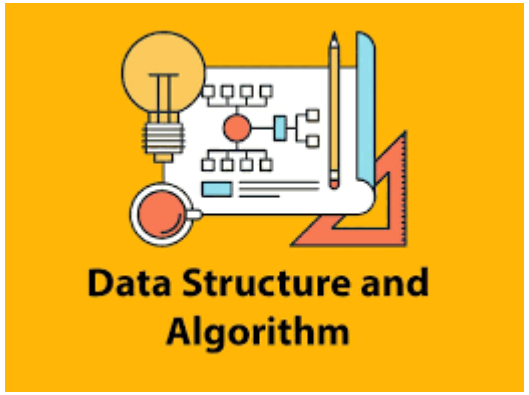
CẤU TRÚC DỮ LIỆU & GIẢI THUẬT

NGÀY 4: CHIA ĐỂ TRỊ



Giảng viên: Th.S Bùi Văn Kiên

NỘI DUNG

- Khái niệm – phân tích độ phức tạp
- Các bài toán ứng dụng:
  - Tìm kiếm nhị phân
  - Nhân số nguyên lớn
  - Tính lũy thừa
  - Merge-sort
  - Quick-sort
  - Dãy xâu Fibonacci
  - Tính số Fibonacci thứ N với N lớn

1. Khái niệm

Thuật toán chia để trị (Divide and Conquer)

- Mục tiêu: Giảm thời gian chạy
- Ý tưởng: Áp dụng đệ quy theo 3 bước
  - **Divide (Chia):** Chia bài toán lớn thành những bài toán con có cùng kiểu với bài toán lớn.
  - **Conquer (Trị):** Giải các bài toán con. Thông thường các bài toán con chỉ khác nhau về dữ liệu vào nên ta có thể thực hiện bằng một thủ tục đệ quy.
  - **Combine (Tổng hợp):** Tổng hợp lại kết quả của các bài toán con để nhận được kết quả của bài toán lớn.

1. Khái niệm

Áp dụng khi nào?

- Bài toán có thể được chia thành các bài toán con giống bài toán gốc nhưng với kích thước nhỏ hơn
- Ứng dụng trong xử lý song song ...

- Một số thuật ngữ liên quan
  - Chia và trị (Divide and Conquer)
  - Giảm và trị (Decrease and Conquer)
  - Chặt nhị phân
  - Chặt tam phân

1. Độ phức tạp thuật toán

- Gọi T(n) là độ phức tạp tính toán, n_0 là trường hợp dừng.
- Theo thuật toán tổng quát
  - T(n) = O(1) nếu $n = n_0$
  - T(n) = aT(n/b) + D(n) + C(n) nếu $n > n_0$
- Trong đó
  - a: Số bài toán con
  - n/b: Kích thước bài toán con
  - D(n): Độ phức tạp thời gian của phần Divide
  - C(n): Độ phức tạp thời gian của phần Combine

1. Độ phức tạp thuật toán

- Gọi T(n) là độ phức tạp tính toán, n_0 là trường hợp dừng.
- Theo thuật toán tổng quát
  - T(n) = O(1) nếu n = n_0
  - T(n) = aT(n/b) + D(n) + C(n) nếu n > n_0
- Định lý Master về độ phức tạp thuật toán cho bài toán chia để trị

Application to common algorithms [edit]

Algorithm	Recurrence relationship	Run time
Binary search	$T(n) = T\left(\frac{n}{2}\right) + O(1)$	$O(\log n)$
Binary tree traversal	$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$	$O(n)$
Optimal sorted matrix search	$T(n) = 2T\left(\frac{n}{2}\right) + O(\log n)$	$O(n)$
Merge sort	$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$	$O(n \log n)$

2. Các bài toán áp dụng

- Tìm kiếm nhị phân
- Nhân số nguyên lớn
- Tính lũy thừa
- Merge-sort
- Quick-sort
- Dãy xâu Fibonacci
- Tính số Fibonacci thứ N với N lớn

2-1. Tìm kiếm nhị phân

- Bài toán: Cho số x và mảng A[] các số nguyên được sắp xếp theo thứ tự không giảm. Tìm i sao cho A[i] = x.

2-1. Tìm kiếm nhị phân

- Chia danh sách thành hai phần. So sánh X với phần tử ở giữa.
- Có 3 trường hợp :
  - Trường hợp 1: nếu x bằng phần tử ở giữa A[mid], thì mid chính là vị trí của x trong danh sách A[].
  - Trường hợp 2: Nếu x lớn hơn phần tử ở giữa thì ta chỉ cần tìm các phần tử từ mid+1 đến vị trí thứ n.
  - Trường hợp 3: Nếu x nhỏ hơn A[mid] thì x chỉ có thể ở dãy con bên trái, tức là vị trí 1 tới mid-1.
- Lặp lại quá trình trên cho đến khi cận dưới vượt cận trên của dãy A[] mà vẫn chưa tìm thấy X → kết luận X không có mặt trong dãy A[].

2-1. Tìm kiếm nhị phân

- Độ phức tạp thuật toán:

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

- Kết luận: $T(n) = O(\log_2 n)$

2-1. Tìm kiếm nhị phân

Thuật toán thông thường

```
int Binary_Search(int A[], int n, int X) {
    int  low = 0;
    int  high = n-1;
    int  mid;
    while (low <= high) {
        mid = (low + high)/2;
        if (X > A[mid])
            low = mid + 1;
        else if (X < A[mid])
            high = mid -1;
        else
            return (mid);
    }
    return (-1);
}
```

Ví dụ 1:
A = [1 2 3 4 5 6]
X = 3
Return 2

Ví dụ 2:
A = [1 3 3 3 5 6]
X = 3
Return 2

2-1. Tìm kiếm nhị phân

Lower_bound và upper_bound

```
int LowerBound_Search(int A[], int n, int X) {
    int  low = 0;
    int  high = n-1;
    int  ans = -1, mid;
    while (low <= high) {
        mid = (low+high)/2;
        if (X > A[mid]) {
            low = mid + 1;
        }
        else if (X <= A[mid]) {
            high = mid -1;
            ans = mid;
        }
    }
    return ans;
/// Kiểm tra lại A[ans] == X hay A[ans] > X
}
```

Ví dụ 2:
A = [1 3 3 3 5 6]
X = 3
Lower_bound: phần tử đầu tiên >= X
Upper_bound: phần tử cuối cùng <=X

Lower_bound = 1
Upper_bound = 3

Định nghĩa Upper_bound trong STL library khác với định nghĩa phía trên. (Trong STL là phần tử đầu tiên > X)

2-2. Nhân số nguyên lớn

- Thuật toán cổ điển

Nhân hai số nguyên có n chữ số có độ phức tạp O(n^2).

- Cải tiến 1:
  - Rút gọn phép nhân hai số nguyên n chữ số còn bốn phép nhân hai số nguyên n/2

- Cải tiến 2 (thuật toán Karatsuba)
  - Giảm xuống còn 3 phép nhân

2-2. Nhân số nguyên lớn

- Input: $x = x_{n-1}x_{n-2}...x_1x_0$ và $y = y_{n-1}y_{n-2}...y_1y_0$
- Output: $z = x * y = z_{2n-1}z_{2n-2}...z_1z_0$
- Phân tích:

$$x = \sum_{i=0}^{n-1} x_i * 10^i = x_{n-1} * 10^{n-1} + x_{n-2} * 10^{n-2} + ... + x_1 * 10^1 + x_0 * 10^0$$

$$y = \sum_{i=0}^{n-1} y_i * 10^i = y_{n-1} * 10^{n-1} + y_{n-2} * 10^{n-2} + ... + y_1 * 10^1 + y_0 * 10^0$$

$$\Rightarrow z = x * y = \sum_{i=0}^{2n-1} z_i * 10^i = (\sum_{i=0}^{n-1} x_i * 10^i) * (\sum_{i=0}^{n-1} y_i * 10^i)$$

2-2. Nhân số nguyên lớn

- Cải tiến 1:

Đặt các biến phụ a, b, c, d:

$$a = x_{n-1}x_{n-2}\ldots x_{n/2} \qquad b = x_{(n/2)-1}x_{(n/2)-2}\ldots x_0$$

$$c = y_{n-1}y_{n-2}\ldots y_{n/2} \qquad d = y_{(n/2)-1}y_{(n/2)-2}\ldots y_0$$

Khi đó: $x = a*10^{n/2} + b$ $y = c*10^{n/2} + d$

$$\Rightarrow z = x*y = (a*10^{n/2}+b)(c*10^{n/2}+d) = (a*c)*10^{n} + (a*d+b*c)*10^{n/2} + (b*d)$$

2-2. Nhân số nguyên lớn

- Cải tiến 2: Thuật toán Karatsuba

Đặt $U = a \times c; V = b \times d; W = (a+b) \times (c+d)$

$\Rightarrow a \times d + b \times c = W - U - V$

$\Rightarrow Z = U \times 10^n + (W - U - V) \times 10^{n/2} + V$

2-2. Nhân số nguyên lớn

- Cải tiến 2: Thuật toán Karatsuba

```
Large_integer Karatsuba(x, y, n){
    If  n == 1  Return x*y
    Else  {
        a = x[n-1]. . . x[n/2]; b = x[n/2-1] . . . x[0];
        c = y[n-1]. . . y[n/2]; d = y[n/2-1] . . . y[0];
        U = Karatsuba(a, c, n/2);
        V = Karatsuba(b, d, n/2);
        W = Karatsuba(a+b, c+d, n/2);
        Return U*10n + (W-U-V)*10n/2 + V
    }
}
```

2-2. Nhân số nguyên lớn

- Cải tiến 2: Thuật toán Karatsuba
- Độ phức tạp thuật toán:

$$T(n) = \begin{cases} 1 & n = 1 \\ 3T(\frac{n}{2}) + cn & n > 1 \end{cases}$$

- $T(n) = O(n\log 3) \approx O(n^{1.58})$

2-3. Tính lũy thừa

- Bài toán: Tính a^n
- Điều kiện: a, n là các số nguyên và n không âm.

- Thuật toán lặp:

```
int result = 1;
for (int i = 1; i <= n; ++i)
        result *= a;
}
```

- Độ phức tạp O(n)

2-3. Tính lũy thừa

- Bài toán: Tính a^n
- Cải tiến bằng cách dùng chia để trị:
  - $a^n = 1$ nếu n = 0
  - $a^n = a^{n/2} \times a^{n/2}$ nếu n chẵn
  - $a^n = a \times a^{n/2} \times a^{n/2}$ nếu n lẻ

2-3. Tính lũy thừa

- Bài toán: Tính a^n
- Cải tiến bằng cách dùng chia để trị:

```
int power(int a, int n) {
    if(n == 0)
        return 1;
    int x = power(a,n/2);
    if(n %2 == 0) return x*x;
    return a*x*x;
}
```

- Độ phức tạp: O(log n)
- Lưu ý: chọn kiểu dữ liệu cho phù hợp để tránh bị tràn số, hoặc sử dụng them lấy phần dư theo modulo.

2-3. Tính lũy thừa

- Bài toán: Tính a^n % m
- **Tính chất đồng dư:**
- Với một giá trị m, các phép toán khi chia dư modulo cho m sẽ có tính chất sau:
  - Phép cộng: (A+B) mod m = A mod m + B mod m
  - Phép trừ: (A-B) mod m = (A mod m – B mod m + m) mod m
  - Phép nhân: (A*B) mod m = (A mod m * B mod m) mod m

- Vì sao thường chọn cơ số m = 10^9+7
  - Là một số nguyên tố
  - Tích hai số vẫn chưa vượt quá 2^63

2-4. Sắp xếp trộn (Merge-sort)

- Merge-Sort cũng được xây dựng theo mô hình chia để trị (Divide and Conquer).
- Thuật toán chia dãy cần sắp xếp thành hai nửa. Sau đó gọi đệ qui lại cho mỗi nửa và hợp nhất lại các đoạn đã được sắp xếp.
- Thuật toán được tiến hành theo 4 bước dưới đây:
  - Tìm điểm giữa của dãy và chi dãy thành hai nửa.
  - Thực hiện Merge-Sort cho nửa thứ nhất.
  - Thực hiện Merge-Sort cho nửa thứ hai.
  - Hợp nhất hai đoạn đã được sắp xếp (Merge).

2-4. Sắp xếp trộn (Merge-sort)

- Input: Dãy số Arr[], cận dưới L, cận trên R
- Output: Dãy số Arr[] được sắp theo thứ tự tăng dần.

- void Merge-Sort(Arr[], int L, int R){

```
        if (L<R) {
                M = (L + R) / 2;
                Merge-Sort(Arr, L, M);
                Merge-Sort(Arr, M+1, R);
                Combine(Arr, L, M, R);
        }
}
```

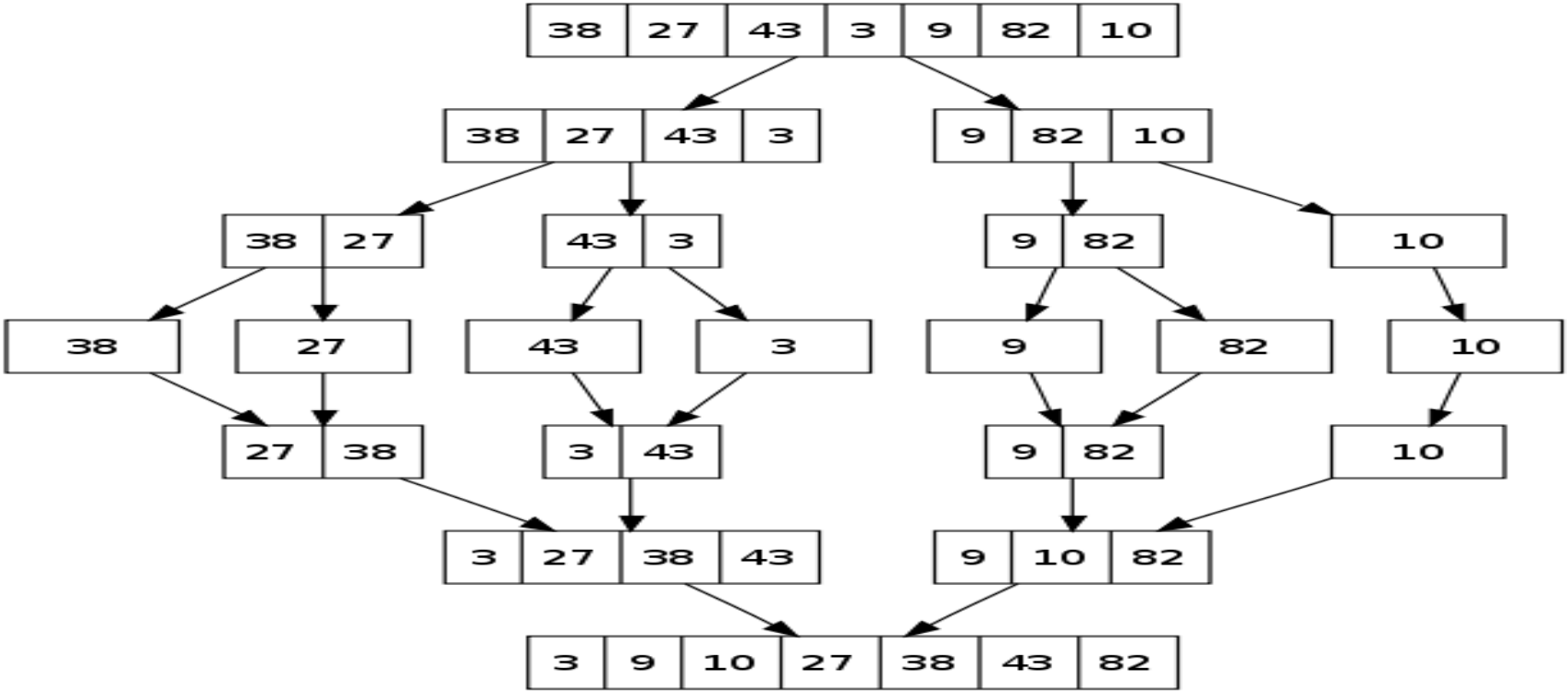
2-4. Sắp xếp trộn (Merge-sort)

- Bài toán hợp nhất 2 nửa dãy con Arr[L, M] và Arr[M+1, R] đã được sắp xếp thành dãy mới Arr[L, R].
- Ví dụ:
  - Arr[1, 4] = {3, 27, 38, 43}
  - Arr[5, 7] = {9, 10, 82}
  - ans[] = Arr[1, 7] = {3, 9, 10, 27, 38, 43, 82}
- Cách thực hiện:

Duy trì 2 biến i, j trên dãy con 1 và 2, so sánh Arr[i] <= Arr[j] hay không? Nếu có thì chọn Arr[i], ngược lại chọn Arr[j] để đưa vào dãy số kết quả ans[]

2-4. Sắp xếp trộn (Merge-sort)

- Mô tả quá trình gọi đệ quy:



2-4. Sắp xếp trộn (Merge-sort)

- Độ phức tạp thuật toán về mặt thời gian:

  Xấu nhất, trung bình, tốt nhất: O(n log n)

- Độ phức tạp thuật toán về mặt không gian: O(n)

2-5. Sắp xếp nhanh (Quick-sort)

- Sử dụng chia để trị (Devide and Conquer).
- Thuật toán được thực hiện xung quanh một phần tử gọi là chốt (pivot). Mỗi cách lựa chọn vị trí phần tử chốt trong dãy sẽ cho ta một phiên bản khác nhau của thuật toán.

- Các phiên bản của thuật toán Quick-Sort thông dụng là:
  - Chọn phần tử đầu tiên trong dãy làm chốt.
  - Chọn phần tử cuối cùng trong dãy làm chốt.
  - Chọn phần tử ở giữa dãy làm chốt.
  - Chọn phần tử ngẫu nhiên trong dãy làm chốt.

2-5. Sắp xếp nhanh (Quick-sort)

- Mấu chốt của thuật toán Quick-Sort là làm thế nào ta xây dựng được một thủ tục phân đoạn (Partition).

Thủ tục Partition có hai nhiệm vụ chính:

- Định vị chính xác vị trí của chốt trong dãy nếu được sắp xếp
- Chia dãy ban đầu thành hai dãy con:
  - Dãy con ở phía trước phần tử chốt gồm các phần tử nhỏ hơn hoặc bằng chốt.
  - Dãy ở phía sau chốt có giá trị lớn hơn chốt.

2-5. Sắp xếp nhanh (Quick-sort)

- Input : Dãy Arr[] gồm n phần tử, cận dưới L, cận trên R.
- Output: Dãy Arr[] được sắp xếp.

- **Formats:** Quick-Sort(Arr, L, R);

Actions:

```
        if (L<R) {
                pivot = Partition(Arr, L, R);
                Quick-Sort(Arr, L, pivot-1);
                Quick-Sort(Arr, pivot+1, R);
        }
End.
```

2-5. Sắp xếp nhanh (Quick-sort)

- Chọn phần tử cuối cùng A[R] làm chốt
- Vị trí chính xác của Arr[R] sau khi Arr[L→R] được sắp xếp.
- **Formats**: Partition(Arr, L, R);

Actions:

```
x = Arr[R];   i = (L - 1);
for (j = L; j <= R-1; j++) {
        if (Arr[j] <= x){
                i++;
                swap(Arr[i], Arr[j]);
        }
}
swap(Arr[i + 1], Arr[R]);
return (i + 1);
```

End.

2-5. Sắp xếp nhanh (Quick-sort)

- Độ phức tạp thuật toán:
- Xấu nhất: $O(n^2)$
- Trung bình: O(n log n)
- Tốt nhất: O(n log n)

2-5. Sắp xếp nhanh (Quick-sort)

- Ví dụ A[10] = {10, 27, 15, 29, 21, 11, 14, 18, 12, 17}

L = 0, R = 9, A[R] = 17	{10 15 11 14 12}, **{17},** {29 18 21 27}
→ Pivot = 5	
L = 0, R = 4, A[R] = 12	{10 11}, **{12},** {14, 15}
→ Pivot = 2	
L = 0, R = 1, A[R] = 11	10, **11** → No swap
L = 3, R = 4, A[R] = 15	14, **15** → No swap
L = 6, R = 9, A[R] = 27	{18, 21}, **{27}**, {29}
→ Pivot = 8	
……	……

Dãy số sau khi được sắp xếp
10, 11, 12, 14, 15, 17, 18, 21, 27, 29

2-6. Xâu Fibonacci

- Dãy Fibonacci
  - F[1] = 1
  - F[2] = 1
  - F[n] = F[n-1] + F[n-2]
- Các phần tử đầu tiên: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
- **Bài toán Fibonacci Word**

Cho trước hai xâu g1 và g2. Quy tắc tạo xâu tiếp theo như sau (với dấu + là nối xâu)

- g(1) = A
- g(2) = B
- g(n) = g(n-2) + g(n-1)

https://code.ptit.edu.vn/student/question/DSA04005

2-6. Xâu Fibonacci

- Nhận xét: Độ dài của xâu thứ i chính là F[i]
- **Bài toán:** Cho số tự nhiên N không quá 92 và vị trí K <= 10^{18}. Xác định ký tự thứ K trong xâu G(n).

Các xâu đầu tiên trong dãy G

A
B
AB
BAB
ABBAB
BABABBAB
ABBABBABABBAB
BABABBABABBABBABABBAB

N = 6, K = 4
→ Kí tự 'A'

N = 8, K = 19
→ Kí tự 'B'

https://code.ptit.edu.vn/student/question/DSA04005

QUESTIONS & ANSWERS