H., Leiserson, C. E, and Rivest, R. L., *Introduction to Algorithms*, The MIT Press, 1997.

[2] Levitin, A., *Introduction to the Design and Analysis of Algorithms*, Addison Wesley, 2003

[3] Sedgewick, R., *Algorithms in C++*, Addison-Wesley, 1998

[4] Weiss, M.A., *Data Structures and Algorithm Analysis in C*, TheBenjamin/Cummings Publishing, 1993

Đề cương Môn học

1. Các khái niệm căn bản
2. Chiến lược chia-để-trị
3. Chiến lược giảm-để-trị
4. Chiến lược biến thể-để-trị
5. Qui hoạch động và giải thuật tham lam
6. Giải thuật quay lui
7. Vấn đề NP-đầy đủ
8. Giải thuật xấp xỉ

Môn học: Phân tích và thiết kế giải thuật

Chương 1

CÁC KHÁI NIỆM CĂN BẢN

Nội dung

1. Đệ quy và hệ thức truy hồi
2. Phân tích độ phức tạp giải thuật
3. Phân tích giải thuật lặp
4. Phân tích giải thuật đệ quy
5. Chiến lược thiết kế giải thuật
6. Thiết kế giải thuật kiểu “trực tiếp” (bruce-force)

1. Đệ quy

Hệ thức truy hồi

Thí dụ 1: Hàm tính giai thừa

N! = N.(N-1)! **với N ≥ 1**

0! = 1

Những định nghĩa hàm đệ quy mà chứa những đối số nguyên được gọi là những *hệ thức truy hồi* (*recurrence relation*).

```
function factorial (N: integer): integer;
begin
   if N = 0
   then factorial: = 1
   else factorial: = N*factorial (N-1);
end;
```

Hệ thức truy hồi

<u>Thí dụ 2</u>: Số Fibonacci

Hệ thức truy hồi:

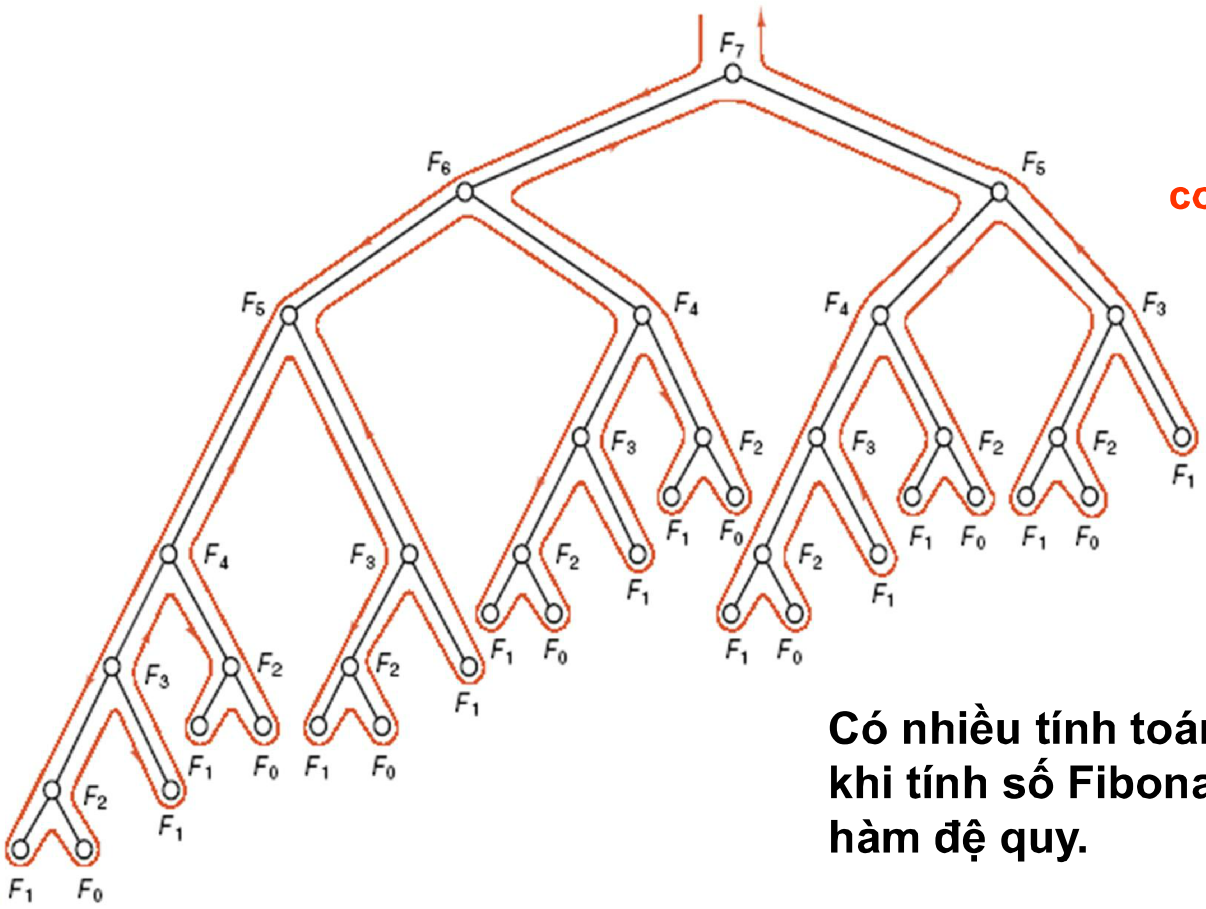
$F_N = F_{N-1} + F_{N-2}$ **for** $N \geq 2$

$F_0 = F_1 = 1$

1, 1, 2, 3, 5, 8, 13, 21, …

```
function fibonacci (N: integer): integer;
begin
  if N <= 1
  then fibonacci: = 1
  else fibonacci: = fibonacci(N-1) +
                                    fibonacci(N-2);
end;
```

Số Fibonacci – Cây đệ quy



computed

Có nhiều tính toán dư thừa khi tính số Fibonacci bằng hàm đệ quy.

Một cách khác: Ta có thể dùng một mảng để chứa những trị số đi trước trong khi tính hàm fibonacci. Ta có một giải thuật không đệ quy.

```
procedure fibonacci;
const max = 25;
var i: integer;
F: array [0..max] of integer;
begin
    F[0]: = 1; F[1]: = 1;
    for i: = 2 to max do
       F[i]: = F[i-1] + F[i-2]
end;
```

Giải thuật không đệ quy thường làm việc hữu hiệu và dễ kiểm soát hơn 1 giải thuật đệ quy.

Nhờ vào sử dụng stack, ta có thể chuyển đổi một giải thuật đệ quy thành một giải thuật lặp tương đương.

2. Phân tích độ phức tạp giải thuật

Với phần lớn các bài toán, thường có nhiều giải thuật khác nhau để giải một bài toán.
Làm cách nào để chọn giải thuật tốt nhất để giải một bài toán?
Làm cách nào để so sánh các giải thuật cùng giải được một bài toán?

Phân tích độ phức tạp của một giải thuật: dự đoán các tài nguyên mà giải thuật đó cần.

Tài nguyên: Chỗ bộ nhớ
Thời gian tính toán

Thời gian tính toán là tài nguyên quan trọng nhất.

Hai cách phân tích

Thời gian tính toán của một giải thuật thường là một hàm của kích thước dữ liệu nhập.

Chúng ta quan tâm đến:

- ***Trường hợp trung bình*** **(*average case*): thời gian tính toán mà một giải thuật cần đối với một "dữ liệu nhập thông thường" (typical input data).**
- ***Trường hợp xấu nhất*** **(*worst case*): thời gian tính toán mà một giải thuật cần đối với một "dữ liệu nhập xấu nhất"**

Khung thức của sự phân tích

♦ **Bước 1: Đặc trưng hóa dữ liệu nhập và quyết định kiểu phân tích thích hợp.**

Thông thường, ta tập trung vào việc

- chứng minh rằng thời gian tính toán luôn nhỏ hơn một "cận trên" (upper bound), hay

- dẫn xuất ra thời gian chạy trung bình đối với một dữ liệu nhập ngẫu nhiên.

♦ **Bước 2: nhận dạng thao tác trừu tượng (*abstract operation)* mà giải thuật dựa vào đó làm việc.**

Thí dụ: thao tác so sánh trong giải thuật sắp thứ tự.

Tổng số thao tác trừu tượng thường tùy thuộc vào một vài đại lượng.

♦ **Bước 3: thực hiện phân tích toán học để tìm ra các giá trị trung bình và giá trị xấu nhất của các đại lượng quan trọng.**

Hai trường hợp phân tích

- **Thường thì không khó để tìm ra cận trên của thời gian tính toán của một giải thuật.**
- **Nhưng phân tích trường hợp trung bình thường đòi hỏi một sự phân tích toán học cầu kỳ, phức tạp.**
- **Về nguyên tắc, một giải thuật có thể được phân tích đến một mức độ chính xác rất chi li. Nhưng trong thực tế, chúng ta thường chỉ tính ước lượng (*estimating*) mà thôi**
- **Tóm lại, chúng ta tìm kiếm một ước lượng thô về thời gian tính toán của một giải thuật (nhằm mục đích phân lớp độ phức tạp).**

Phân lớp độ phức tạp

Hầu hết các giải thuật thường có một thông số chính, N, số mẫu dữ liệu nhập mà được xử lý.

Thí dụ:
Kích thước của mảng (array) được sắp thứ tự hoặc tìm kiếm.
Số nút của một đồ thị.

Giải thuật có thể có thời gian tính toán tỉ lệ với

1. Nếu tác vụ chính được thực thi một vài lần.
⇒ thời gian tính toán là hằng số.

2. lgN (logarithmic) $\log_2 N \equiv \lg N$
Giải thuật tăng chậm hơn sự tăng của N.

3. N (linear)

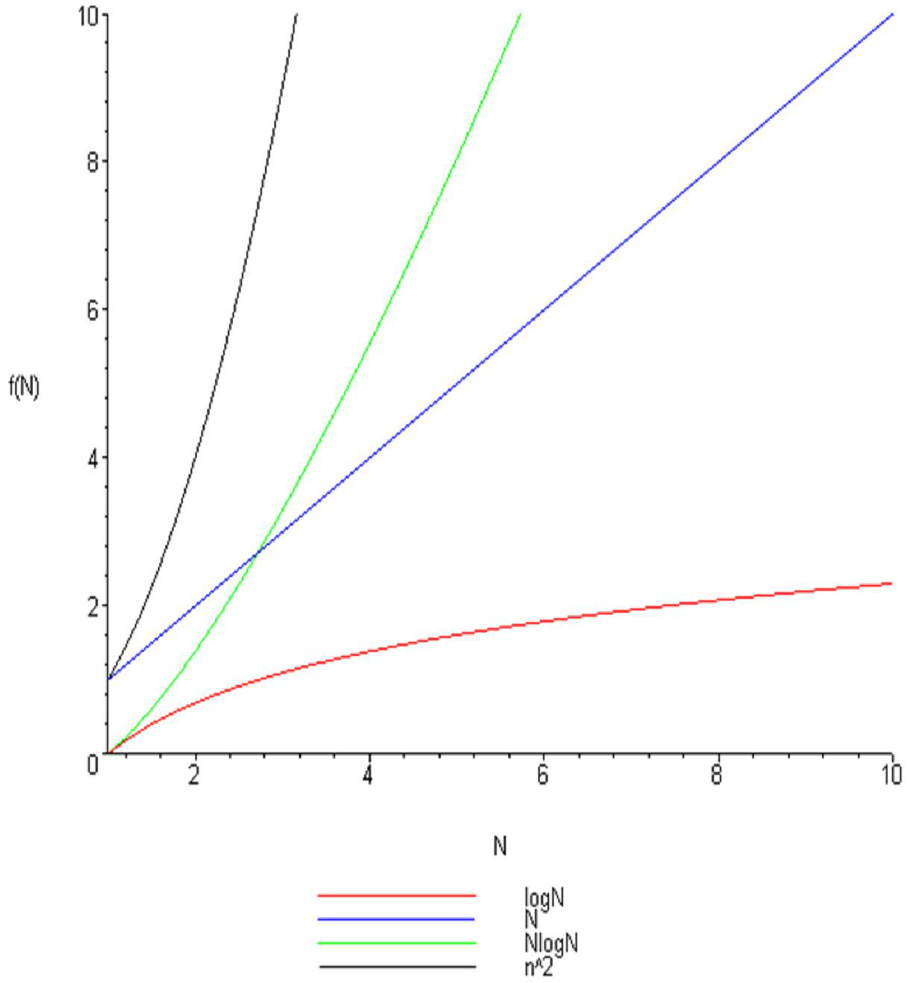
4. NlgN

5. N^2 (quadratic) khi giải thuật là vòng lặp lồng hai

6. N^3 (cubic) khi giải thuật là vòng lặp lồng ba

7. 2^N một số giải thuật có thời gian chạy luỹ thừa.

Một vài giải thuật khác có thể có thời gian chạy $N^{3/2}$, $N^{1/2}$, $(lgN)^2$ …


10
8
6
4
2
0
f(N)
2
4
6
8
10
N
logN
N
NlogN
n^2

Độ phức tạp tính toán

Chúng ta tập trung vào phân tích trường hợp xấu nhất. Khi phân tích, bỏ qua những thừa số hằng số để xác định sự phụ thuộc hàm của thời gian tính toán đối với kích thước dữ liệu nhập.

Thí dụ: Thời gian tính toán của sắp thứ tự bằng phương pháp trộn (*mergesort*) là tỉ lệ với NlgN.

Khái niệm "tỉ lệ với" (*proportional to*)

Công cụ toán học để làm chính xác khái niệm này là *ký hiệu – O* (*O-notation*).

Định nghĩa: Một hàm g(N) được gọi là O(f(N)) nếu tồn tại hai hằng số c_0 và N_0 sao cho g(N) nhỏ hơn c_0f(N) với mọi $N > N_0$.

Ký hiệu O

Ký hiệu O là một cách hữu ích để phát biểu cận trên về thời gian tính toán mà độc lập đối với đặc tính dữ liệu nhập và chi tiết hiện thực hóa.

Chúng ta cố gắng tìm cả "cận trên" lẫn "cận dưới" của thời gian tính toán trong phân tích trường hợp xấu nhất.

Nhưng cận dưới (*lower-bound*) thì thường khó xác định.

Phân tích trường hợp trung bình

Với kiểu phân tích này, ta phải

- đặc trưng hóa dữ liệu nhập của giải thuật

- tính giá trị trung bình của số lần một phát biểu được thực thi.

- tính thời gian tính toán trung bình của toàn giải thuật.

Nhưng thường thì khó

- xác định thời gian chạy của mỗi phát biểu.

- đặc trưng hóa chính xác dữ liệu nhập trong thực tế.

Các kết quả tiệm cận và xấp xỉ

Kết quả của một sự phân tích toán học thường mang tính xấp xỉ (*approximate*): nó có thể là một biểu thức gồm một chuỗi những số hạng giảm dần tầm quan trọng.

Ta thường để ý đến các số hạng dẫn đầu trong biểu thức toán học.

<u>Thí dụ</u>: Thời gian tính toán trung bình của một chương trình là:

$$a_0 N \lg N + a_1 N + a_2$$

Ta có thể viết lại là:

$$a_0 N \lg N + O(N)$$

Với N lớn, ta không cần tìm giá trị của a_1 hay a_2.

Các kết quả xấp xỉ

Ký hiệu O cho ta một cách tìm ra kết quả xấp xỉ khi N lớn.

Do đó, thông thường chúng ta có thể bỏ qua một số đại lượng khi có tồn tại một số hạng dẫn đầu trong biểu thức.

Example: nếu biểu thức là N(N-1)/2, chúng ta có thể bảo rằng nó khoảng chừng $N^2/2$.

3.Phân tích một giải thuật lặp

<u>Thí dụ 1</u> Cho một giải thuật tìm phần tử lớn nhất trong một mảng 1 chiều.

```
procedure MAX(A, n, max)
/* Set max to the maximum of A(1:n)  */
begin
   integer i, n;
   max := A[1];
   for i:= 2 to n do
     if A[i] > max then max := A[i]
 end
```

Nếu C(n) là độ phức tạp tính toán của giải thuật được tính theo thao tác so sánh (A[i]> max). Hãy xác định C(n) trong trường hợp xấu nhất.

Phân tích một giải thuật lặp (tt.)

Thao tác căn bản của thủ tục MAX là thao tác *so sánh*.

Tổng số thao tác so sánh của thủ tục MAX chính là số lần thân vòng lặp được thực thi: (n-1).

Vậy độ phức tạp tính toán của giải thuật là O(n).

Đây là độ phức tạp của cả hai trường hợp trung bình và xấu nhất.

<u>Ghi chú</u>: Nếu thao tác căn bản là *phát biểu gán* (max := A[i]) thì O(n) là độ phức tạp trong trường hợp xấu nhất.

Phân tích một giải thuật lặp (tt.)

<u>Thí dụ 2</u>: Giải thuật kiểm tra xem có phải mọi phần tử trong mảng 1 chiều là khác biệt nhau.

```
function UniqueElements(A, n)
begin
  for i:= 1 to n –1 do
     for j:= i + 1 to n do
        if A[i] = A[j] return false
  return true
end
```

Trong trường hợp xấu nhất, mảng không hề có hai phần tử nào bằng nhau hoặc mảng có hai phần tử cuối cùng bằng nhau. Lúc đó một sự so sánh diễn ra mỗi khi thân vòng lặp trong được thực hiện.

i = 1 **j chạy từ 2 cho đến n tức n – 1 lần so sánh**
i = 2 **j chạy từ 3 cho đến n tức n – 2 lần so sánh**
.
.
i = n -2 **j chạy từ n-1 cho đến n tức 2 lần so sánh**
i = n -1 **j chạy từ n cho đến n tức 1 lần so sánh**

Tóm lại, tổng số lần so sánh là:

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

Vậy độ phức tạp tính toán của giải thuật trong trường hợp xấu nhất là $O(n^2)$.

Phân tích một giải thuật lặp (tt.)

Thí dụ 3 (So trùng dòng ký tự - string matching): Tìm tất cả những sự xuất hiện của một khuôn mẫu (pattern) trong một văn bản (text).

Văn bản là một mảng T[1..n] gồm *n* ký tự và kiểu mẫu là một mảng P[1..m] gồm *m* ký tự.

Kiểu mẫu P xuất hiện với *độ dịch chuyển* (*shift*) *s* trong văn bản T (tức là, P xuất hiện bắt đầu từ vị trí *s+1* trong văn bản T) nếu $1 \le s \le n - m$ và $T[s+1..s+m] = P[1..m]$.

Một giải thuật đơn giản nhất để tìm tất cả những sự xuất hiện của P trong T sẽ dùng một vòng lặp mà kiểm tra điều kiện P[1..m] = T[s+1..s+m] với mỗi trị trong *n – m + 1* trị có thể có của *s*.

```
procedure NATIVE-STRING-MATCHING(T,P);
begin
    n: = |T|;     m: = |P|;
    for s:= 0 to n – m do
    if P[1..m] = T[s+1,..,s+m] then
        print “Pattern occurs with shift” s;
end
```

```
procedure NATIVE-STRING-MATCHING(T,P);
begin
    n: = |T|;     m: = |P|;
    for s:= 0 to n – m do
    begin
        exit:= false; k:=1;
        while k ≤ m and not exit do
          if P[k] ≠ T[s+k] then exit := true
          else k:= k+1;
        if not exit then
         print "Pattern occurs with shift" s;
     end
end
```

Giải thuật NAIVE STRING MATCHER có hai vòng lặp lồng nhau:

- **vòng lặp ngoài lặp *n – m + 1* lần.**
- **vòng lặp trong lặp tối đa *m* lần.**

Do đó, độ phức tạp của giải thuật trong trường hợp xấu nhất là:

O((n – m + 1)m).

4. Phân tích giải thuật đệ quy: các công thức truy hồi căn bản

Có một phương pháp căn bản để phân tích độ phức tạp của các giải thuật đệ quy.

Tính chất của một giải thuật đệ quy $\Rightarrow$ thời gian chạy đối với bộ dữ liệu nhập kích thước N tùy thuộc vào thời gian chạy của những bộ dữ liệu nhập nhỏ hơn.

Tính chất này được mô tả bằng một công thức toán học được gọi là *hệ thức truy hồi* (*recurrence relation*).

Để dẫn xuất ra độ phức tạp của một giải thuật đệ quy, chúng ta phải giải hệ thức truy hồi này.

Phân tích giải thuật đệ quy bằng phương pháp lặp

<u>Công thức 1</u>: 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:

$$C_N = C_{N-1} + N \qquad N \geq 2$$
$$C_1 = 1$$

Cách suy ra độ phức tạp bằng phương pháp lặp:

$$C_N = C_{N-1} + N$$
$$= C_{N-2} + (N-1) + N$$
$$= C_{N-3} + (N-2) + (N-1) + N$$
$$\vdots$$
$$= C_1 + 2 + \ldots + (N-2) + (N-1) + N$$
$$= 1 + 2 + \ldots + (N-1) + N$$
$$= N(N-1)/2$$
$$= N^2/2$$

Thí dụ 2

<u>Công thức 2</u>: Một chương trình đệ quy mà tách đôi bộ dữ liệu nhập trong một bước làm việc. Hệ thức truy hồi là:

$C_N = C_{N/2} + 1 \qquad N \geq 2$

$C_1 = 0$

Cách suy ra độ phức tạp:

Giả sử $N = 2^n$

$C(2^n) = C(2^{n-1}) + 1$

$= C(2^{n-2}) + 1 + 1$

$= C(2^{n-3}) + 3$

.

.

$= C(2^0) + n$

$= C_1 + n = n$

$C_N = n = \lg N$

$C_N \approx \lg N$

Thí dụ 3

<u>Công thức 3</u>. Một chương trình đệ quy mà tách 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

$$C_N = 2C_{N/2} + N \qquad \text{for} \quad N \geq 2$$
$$C_1 = 0$$

Cách suy ra độ phức tạp:

Assume $N = 2^n$

$C(2^n) = 2C(2^{n-1}) + 2^n$

$C(2^n)/2^n = C(2^{n-1})/ 2^{n-1} + 1$

$= C(2^{n-2})/ 2^{n-2} + 1 + 1$

.

.

$= n$

$\Rightarrow C(2^n) = n.2^n$

$C_N = NlgN$

$C_N \approx NlgN$

Thí dụ 4

Công thức 4. Một chương trình đệ quy mà tách đôi dữ liệu nhập thành hai nửa trong một bước làm việc . Hệ thức truy hồi là

$$C(N) = 2C(N/2) + 1 \qquad \text{for } N \geq 2$$
$$C(1) = 0$$

Cách suy ra độ phức tạp:

Giả sử $N = 2^n$.

$$C(2^n) = 2C(2^{n-1}) + 1$$
$$C(2^n)/ 2^n = 2C(2^{n-1})/ 2^n + 1/2^n$$
$$= C(2^{n-1})/ 2^{n-1} + 1/2^n$$
$$= [C(2^{n-2})/ 2^{n-2} + 1/2^{n-1}] + 1/2^n$$
$$\cdot$$
$$\cdot$$
$$\cdot$$
$$= C(2^{n-i})/ 2^{n-i} + 1/2^{n-i} + 1 + \ldots + 1/2^n$$

Cuối cùng, khi i = n -1, ta được:

$$C(2^n)/2^n = C(2)/2 + ¼ + 1/8 + \ldots + 1/2^n$$
$$= ½ + ¼ + \ldots. + 1/2^n$$
$$\approx 1$$
$$\Rightarrow \quad C(2^n) = 2^n$$

$$C(N) \approx N$$

Một số hệ thức truy hồi có vẻ giống nhau nhưng mức độ khó khi giải chúng để tìm độ phức tạp thì có thể rất khác nhau.

Nguyên tắc phân tích độ phức tạp trung bình

Để tính độ phức tạp trung bình của một giải thuật A, ta phải làm một số bước:

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 là S = { I_1, I_2,…, I_k}**
2. **Ta phải định nghĩa một phân bố xác xuất *p* trên S mà biểu diễn mức độ chắc chắn mà dữ liệu đầu vào đó có thể xảy ra.**
3. **Ta phải tính tổng số tác vụ căn bản được giải thuật A thực hiện để xử lý một trường hợp mẫu. Ta dùng v(I_k) ký hiệu tổng số tác vụ được thực hiện bởi A khi dữ liệu đầu vào thuộc trường hợp I_k.**

Phân tích độ phức tạp trung bình (tt.)

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

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

Thí dụ: Cho một mảng A có *n* phần tử. Tìm kiếm vị trí mà trị X xuất hiện trong mảng A.

```
begin
   i := 1;
  while i <= n and X <> A[i] do
  i := i+1;
end
```

Thí dụ: Tìm kiếm tuần tự

Giả sử X có xuất hiện trong mảng và giả định rằng xác xuất để nó xuất hiện tại một vị trí bất kỳ trong mảng là đều nhau và xác xuất 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:

$$C(n) = 1.(1/n) + 2.(1/n) + \ldots + N.(1/n)$$
$$= (1 + 2 + \ldots + n).(1/n)$$
$$= (1+2+\ldots+n)/n = n(n+1)/2.(1/n) = (n+1)/2.$$

Vài chuỗi số thông dụng

Có một vài chuỗi số thông dụng trong việc phân tích độ phức tạp giải thuật.

- **Chuỗi số cọng**

$S_1 = 1 + 2 + 3 + \ldots + n$

$S_1 = n(n+1)/2 \approx n^2/2$

$S_2 = 1 + 2^2 + 3^2 + \ldots + n^2$

$S_2 = n(n+1)(2n+1)/6 \approx n^3/3$

- **Chuỗi số nhân**

$S = 1 + a + a^2 + a^3 + \ldots + a^n$

$S = (a^{n+1} - 1)/(a-1)$

If $0 < a < 1$, then

$S \leq 1/(1-a)$

Và khi $n \rightarrow \infty$, S tiến về $1/(1-a)$.

Vài chuỗi số thông dụng (tt.)

- **Tổng chuỗi số điều hoà (Harmonic sum)**

$H_n = 1 + ½ + 1/3 + ¼ + \ldots + 1/n$

$H_n = \log_e n + \gamma$

$\gamma \approx 0.577215665$ được gọi là *hằng số Euler*.

Một chuỗi số khác cũng rất thông dụng khi phân tích các thao tác làm việc trên cây nhị phân:

$1 + 2 + 4 + \ldots + 2^{m-1} = 2^m - 1$

5. Chiến lược thiết kế giải thuật

- **Một *chiến lược thiết kế giải thuật* (Algorithm Design Strategy) là một cách tiếp cận tổng quát để giải quyết vấn đề bằng giải thuật mà có thể áp dụng cho nhiều bài toán khác nhau trong nhiều lãnh vực khác nhau.**
- **Việc học những chiến lược thiết kế này hết sức quan trọng vì những lý do sau:**
  - Chúng cung cấp những chỉ dẫn để thiết kế giải thuật cho những bài toán mới.
  - Giải thuật đóng một vai trò quan trọng trong khoa học máy tính. Dựa vào các chiến lược thiết kế giải thuật, ta có thể phân loại giải thuật dựa vào ý tưởng thiết kế nền tảng của chúng.

Chiến lược thiết kế giải thuật (tt.)

- "Chia-để-trị" là một ví dụ điển hình của một chiến lược thiết kế giải thuật.
- Ngoài ra còn có nhiều chiến lược thiết kế giải thuật nổi tiếng khác
- Tập hợp những chiến lược thiết kế giải thuật tạo thành một bộ công cụ rất mạnh có sẵn giúp chúng ta nghiên cứu và xây dựng giải thuật.
- Một chiến lược thiết kế giải thuật sẽ được đề cập ngay trong chương này là chiến lược thiết kế *kiểu "trực tiếp"* (bruce-force)

Chiến lược thiết kế giải thuật "trực tiếp" (bruce-force approach)

- Thiết kế giải thuật theo lối "trực tiếp" là thiết kế giải thuật một cách đơn giản, chân phương dựa trực tiếp vào sự phát biểu bài toán và những định nghĩa về các khái niệm liên quan.
- "Just do it" là một cách khác để mô tả chiến lược thiết kế này.
- Giải thuật thiết kế theo lối "trực tiếp" là loại giải thuật dễ hiểu nhất và dễ hiện thực nhất.
- Tìm kiếm tuần tự (sequential search) là thí dụ điển hình của kiểu thiết kế bruce-force.
- Selection sort, NAÏVE-STRING-MATCHER (so trùng dòng ký tự) là những thí dụ khác của lối thiết kế bruce-force.

- Mặc dù đơn sơ và không tinh xảo, nhưng những giải thuật thuộc loại bruce-force vẫn không nên xem thường, hoặc bỏ qua vì những lý do sau:
  - Giải thuật bruce-force thường có khả năng áp dụng rộng rãi.
  - Với một số bài toán quan trọng, những giải thuật bruce-force có những giá trị thực tế nhất định.
  - Những giải thuật tinh xảo thường khó hiểu và khó hiện thực hơn những giải thuật bruce-force.
  - Giải thuật bruce-force có ích trong việc giảng dạy, dùng làm thước đo để đánh giá những cách khác hữu hiệu hơn để giải cùng một vấn đề.