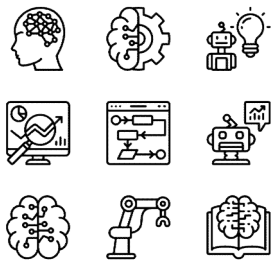
Computer Science for Practicing Engineers

# Các khái niệm trong thuật toán



TS. Huỳnh Bá Diệu
Email: dieuhb@gmail.com
Phone: 0914146868

## Các khái niệm trong Thuật toán (Algorithm)

**Nội dung**

1. Thuật toán
2. Các tính chất
3. Đánh giá thuật toán

## Thuật toán (Algorithm)

- Định nghĩa

Tập câu lệnh (hay chỉ thị) *xác định*, *có thứ tự* nhằm giải bài toán đặt ra.

Ví dụ:

Thuật toán nấu cơm:

1. Rửa nồi
2. Đong và vo gạo
3. Cắm điện, bật nút nguồn
4. Chờ kiểm tra cho đến lúc cơm chín.

## Thuật toán (Algorithm)

Thuật toán nấu cơm:

1. Rửa nồi
2. Đong và vo gạo
3. Cắm điện, bật nút nguồn
4. Chờ kiểm tra cho đến lúc cơm chín.

Có thể đổi bước 1 và bước 3 được hay không?

## Thuật toán (Algorithm)


Muhammad ibn Musa al-Khwarizmi

Thuật toán: là tập hợp các lệnh (chỉ thị) xác định, có thứ tự, nhằm giải bài toán đặt ra.

Ví dụ thuật toán hoán vị hai số a, b

B1: a= a+b;

B2: b= a-b;

B3: a= a-b;

a= 7, b=12
B1: a= 7+ 12= 19
B2: b= 19 – 12= 7
B3: a= 19 -7= 12

## Thuật toán (Algorithm)

Thuật toán: là tập hợp các lệnh (chỉ thị) xác định, ***có thứ tự***, nhằm giải bài toán đặt ra.

Ví dụ thuật toán hoán vị hai số a, b

B1: a= a+b;

B2: b= a-b;

B3: a= a-b;

a= 7, b=12
B1: a= 7+ 12= 19
B2: b= 19 – 12= 7
B3: a= 19 -7= 12

Không thay đổi **thứ tự** thực hiện các bước thuật toán

## Thuật toán (Algorithm)

Hoán vị hai số a, b

B1: a= a+b;

B2: b= a-b;

B3: a= a-b;

**Giả sử cần hoán vị hai số nguyên trong một chương trình máy tính. Thuật toán trên có thể sai không? Vì sao?**

## Thuật toán (Algorithm)

Thuật toán: là tập hợp các lệnh (chỉ thị) xác định, ***có thứ tự***, nhằm giải bài toán đặt ra.

Các lệnh chứa các biến, biểu thức độc lập nhau thì có thể thay đổi thứ tự.
Ví dụ: Tính độ lệch trung bình các giá trị của các phần tử nửa trước và nửa sau trong mảng.
- Có thể tính trung bình nửa trước hoặc nửa sau độc lập nhưng phải tính trước khi tính độ lệch

## Thuật toán (Algorithm)

Các tính chất của thuật toán:

Tính đúng: Thuật toán phải giải được bài toán đặt ra.

Tính dừng: Thuật toán phải dừng sau một số hữu hạn các bước

Tính xác định: Các bước của thuật toán phải rõ ràng, xác định. Với một giá trị đầu vào, nếu thực hiện đúng theo các bước của thuật toán thì ta có duy nhất 1 đầu ra.

Tính phổ quát: Khi xây dựng thuật toán phải giải cho 1 lớp các bài toán (ví dụ giải phương trình bậc 2 có thể giải cho bất kỳ phương trình bậc 2 nào)

……

## Thuật toán (Algorithm)

- Đánh giá thuật toán

Một bài toán (problem) có thể có nhiều cách giải khác nhau. Để so sánh giữa các cách giải ta thường dựa vào độ phức tạp của thuật toán.

## Độ phức tạp của Thuật toán (algorithm complexity)

Độ phức tạp của thuật toán (algorithm complexity)

- Độ phức tạp theo không gian: Số ô nhớ cần thiết để thuật toán có thể thực hiện được.
- Độ phức tạp theo thời gian: Thời gian thực hiện xong các lệnh và trả về kết quả kể từ lúc nhận dữ liệu vào.

Thông thường khi nói đến độ phức tạp ta chỉ nói đến độ phức tạp theo thời gian.

Ký hiệu độ phức tạp là T(n) với n là kích thước dữ liệu vào.

Nếu kích thước dữ liệu càng lớn thì thời gian thực hiện thuật toán để trả về kết quả xử lý càng lớn (!!! Không tỉ lệ thuận)

## Độ phức tạp của Thuật toán (algorithm complexity)

Các nguyên lý xác định độ phức tạp:

+ Nguyên lý cộng

+ Nguyên lý nhân

## Độ phức tạp của Thuật toán (algorithm complexity)

Xác định độ phức tạp của thuật toán

**Nguyên lý cộng:** Nếu công việc A được chia làm hai công việc con rời nhau $A_1$ và $A_2$, trong đó công việc $A_1$ có độ phức tạp f1(n), công việc $A_2$ có độ phức tạp f2(n) thì độ phức tạp của công việc A là f1(n) + f2(n).

Dùng ký pháp O (big Omega)

T(n) = O(f1(n) + f2(n)) = O(**max(f1(n), f2(n))** )

Ví dụ **sắp xếp** mảng n phần tử theo thuật toán sắp xếp nổi bọt rồi in.

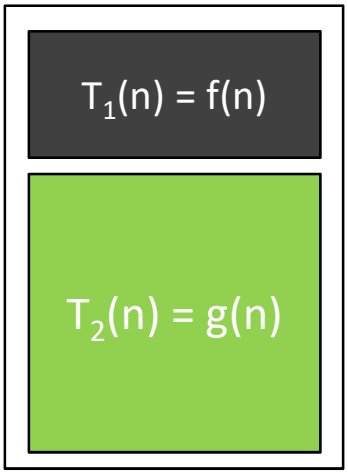
+ Sắp xếp: $O(n^2)$

+ In mảng: O(n)

## Các nguyên lý tính độ phức tạp

Nguyên lý cộng

Cho thuật toán A được chia làm hai đoạn con là $A_1$ và $A_2$. Đoạn con $A_1$ có độ phức tạp là $T_1(n) = f(n)$, đoạn con $A_2$ có độ phức tạp là $T_2(n) = g(n)$. Khi đó độ phức tạp của thuật toán A là $T(n)= f(n)+ g(n) = O(max(f(n), g(n)))$




Ví dụ: Cho a gồm n số nguyên. Hãy sắp xếp mảng A và in mảng đã sắp xếp ra màn hình.

1. Sắp xếp: $(n^2 - n)/2$
2. In: n

Vậy độ phức tạp $T(n) = O(max((n^2 - n)/2, n)) = O((n^2 - n)/2) = O(n^2)$

14

# Các nguyên lý tính độ phức tạp

Xác định độ phức tạp của thuật toán

**Nguyên lý nhân**: Nếu công việc A có f1(n) lần gọi hàm X, trong đó hàm X có độ phức tạp là g(n) thì độ phức tạp của công việc A là f1(n) * g(n)

T(n) = O(f1(n)*g(n))

Ví dụ: In ra tổng từng hàng của ma trận A (m hàng, n cột) thì độ pt là O(m*n)

```
void tinhtong()
{
    for(int i=0; i<m; i++)                    // O(m)
        S.O.P ("\n :" + tonghang(i));                  // O(n)
}
```

# Các nguyên lý tính độ phức tạp

Nguyên lý nhân

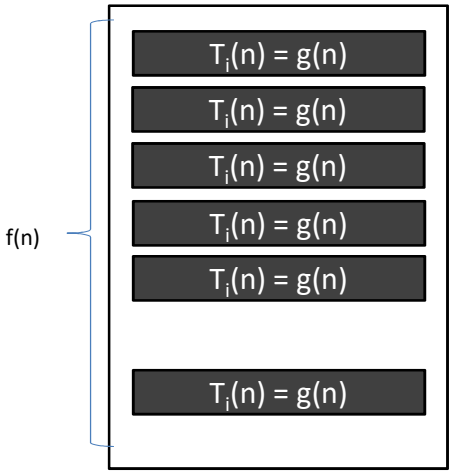
Thuật toán A gồm f(n) lời gọi câu lệnh (hàm), trong đó mỗi lệnh có độ phức tạp là = g(n). Khi đó độ phức tạp của thuật toán A là T(n)= O(f(n)* g(n))

*Ví dụ: Cho dãy A gồm n số. Cần in số các phần tử trong dãy nhỏ hơn A[i]*

```
for(i=0; i<n; i++)
    demnhohon(A[i]);
```

a = { 3, 2, 7, 6, 9, 8, 5 }

1, 0, 4, 3, 6, 5, 2



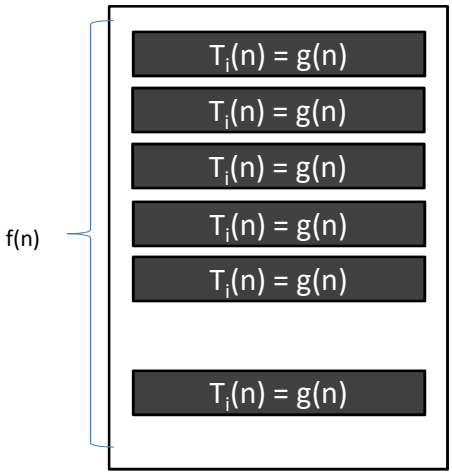

16

# Các nguyên lý tính độ phức tạp

*Ghi chú:*
*Nếu nhiều vòng lặp for lồng nhau thì lấy tích.*

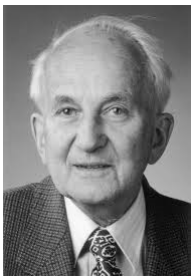
```
for(int i=0; i<n-1; i++)
  for(int j=i+1; j<n; j++)
        if(a[i]>a[j]) hoanvi(a[i], a[j])
```

**T(n) = O(n²)**




17

**Lothar Collatz,**
July 6, 1910 – September 26, 1990



Một số bài toán không thể xác định được số lần thực hiện.

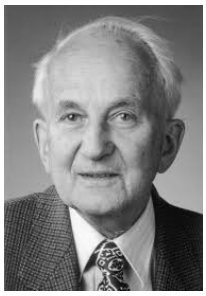
**Giả thiết Collatz (1937)**

Nếu số đó chẵn, ban chia số đó cho 2.
Nếu số đó là số lẻ, nhân số đó với 3 rồi cộng thêm 1.

N = 3; 10, 5, 16, 8, 4, 2, **1**
N = 4; 2, **1**
N = 5; 16, 8, 4, 2, **1**
N = 6; 3, 10, 5, 16, 8, 4, 2, **1**
N = 7; 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, **1**

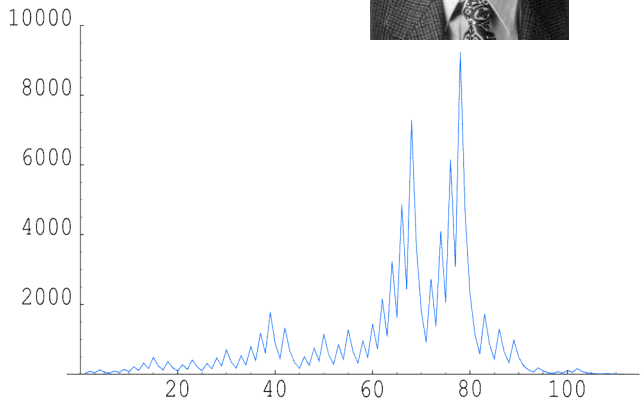
**Cho biết trường hợp n=27**
**N=27:???**

**Lothar Collatz,**
July 6, 1910 – September 26, 1990



The sequence for n = 27, listed and graphed below, takes 111 steps (41 steps through odd numbers, in large font), climbing to a high of 9232 before descending to 1.

27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1




## Các nguyên lý xác định độ phức tạp

Khi so sánh hai thuật toán thì ta bỏ qua các yếu tố như:

- Ngôn ngữ lập trình
- Yếu tố phần cứng
- Kỹ thuật cài đặt thuật toán
- Vào ra dữ liệu
- Hệ điều hành

→ Số lệnh cần phải thực hiện để giải xong bài toán và cho ra kết quả

## Độ tăng của hàm

Giả sử có 4 thuật toán cùng giải quyết bài toán và có độ phức tạp thời gian tương ứng với kích thước dữ liệu n lần lượt là :

**$100n$ , $n^2$, $n^3/2$, $2^n$**

Nếu có **1000 s** để giải quyết bài toán.

Thì thuật toán 1 có thể giải với n= 10

Thì thuật toán 2 có thể giải với n= 31

Thì thuật toán 3 có thể giải với n= 12

Thì thuật toán 4 có thể giải với n= 9

| n | 100n | $n^2$ | $n^3/2$ | $2^n$ |
|---|---|---|---|---|
| 1 | 100 | 1 | 1 | 2 |
| 2 | 200 | 4 | 4 | 4 |
| 3 | 300 | 9 | 13 | 8 |
| 5 | 500 | 25 | 62 | 32 |

## Độ tăng của hàm

Nếu cho thêm thời gian lên 10 lần: 10000

**Thì thuật toán 1 có thể giải với n= ?**

**Thì thuật toán 2 có thể giải với n=?**

**Thì thuật toán 3 có thể giải với n= ?**

**Thì thuật toán 4 có thể giải với n= ?**

$100n$ , $n^2$ , $n^3/2$, $2^n$

## Độ tăng của hàm

Nếu cho thêm thời gian lên 10 lần: 10000

**Thì thuật toán 1 có thể giải với n= 100 (10x)**

**Thì thuật toán 2 có thể giải với n= 100 (3x)**

**Thì thuật toán 3 có thể giải với n= 27 (2.3x)**

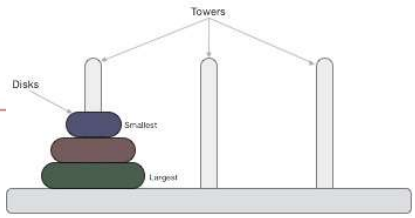
**Thì thuật toán 4 có thể giải với n= 13 (1.4x)**

$100n$ , $n^2$ , $n^3/2$, $2^n$

## Độ tăng của hàm

**Bằng cách theo dõi độ tăng của hàm, ta biết được kích thước bài toán tối đa có thể giải.**

## Độ tăng của hàm




Ví dụ: Bài toán Tháp Hà Nội

Cho 3 cột A, B, C. Ở cột A có n đĩa. Yêu cầu chuyển n đĩa từ A sang cột C. Mỗi lần chuyển 1 đĩa, đĩa nhỏ phải ở trên đĩa lớn. Quá trình chuyển có thể dùng cột trung gian.

Ví dụ chuyển 3 đĩa.

Chuyển 1 đĩa từ A sang C; chuyển đĩa từ A sang B; chuyển 1 đĩa từ C sang B; Chuyển 1 đĩa từ A sang C; Chuyển 1 đĩa từ B sang A; Chuyển 1 đĩa từ B sang C; Chuyển 1 đĩa từ A sang C;

## Độ tăng của hàm

Có một năm để chuyển đĩa thì số đĩa có thể chuyển tối đa là bao nhiêu?

**+ Số giây 1 năm bằng = ???**

**+ Số lần chuyển đĩa có thể chuyển trong 1 năm= ???**

**+ Số đĩa tối đa có thể chuyển là=???**

## Độ tăng của hàm

**Bằng cách theo dõi độ tăng của hàm, ta biết được kích thước bài toán tối đa có thể giải.**

Giả sử mỗi một giây có thể 1000 lần chuyển đĩa.

31536000000 = 365 * 24 * 60 * 60 * 1000

Số đĩa tối đa có thể chuyển là: $\log_2 (31536000000+1) = 34$

Số lần chuyển đĩa của bài toán tháp Hà Nội là $2^n - 1$

Độ phức tạp của thuật toán tháp Hà Nội là: $O(2^n)$

## Độ phức tạp không gian (Bộ nhớ)

Có thể dùng ký pháp O để biểu diễn số ô nhớ cần cho thuật toán.

```
for(i=1; i<=n; i++)
  for(j=i; j<n; j++)
   {
        // lệnh
        store val;
        // lệnh
  }
```

Ví dụ n=50, thì cần bao nhiêu ô nhớ?

i = 1 j = 1 → 49 : 49

i = 2 j = 2 → 49 : 48

i = 3 j = 3 → 49 : 47

- - -

i = 49 j = 49 → 49 : 1

i = 50 j = 50 → 49 : 0

Số ô nhớ = $1 + 2 + \ldots + 49 = \frac{49 \cdot 50}{2} = 1225$

## Độ phức tạp không gian (Bộ nhớ)

Nếu có thuật toán cần $2^n$ ô nhớ và n= 256.

Giả sử rằng dùng 1 electron để lưu 1 bit thì số electron cần là ???

## Độ phức tạp không gian (Bộ nhớ)

Nếu có thuật toán cần $2^n$ ô nhớ và n= 256.

Giả sử rằng dùng 1 electron để lưu 1 bit thì số electron cần là ???

---→Dùng hết số electron trong vũ trụ ($10^{130}$ ) cũng không có thể triển khai thuật toán.

## Độ phức tạp không gian (Bộ nhớ)

Ví dụ: Cho file F chứa các số nguyên. Hãy sắp xếp các số trong file F và ghi ra file kết quả F'.

Giải pháp:

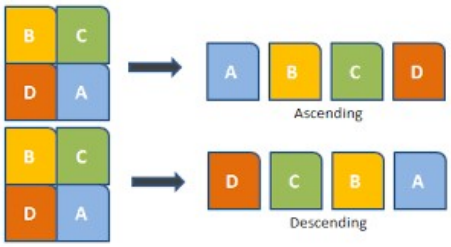
- Đọc dữ liệu từ file F và lưu vào mảng A
- Sắp xếp mảng A
- Ghi mảng A ra file kết quả F'.

## Độ phức tạp không gian (Bộ nhớ)

Ví dụ: Cho file F chứa các số nguyên. Hãy sắp xếp các số trong file F và ghi ra file kết quả F'.

Giải pháp:

- Đọc và lưu vào mảng
- Sắp xếp mảng
- Ghi ra file kết quả.

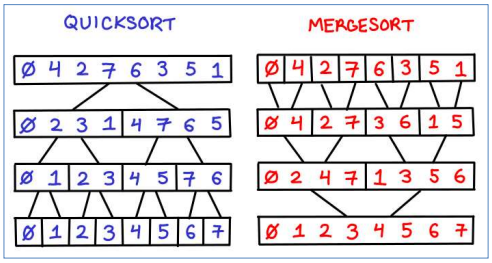



**Đánh giá của các bạn về giải pháp này???**

## Độ phức tạp không gian (Bộ nhớ)

Giải pháp sắp xếp các số trong file F và ghi ra file kết quả F'.

- Đọc và lưu vào mảng
- Sắp xếp mảng
- Ghi ra file kết quả.



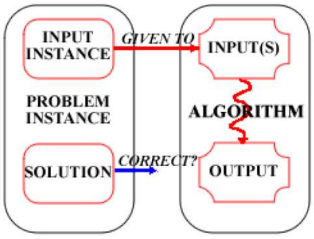

Giả sử các bạn có file gồm 10 tỉ số nguyên. Giải pháp trên có thực hiện được không?

## Tính đúng đắn của thuật toán

Cách ngăn chặn và tìm lỗi

Các phương pháp kiểm thử

- hộp đen
- hộp trắng
- hộp xám




Here is a list of classes of functions that are commonly encountered when analyzing the running time of an algorithm. In each case, *c* is a constant and *n* increases without bound. The slower-growing functions are generally listed first.

| Notation | Name | Example |
|---|---|---|
| $O(1)$ | constant | Determining if a binary number is even or odd; Calculating $(-1)^n$; Using a constant-size lookup table |
| $O(\log \log n)$ | double logarithmic | Number of comparisons spent finding an item using interpolation search in a sorted array of uniformly distributed values |
| $O(\log n)$ | logarithmic | Finding an item in a sorted array with a binary search or a balanced search tree as well as all operations in a Binomial heap |
| $O(\log^c n),\ c > 1$ | polylogarithmic | Matrix chain ordering can be solved in polylogarithmic time on a Parallel Random Access Machine. |
| $O(n^c),\ 0 < c < 1$ | fractional power | Searching in a kd-tree |
| $O(n)$ | linear | Finding an item in an unsorted list or a malformed tree (worst case) or in an unsorted array; adding two *n*-bit integers by ripple carry |
| $O(n \log^* n)$ | n log-star n | Performing triangulation of a simple polygon using Seidel's algorithm, or the union–find algorithm. Note that $\log^*(n) = \begin{cases} 0, & \text{if } n \leq 1 \\ 1 + \log^*(\log n), & \text{if } n > 1 \end{cases}$ |
| $O(n \log n) = O(\log n!)$ | linearithmic, loglinear, or quasilinear | Performing a fast Fourier transform; heapsort, quicksort (best and average case), |
| $O(n^2)$ | quadratic | Multiplying two *n*-digit numbers by a simple algorithm; bubble sort (worst case or naive implementation), Shell sort, quicksort (worst case), selection sort or insertion sort |
| $O(n^c),\ c > 1$ | polynomial or algebraic | Tree-adjoining grammar parsing; maximum matching for bipartite graphs |
| $L_n[\alpha, c],\ 0 < \alpha < 1 = e^{(c+o(1))(\ln n)^\alpha (\ln \ln n)^{1-\alpha}}$ | L-notation or sub-exponential | Factoring a number using the quadratic sieve or number field sieve |
| $O(c^n),\ c > 1$ | exponential | Finding the (exact) solution to the travelling salesman problem using dynamic programming; determining if two logical statements are equivalent using brute-force search |
| $O(n!)$ | factorial | Solving the traveling salesman problem via brute-force search; generating all unrestricted permutations of a poset; finding the determinant with expansion by minors; enumerating all partitions of a set |
| $O(n \cdot n!)$ | n × n factorial | Attempting to sort a list of elements using the incredibly inefficient bogosort algorithm (average case). |

The statement $f(n) = O(n!)$ is sometimes weakened to $f(n) = O\left(n^n\right)$ to derive simpler formulas for asymptotic complexity. For any $k > 0$ and $c > 0$, $O(n^c (\log n)^k)$ is a subset of $O(n^{c+\varepsilon})$ for any $\varepsilon > 0$, so may be considered as a polynomial with some bigger order.

# Ví dụ xác định độ phức tạp của thuật toán

**Cho biết độ phức tạp của thuật toán sau (1)?**

Thuật toán tạo ra ma trận đơn vị A cấp *n*.

```
(1)  for (i = 0 ; i < n ;  i++)
(2)        for (j = 0 ; j < n ; j++)
(3)                  A[i][j] = 0;

(4)  for (i = 0 ; i < n ; i++)
(5)        A[i][i] = 1;
```

$n^2$ + $n$ → $O(n^2)$

# Ví dụ xác định độ phức tạp của thuật toán

**Cho biết độ phức tạp của thuật toán sau (2)?**

Thuật toán tạo ra ma trận đơn vị A cấp *n*.

```
(1)  for (i = 0 ; i < n ;  i++)
(2)        for (j = 0 ; j < n ; j++)
(3)              if (i == j)
(4)                    A[i][j] = 1;
(5)              Else
(6)                    A[i][j] = 0;
```

1 → $n^2$ → $O(n^2)$

## Ví dụ xác định độ phức tạp của thuật toán

**Cho biết độ phức tạp của thuật toán sau (3)?**

```
1)  sum  = 0;
2)  for ( i = 0; i < n; i + +)
3)    for ( j = i + 1; j < = n; j + +)
4)      for ( k = 1; k < 10; k + +)
5)          sum = sum + i * j * k  ;
```

## Ví dụ xác định độ phức tạp của thuật toán

**Cho biết độ phức tạp của thuật toán sau (4)?**

```
1)  sum  = 0;
2)  for ( i = 0; i < n; i + +)
3)    for ( j = i + 1; j < = n; j + +)
4)      for ( k = 1; k < m; k + +) {
5)          x = 2*y;
6)          sum = sum + i * j * k  ;
7)      }
```

## Ví dụ xác định độ phức tạp của thuật toán

a. Give an analysis of the running time (Big-Oh will do).
b. Implement the code in Java, and give the running time for several values of $N$.
c. Compare your analysis with the actual running times.

(1)
```
sum = 0;
for( i = 0; i < n; i++ )
    sum++;
```

(2)
```
sum = 0;
for( i = 0; i < n; i++ )
    for( j = 0; j < n; j++ )
        sum++;
```

(3)
```
sum = 0;
for( i = 0; i < n; i++ )
    for( j = 0; j < n * n; j++ )
        sum++;
```

(4)
```
sum = 0;
for( i = 0; i < n; i++ )
    for( j = 0; j < i; j++ )
        sum++;
```

## Thuật toán (Algorithm)

Bài 1:

Cho dãy A có n phần tử. Hãy xoay dãy A về bên trái k lần.

Ví dụ: **A= {0 1 2 3 4 5 6 7 8 9}** và k =3 thì kết quả sau khi xoay là:

**A= {3 4 5 6 7 8 9 0 1 2 }**

**A= {0 1 2 3 4 5 6 7 8 9}**

**A1= {1 2 3 4 5 6 7 8 9 0}**

**A2= {2 3 4 5 6 7 8 9 0 1}**

**A3= {3 4 5 6 7 8 9 0 1 2}**

# Thuật toán (Algorithm)

Bài 1: Cho dãy A có n phần tử. Hãy xoay dãy A về bên trái k lần.

**Giải pháp 1:**

```
Lặp từ 1 đến k
{
    int t= a[0];      // lấy phần tử ở ô đầu tiên ra
    //Dịch các phần tử a[1] đến a[n-1] về phía trước 1 ô.

    a[n-1]=t;// gán phần tử ở ô cuối =t
}
```

**A= {0 1 2 3 4 5 6 7 8 9}**

**A'= {3 4 5 6 7 8 9 0 1 2 }**

# Thuật toán (Algorithm)

Bài 1: Cho dãy A có n phần tử. Hãy xoay dãy A về bên trái k lần.

**Giải pháp 1:**

```
for (int ll=1; ll<=k; ll++)
 {
        int t= a[0];
       //Dịch các phần tử a[1] đến a[n-1] về phía trước 1 ô.
       for(int i=0; i<n-1; i++) a[i]= a[i+1];
       a[n-1]= t;
 }
```

**Độ phức tạp của thuật toán là O(kn)**

## Thuật toán (Algorithm)

Bài 1: Cho dãy A có n phần tử. Hãy xoay dãy A về bên trái k lần.

**Giải pháp 2: Dùng phương pháp đảo mảng**

**A= {0 1 2 3 4 5 6 7 8 9}**

**Đảo 1: Đảo từ 0→ k-1 : {2 1 0 3 4 5 6 7 8 9} O(k)**

**Đảo 2: Đảo từ k → n-1 : {2 1 0 9 8 7 6 5 4 3} O(n-k)**

**Đảo 3: Đảo từ 0 → n-1 : {3 4 5 6 7 8 9 0 1 2} O(n)**

**Độ phức tạp của thuật toán là T(n) = O(max(k, n-k, n))= O(n)**

## Thuật toán (Algorithm)

Bài 1: Cho dãy A có n phần tử. Hãy xoay dãy A về bên trái k lần.

**Giải pháp 2: Dùng phương pháp đảo mảng**

```
void rev(int l, int r)
  {
     int i=l, j=r;
     while(i<j) {int t=a[i]; a[i] =a[j]; a[j]=t; i++; j--;}
  }
  void solution2(int k)
  {
     rev(0, k-1);
     rev(k, a.length-1);
     rev(0, a.length-1);
  }
```

## Thuật toán (Algorithm)

**Giải pháp 2: Dùng phương pháp đảo mảng**

```
public static void main(String[] args) {
    DICH_MANG_VE_TRAI_K_LAN m= new DICH_MANG_VE_TRAI_K_LAN();
    long t1, t2;
    m.sinhmang(10000000);
    t1= System.currentTimeMillis(); m.solution2(400000);  t2= System.currentTimeMillis();
    System.out.println("\n Thoi gian thuc hien theo giai phap 2=" + (t2-t1));
    m.sinhmang(10000000);
    t1= System.currentTimeMillis();   m.solution1(400000);   t2= System.currentTimeMillis();
    System.out.println("\n Thoi gian thuc hien theo giai phap 1=" + (t2-t1));
 }
```

## Thuật toán (Algorithm)

**So sánh thời gian hai giải pháp**

**Thoi gian thuc hien theo giai phap 2=30**

**Thoi gian thuc hien theo giai phap 1=5630107**

**BUILD SUCCESSFUL (total time: 93 minutes 50 seconds)**

## Bài tập về nhà

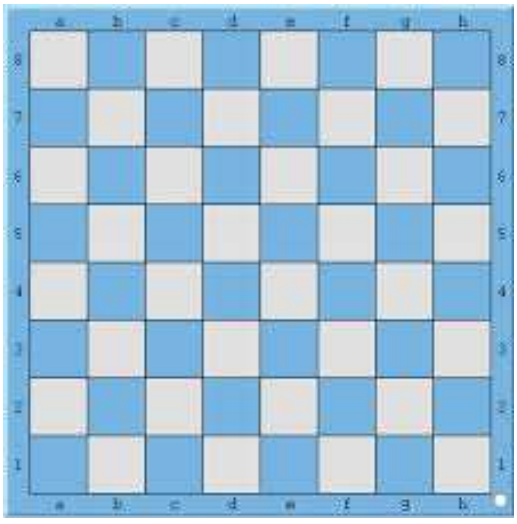
Xây dựng chương trình xoay mảng. Ghi các thông số của máy tính thực hiện chương trình, thực hiện chương trình theo hai phương pháp rồi điền vào bảng sau:

| **Tốc độ CPU:** | | **RAM:** | |
|---|---|---|---|
| | | | |
| Giá trị n | Giá trị k | Thời gian PP1 | Thời gian PP2 |
| 10000 | 900 | | |
| 100000000 | 50000 | | |
| 100000000 | 9000000 | | |

## Bài tập về nhà

Bài tập về nhà:

Tính bao nhiêu tấn gạo nếu thưởng cho bài toán bàn cờ. Biết 1 kg khoảng xấp xỉ 1 triệu hạt.

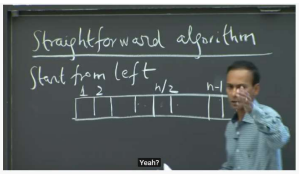


## Tài liệu đọc thêm

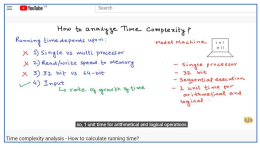
Papadimitriou, C. H. (2003). Computational complexity (pp. 260-265). John Wiley and Sons Ltd..

## Link các bài bài giảng về độ phức tạp của thuật toán

1. MIT:https://www.youtube.com/watch?v=HtSuA80QTyo&list=PLUl4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb




2. https://www.youtube.com/watch?v=8syQKTdgdzc




3. https://www.youtube.com/watch?v=9TlHvipP5yA