

Tên bài giảng

Phương pháp phân tích thiết kế chia để trị

Môn học: **Phân tích thiết kế thuật toán**

Chương: 3

Hệ: Đại học

Giảng viên: TS. Phạm Đình Phong

Email: phongpd@utc.edu.vn

Nội dung bài học

- 1. Phân tích thiết kế, đánh giá thuật toán chia để trị**
- 2. Một số thuật toán giảm chia để trị**
- 3. Một số thuật toán chia để trị**

Phương pháp chia để trị

- Áp dụng cho các bài toán có thể giải quyết bằng cách
 - Chia nhỏ ra thành các bài toán con
 - Giải quyết các bài toán con
 - Lời giải của các bài toán con được tổng hợp lại thành lời giải cho bài toán ban đầu

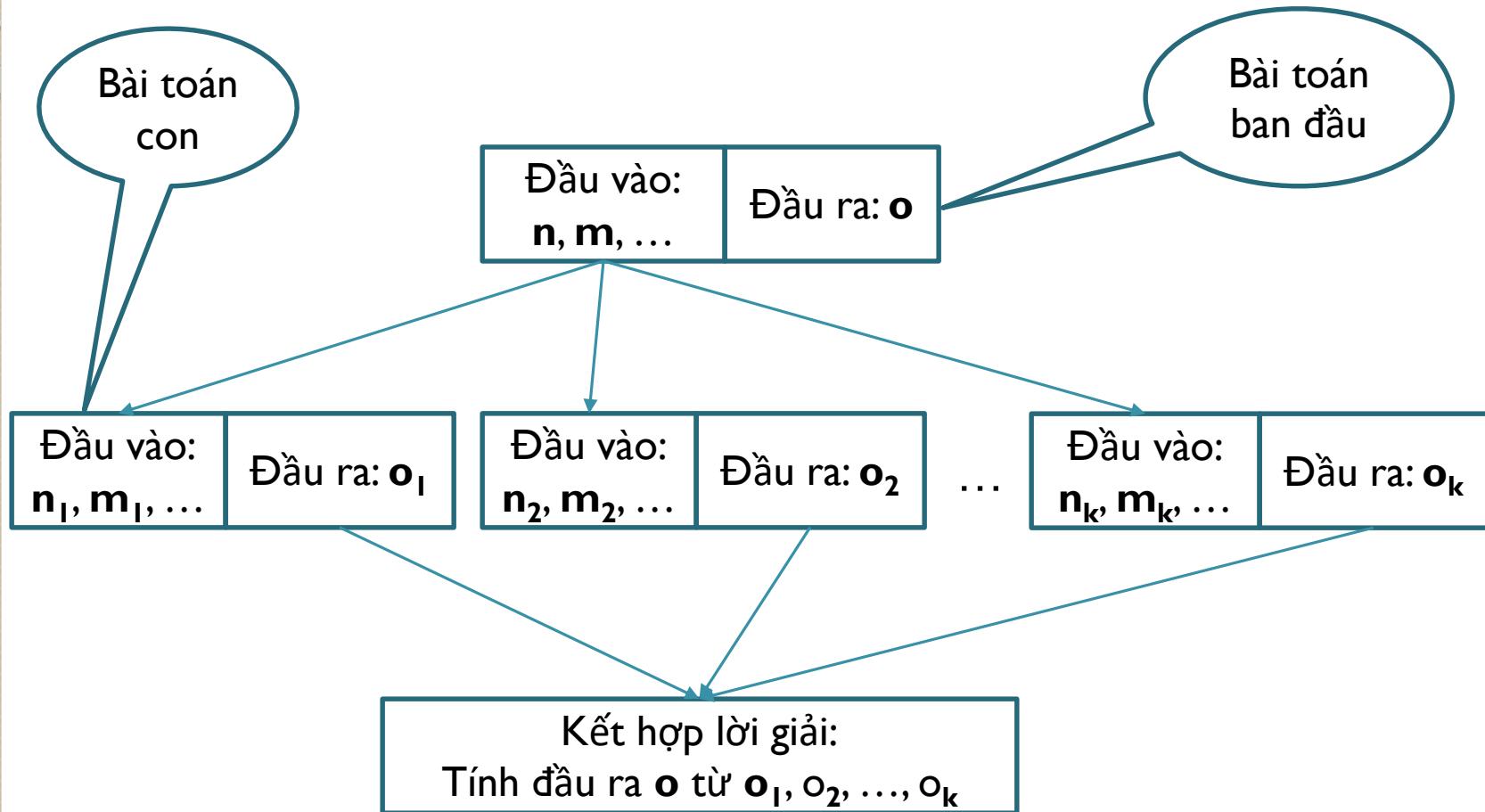
Phương pháp chia để trị

- Phương pháp
 - Bước 1: Chia/Tách nhỏ
 - Chia bài toán ban đầu thành các bài toán con đồng dạng với bài toán ban đầu
 - Bước 2: Trị/Giải quyết bài toán con
 - Giải các bài toán con → các lời giải con
 - Bước 3: Kết hợp lời giải
 - Tổng hợp các lời giải con → lời giải của bài toán ban đầu

Lưu ý:

- Các bài toán con lại được chia thành các bài toán nhỏ hơn nữa
- Dừng lại khi kích thước bài toán đủ nhỏ mà ta có thể giải dễ dàng → **bài toán cơ sở**

Phương pháp chia để trị



Phương pháp chia để trị

solve(*n*)

{

if (*n* đủ nhỏ để có thể giải được) {

 giải bài toán $\rightarrow KQ$

return *KQ*;

 } **else** {

 Chia bài toán thành các bài toán con kích thước *n₁*, *n₂*, ...

KQ₁ = **solve**(*n₁*); //giải bài toán con 1

KQ₂ = **solve**(*n₂*); //giải bài toán con 2

 ...

 Tổng hợp các kết quả *KQ₁*, *KQ₂*, ... $\rightarrow KQ$

return *KQ*;

}

Phương pháp chia để trị

- Tính lũy thừa
 - Cho số nguyên a và số nguyên $n \geq 1$. Tính a^n .
 - Phương pháp thông thường: $a = a \times a^{n-1}$
 $\rightarrow T(n) = O(n)$

Phương pháp chia để trị

- Tính lũy thừa (tiếp)
 - Phương pháp chia để trị:

Ta thấy: $a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}$

$a^{\lfloor n/2 \rfloor}$ và $a^{\lceil n/2 \rceil}$ đều là các bài toán con vì lũy thừa nhỏ hơn

$$a^n = \begin{cases} a^{n/2} \times a^{n/2} & \text{Nếu } n \text{ chẵn} \\ a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil} \times a & \text{Nếu } n \text{ lẻ} \end{cases}$$

Phương pháp chia để trị

- Tính lũy thừa (tiếp)

- Phương pháp chia để trị:

$$a^n = \begin{cases} a^{n/2} \times a^{n/2} & \text{Nếu } n \text{ chẵn} \\ a^{\lfloor n/2 \rfloor} \times a^{\lfloor n/2 \rfloor} \times a & \text{Nếu } n \text{ lẻ} \end{cases}$$

$$T(n) = T(\lfloor n/2 \rfloor) + O(1)$$

Giải phương trình đệ quy:

Ta có: $a = 1, b = 2$

Nghiệm thuận nhất: $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$

$a = d(n) = 1 \rightarrow$ nghiệm riêng là: $n^{\log_b a} \log_b n = 1 \log_2 n$

$\rightarrow T(n) = O(\log n)$

Phương pháp chia để trị

- Tính lũy thừa (tiếp)
 - Phương pháp chia để trị:

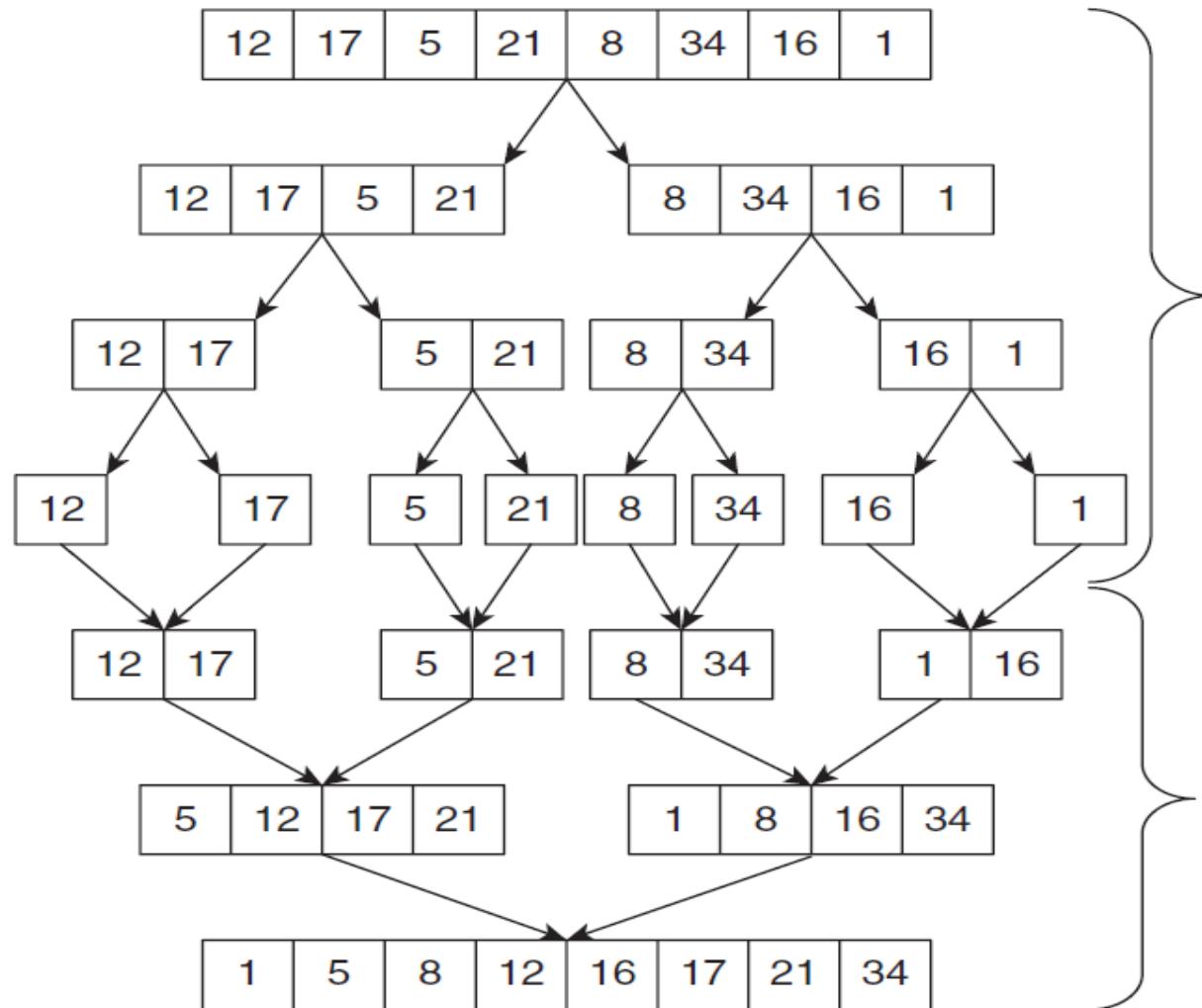
```
long power(int a, int n)
{
    long x;
    if (n == 1)
        return a;
    else {
        x = power(a, n/2); //chia nguyên
        if (n % 2 == 0)      //n chẵn
            return x * x;
        else                  //n lẻ
            return x * x * a;
    }
}
```

Phương pháp chia để trị

- Sắp xếp trộn (John von Neumann 1945)
 - Ý tưởng gồm:
 - **Chia:** Chia mảng $A[1, 2, \dots, n]$ thành hai phần có kích thước xấp xỉ nhau
 - **Trị:** Gọi đệ quy sắp xếp trên mỗi phần
 - **Trộn:** trộn hai mảng con đã được sắp xếp với nhau thành mảng được sắp xếp cho bước đệ quy hiện tại

Phương pháp chia để trị

- Sắp xếp trộn (tiếp)



Phương pháp chia để trị

- Sắp xếp trộn (tiếp)

- Độ phức tạp:

- $T(n) = 2T(n/2) + O(n)$

Số bài toán con

Kích thước bài toán con

Thời gian trộn

Giải phương trình đệ quy:

Ta có: $a = 2$, $d(b) = d(2) = 2$

Nghiệm thuần nhất: $n^{\log_b a} = n^{\log_2 2} = n$

Nghiệm riêng: $n^{\log_b a} \log_b n = n \log_2 n$

$$\rightarrow T(n) = O(n \log n)$$

Phương pháp chia để trị

- Sắp xếp trộn (tiếp)

```
void MergeSort(int A[], int left, int right) {  
    if (right > left)  
    {  
        int mid = (left + right) / 2; // Phần tử ở giữa (chia nguyên)  
        MergeSort(A, left, mid); // Gọi đệ quy mảng con bên trái  
        MergeSort(A, mid + 1, right); // Gọi đệ quy mảng con bên phải  
        Merge(A, left, mid, right); // Hàm trộn hai mảng con  
    }  
}
```

Phương pháp chia để trị

- Sắp xếp trộn (tiếp)

```
void Merge(int A[], int left, int mid, int right) {  
    int i = left, j = mid + 1, n = right - left + 1;  
    int B[n];  
    for (int k = 0; k < n; k++)  
        if (j > right) {  
            B[k] = A[i]; i++;  
        } else if (i > mid) {  
            B[k] = A[j]; j++;  
        } else if (A[i] < A[j]) {  
            B[k] = A[i]; i++;  
        } else {  
            B[k] = A[j]; j++;  
        }  
    for (int k = 0; k < n; k++) A[left + k] = B[k];  
}
```

Phương pháp chia để trị

- Tìm phần tử lớn nhất trong mảng A

- Phương pháp lặp:

```
int FindMax(int A[], int n)
{
    int max = A[0];
    for (int i = 1; i < n; i++)
        if (A[i] > max) max = A[i];
    return max;
}
```

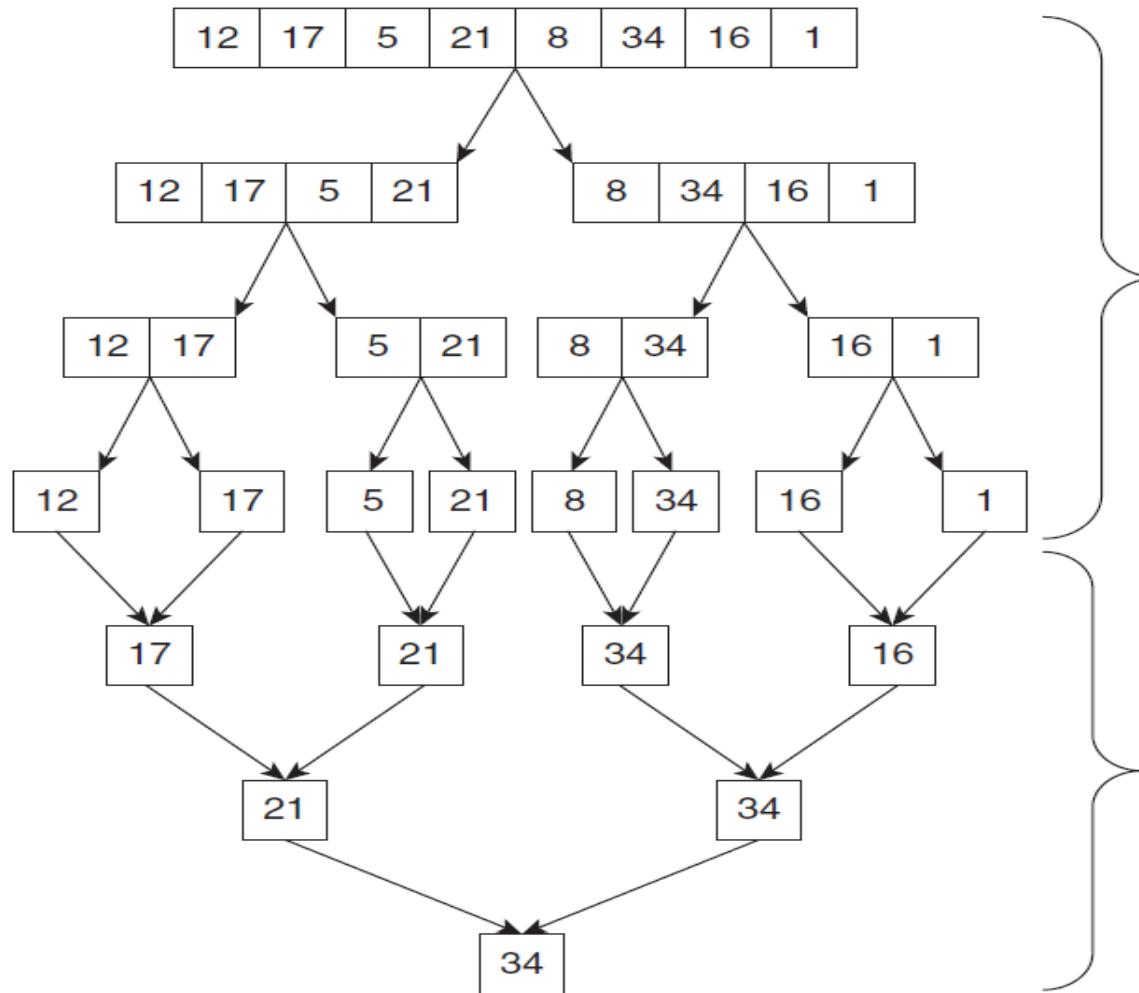
- Độ phức tạp: $T(n) = O(n)$

Phương pháp chia để trị

- Tìm phần tử lớn nhất trong mảng A
 - Chiến lược chia để trị:
 - **Chia:** Chia mảng A thành 2 phần và tiếp tục chia các mảng con cho tới khi chúng chỉ còn 1 phần tử
 - **Trị:** Tìm phần tử lớn nhất của các mảng con bằng phương pháp đệ quy
 - **Tổng hợp:** So sánh phần tử lớn nhất của các mảng con để tìm ra phần tử lớn nhất của mảng
 - Độ phức tạp: $T(n) = O(n)$

Phương pháp chia để trị

- Tìm phần tử lớn nhất trong mảng A



Phương pháp chia để trị

- Tìm phần tử lớn nhất trong mảng A

- Chiến lược chia để trị:

- Phương trình đệ quy:

$$T(n) = 2T(n/2) + O(1)$$

$$a = 2, d(b) = 2^0 = 1 \rightarrow a > d(b)$$

Nghiệm thuần nhất là $n^{\log_b a} = n^{\log_2 2} = n$

- Độ phức tạp: $T(n) = O(n)$

Phương pháp chia để trị

- Tìm phần tử lớn nhất trong mảng A

```
int FindMax(int A[], int low, int high) {  
    if (low == high) { //Nếu chỉ số đầu và cuối bằng nhau  
        int max = A[low];  
        return max; //mảng còn một phần tử  
    } else {  
        int mid = (low + high) / 2;  
        int x= FindMax(A, low, mid);  
        int y = FindMax(A, mid+1, high);  
        //Trả lại phần tử lớn nhất từ hai mảng con  
        if (x> y) return x;  
        else return y;  
    }  
}
```

Phương pháp chia để trị

- Dãy Fibonacci
 - Định nghĩa đệ quy

$$F_n = \begin{cases} 0 & \text{Nếu } n=0 \\ 1 & \text{Nếu } n=1 \\ F_{n-1} + F_{n-2} & \text{Nếu } n \geq 2 \end{cases}$$

0 1 1 2 3 5 8 13 21 34...

Phương pháp chia để trị

- Dãy Fibonacci
 - Định nghĩa đệ quy

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

Thuật toán ngây thơ: $\Omega(\Phi^n)$

(thời gian hàm mũ), trong đó $\Phi = (1 + \sqrt{5})/2$
là tỉ lệ vàng

Phương pháp chia để trị

- Dãy Fibonacci

- Tính toán từ dưới lên (bottom-up)
 - Tính $F_0, F_1, F_2, \dots, F_n$ theo thứ tự, tạo các số bằng cách tính tổng 2 số trước
 - Thời gian thực hiện: $O(n)$

```
long fibonacci(int n) {  
    if (n < 2) return n;  
    long n1 = 1, n2 = 0; // lưu cơ sở vào các biến  
    for (int i = 2; i < n; i++) {  
        long n0 = n1 + n2;  
        n2 = n1;  
        n1 = n0;  
    }  
    return n1 + n2;  
}
```

Giảm để trị

- Trường hợp đặc biệt của chia để trị
- Quy về đúng một bài toán nhỏ hơn
- Có ba chiến lược
 - Giảm bởi một hằng số
 - Giảm bởi một hệ số
 - Giảm kích thước của biến/Chia cắt
 - Tìm điểm chia cắt
 - Xử lý tùy theo điều kiện (vd: $>$, $<$, $=$)

Lưu ý:

- Dừng chia cắt khi không thể chia cắt được nữa
- Kiểm tra điều kiện trước khi chia cắt

Giảm để trị

- Tìm kiếm nhị phân
 - Tìm vị trí của số x trong mảng $A[0, 1, \dots, n-1]$ đã được sắp xếp tăng dần
 - Phần tử đầu tiên có vị trí 0. Trả về vị trí tìm thấy, nếu không tìm thấy trả về -1
 - Kỹ thuật giảm để trị (giảm kích thước biến)
 - So sánh x với phần tử ở giữa mảng $A[n/2]$
 - Nếu $x = A[n/2] \rightarrow$ trả về vị trí của $A[n/2]$
 - Nếu $x < A[n/2] \rightarrow$ tìm kiếm trên $A[0, 1, \dots, (n/2)-1]$ (trái)
 - Nếu $x > A[n/2] \rightarrow$ tìm kiếm trên $A[(n/2)+1, \dots, n-1]$ (phải)
 - Trả về -1

Giảm đế trị

- Tìm kiếm nhị phân (tiếp)

```
int binarySearch(int A[], int l, int r, int x) {  
    if (r >= l) {  
        int mid = (l + r) / 2; // Tính vị trí giữa mảng  
  
        if (A[mid] == x)      // Nếu A[mid] = x, trả về chỉ số và kết thúc.  
            return mid;  
  
        // Nếu A[mid] > x, thực hiện tìm kiếm nửa trái của mảng  
        if (A[mid] > x)  
            return binarySearch(A, l, mid - 1, x);  
  
        // Nếu A[mid] < x, thực hiện tìm kiếm nửa phải của mảng  
        return binarySearch(A, mid + 1, r, x);  
    }  
  
    return -1;  
}
```

Giảm để trị

- Tìm kiếm nhị phân (tiếp)
 - Mỗi bước tìm kiếm loại bỏ ít nhất một nửa số phần tử của mảng
 - Thời gian tìm kiếm

$$T(n) = 1T(n/2) + O(1)$$

Giải phương trình đệ quy:

Ta có: $a = 1, b = 2$

Nghiệm thuần nhất: $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$

$a = d(n) = d(2) = 1$

→ nghiệm riêng là: $n^{\log_b a} \log_b n = 1 \log_2 n$

→ $T(n) = O(\log n)$

Giảm để trị

- **Tìm kiếm nhị phân (tiếp)**



Giảm để trị

- Tìm ước số chung lớn nhất của hai số theo giải thuật Euclid
 - Kỹ thuật giảm để trị (giảm kích thước biến)

```
int gcd(int a, int b) {  
    int r;  
    while(b != 0) {  
        r = a % b; //Phép chia dư  
        a = b;  
        b = r;  
    }  
    return a;  
}
```

Ví dụ: a = 60, b = 24

1. a = 60 và b = 24
2. a = 24 và b = 12
3. a = 12 và b = 0

12 là ước số chung lớn nhất

Giảm để trị

- Sắp xếp chèn
 - Sắp xếp một mảng $a[1..n]$ tăng dần
 - Kỹ thuật giảm để trị (giảm kích thước biến)
 - Giả sử $a[1..n-1]$ đã được sắp
 - Cần chèn $a[n]$ vào mảng con $a[1..n-1]$
 - Có hai cách để thực hiện:
 - Duyệt $a[1..n-1]$ từ trái sang phải cho đến khi tìm được phần tử đầu tiên lớn hơn hoặc bằng $a[n]$ và chèn $a[n]$ vào bên trái phần tử này
 - Duyệt $a[1..n-1]$ từ phải sang trái cho đến khi tìm được phần tử đầu tiên nhỏ hơn hoặc bằng $a[n]$ và chèn $a[n]$ vào bên phải của phần tử này

Giảm để trị

- Sắp xếp chèn (tiếp)

$$a[1] \leq \dots \leq a[j] < a[j+1] \leq a[k-1] \mid a[k] \dots a[n]$$

$$a[j] \leq a[k] < a[j+1]$$

Dãy cần được sắp xếp: **120** 50 89 100 24
 50 **120** 89 100 24
 50 **89** **120** 100 24
 50 **89** **100** **120** 24

Dãy đã được sắp xếp: **24** **50** **89** **100** **120**

Giảm để trị

- Sắp xếp chèn (tiếp)

```
void insertion_sort(int a[], int n) {  
    int i, j, v;  
    for (i = 2; i <= n; i++) {  
        v = a[i];  
        j = i;  
        while (a[j - 1] > v && j>=0) {  
            a[j] = a[j - 1]; //Kéo xuống  
            j--;  
        }  
        a[j] = v;  
    }  
}
```

Giảm để trị

- Sắp xếp chèn (tiếp)
 - Đánh giá độ phức tạp
 - Vòng lặp ngoài (chỉ số i) của thuật toán được thực hiện $n - 1$ lần
 - Trong trường hợp xấu nhất (mảng có thứ tự đảo ngược), vòng lặp trong được thực thi với số lần:
$$(n-1) + (n-2) + \dots + 1 = n(n-1)/2$$

$$\Rightarrow T(n) = O(n^2)$$

Giảm để trị

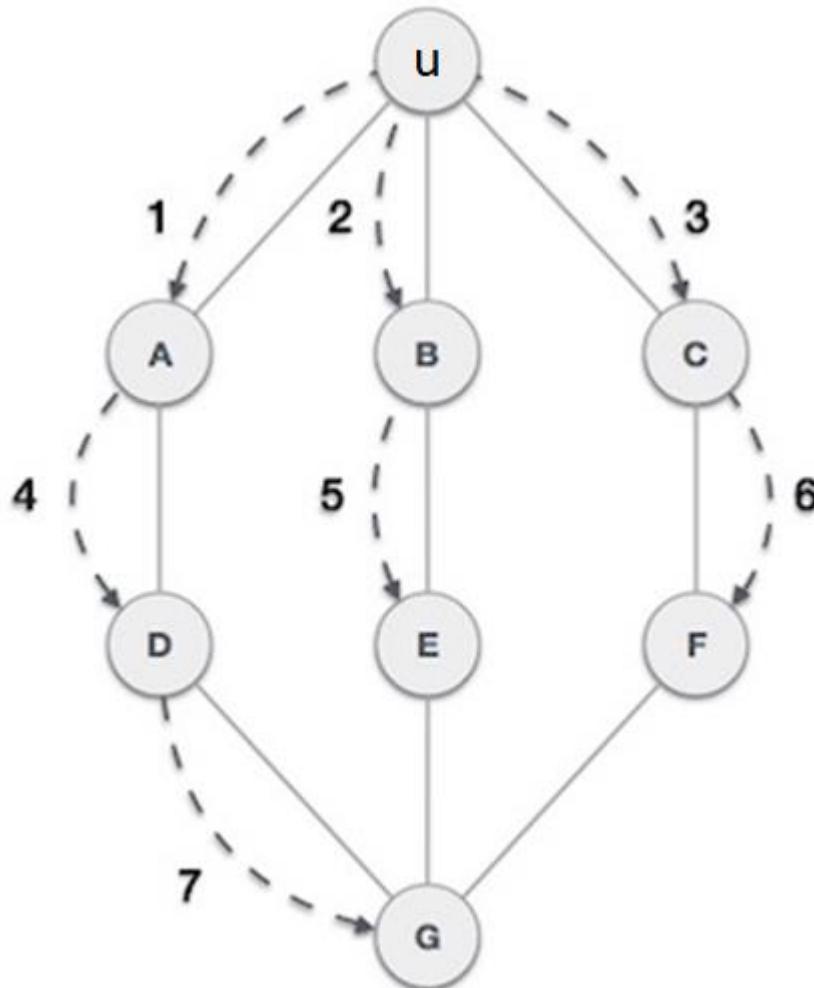
- Thuật toán tìm kiếm theo chiều rộng (BFS) và tìm kiếm theo chiều sâu (DFS)
 - Tại mỗi bước của thuật toán BFS hoặc DFS, thuật toán đánh dấu đỉnh đã được viếng và chuyển sang xét các đỉnh kế cận của đỉnh đó
 - Hai thuật toán này đã áp dụng kỹ thuật giảm bớt một, là một trong ba dạng chính của chiến lược Giảm để trị

Giảm đế trị

- Tìm kiếm theo chiều rộng
 - Thăm tất cả các đỉnh kề với đỉnh u trước khi tiếp tục với các đỉnh khác
 - sử dụng một cấu trúc dữ liệu hàng đợi (queue) để lưu trữ thông tin trung gian thu được trong quá trình tìm kiếm
 - 1. Thêm đỉnh gốc vào queue và đánh dấu đỉnh gốc.
 - 2. Nếu queue chưa rỗng, lấy ra đỉnh u đầu tiên khỏi queue. Xét các đỉnh v kề với đỉnh u
 - Nếu đỉnh v đã được đánh dấu thì bỏ qua.
 - Nếu v chưa được đánh dấu thì thêm đỉnh v vào queue và đánh dấu đỉnh v .
 - 3. Nếu queue rỗng, dừng quá trình tìm kiếm

Giảm đế trị

- Tìm kiếm theo chiều rộng
 - Ví dụ 1:



Giảm để trị

- Tìm kiếm theo chiều rộng
 - Mã giả của thuật toán BFS

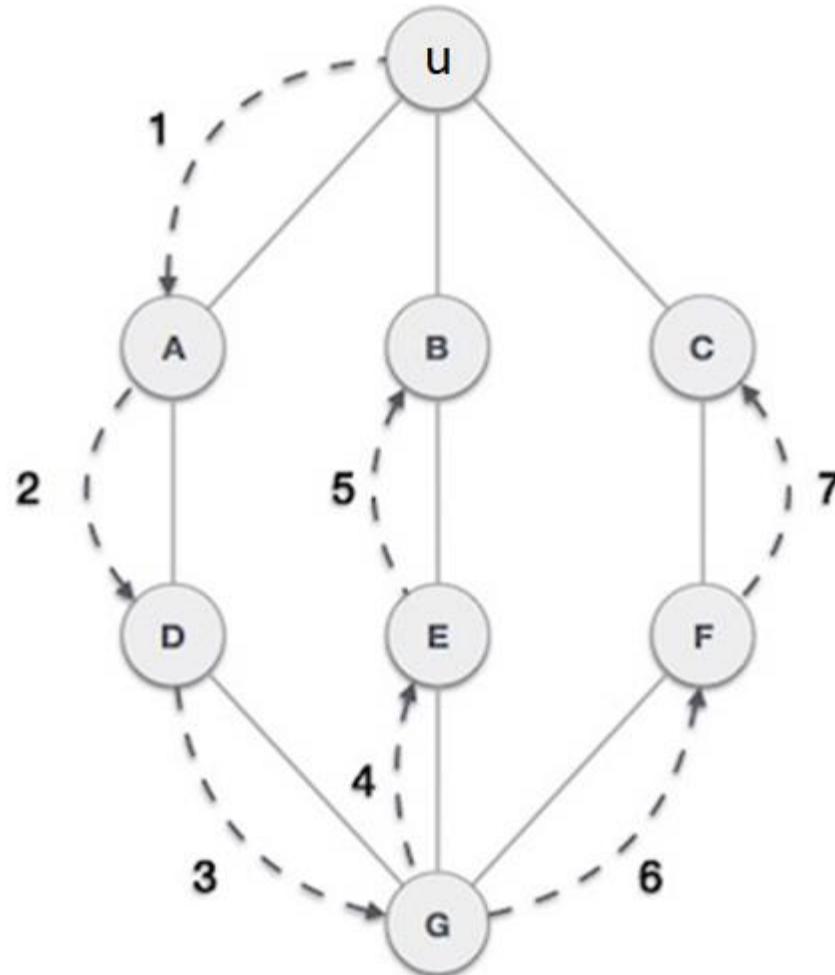
```
void BFS (int u) { //duyệt các đỉnh cùng thành phần liên thông với u
    queue = Ø;
    queue.enqueue(u); //nạp u vào hàng đợi
    chuaxet[u] = false; // đổi trạng thái của u
    while (queue ≠ Ø) { // duyệt tới khi nào hàng đợi rỗng
        p = queue.dequeue(); // lấy p ra từ khỏi hàng đợi
        Tham_dinh(p); // duyệt xong đỉnh p
        for (v ∈ ke(p)) { //duyệt mọi đỉnh v kề với p
            if (chuaxet[v] ) {
                queue.enqueue(v); // đưa v vào hàng đợi
                chuaxet[v] = false; // đổi trạng thái của v
            } //end if
        } //end for
    } //end while
}
```

Giảm để trị

- Tìm kiếm theo sâu
 - Thăm một đỉnh kề của u , sau đó thăm tiếp một đỉnh kề của đỉnh kề đó
→ Sử dụng ngăn xếp (stack) để ghi nhớ đỉnh liền kề để bắt đầu việc tìm kiếm
 - Thuật toán tiếp tục cho tới khi gặp được đỉnh cần tìm hoặc tới một nút không có con. Khi đó thuật toán quay lui về đỉnh vừa mới tìm kiếm ở bước trước

Giảm đế trị

- Tìm kiếm theo sâu
 - Ví dụ:



Giảm đế trị

- Tìm kiếm theo sâu
 - Thuật toán không đệ quy

```
void DFS( int u ) { //Duyệt đỉnh cùng thành phần liên thông với u
    stack = Ø;          //Khởi tạo stack rỗng
    stack.push(u);      //Đưa u vào stack
    while (stack ≠ Ø) { //Lặp cho tới khi stack rỗng
        v = stack.pop(); //Lấy v ra khỏi stack
        Tham_dinh(v);   //Duyệt xong đỉnh v
        if (chuaxet[v]) { //Nếu v chưa được xét
            chuaxet[v] = false; //Đánh dấu v là đã được xét
            for (w ∈ ke(v)) { //Duyệt các đỉnh kề với v
                if (chuaxet[w]) stack.push(w); //Nếu w chưa được xét
            }
        }
    }
}
```

Giảm đế trị

- Tìm kiếm theo sâu
 - Thuật toán đệ quy

```
void DFS(int u) {  
    Tham_dinh(u);  
    chuaxet[u] = false;  
    for (v ∈ ke(u)) {           //Xét các cạnh kề u  
        if (chuaxet[v]) DFS(v); //Gọi đệ quy để duyệt  
    }  
}
```

TỔNG KẾT

1. Tìm hiểu thuật toán chia để trị
2. Minh họa một số bài toán minh họa
3. Ưu điểm:
 - ❖ Giảm độ phức tạp về thời gian đối với nhiều bài toán
 - ❖ Chia để trị là một trong số các kĩ thuật mạnh mẽ để thiết kế thuật toán
4. Nhược điểm:
 - ❖ Sử dụng đệ quy nên tốn bộ nhớ và cần kỹ thuật xử lý
 - ❖ Không phải là giải pháp để giải quyết mọi bài toán

Bài tập

Bài 1: Tìm tổng lớn nhất của đoạn con liên tục

- Bài toán

- Input: a_1, a_2, \dots, a_n
- Output, $i, j \mid a_i + a_{i+1} + \dots + a_j$ là lớn nhất

Bài 2: Thiết kế thuật toán chia để trị tìm 1 dãy con liên tục của 1 dãy số có tổng nhỏ nhất

Bài 3: Phân tích thiết kế đánh giá độ phức tạp của thuật toán sắp xếp nhanh (Quick sort)