



CHƯƠNG 4. DEVIDE AND CONQUER ALGORITHM

NỘI DUNG:

- 4.1. Giới thiệu giải thuật
- 4.2. Thuật toán cộng
- 4.3. Thuật toán nhân
- 4.4. Giải thuật Binary-Search
- 4.5. Giải thuật Quick Sort
- 4.6. Giải thuật Merge Sort
- 4.7. CASE STUDY

4.1. Giới thiệu thuật toán

Thuật toán chia để trị (Devide and Conquer): dùng để giải lớp các bài toán có thể thực hiện được ba bước:

1. Devide (Chia). Chia bài toán lớn thành những bài toán con có cùng kiểu với bài toán lớn.
2. 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 qui.
3. Combine (Tổng hợp). Tổng hợp lại kết quả của các bài toán con để nhận được kết quả của bài toán lớn.

Một số thuật toán chi để trị điển hình:

- Thuật toán tìm kiếm nhị phân (Binary-Search).
- Thuật toán Quick-Sort.
- Thuật toán Merge-Sort.
- Thuật toán Strassen nhân hai ma trận.
- Thuật toán Kuley-Tukey nhân tính toán nhanh dịch chuyển Fourier.
- Thuật toán Karatsuba tính toán phép nhân...

4.2. Thuật toán nhân nhanh Karatsuba

Bài toán : Giả sử ta có hai số nguyên $a = (a_{n-1}a_{n-2}..a_1a_0)_2$, $b = (b_{n-1}b_{n-2}..b_1b_0)_2$. Khi đó phép nhân hai số nguyên thực hiện theo cách thông thường ta cần thực hiện n^2 lần tổng của tất cả các tích riêng theo thuật toán sau:

Thuật toán Multiple(a, b: positive integer):

Input :

- $a = (a_{n-1}a_{n-2}..a_1a_0)_2$: số a được biểu diễn bằng n bit ở hệ cơ số 2.
- $b = (b_{n-1}b_{n-2}..b_1b_0)_2$: số b được biểu diễn bằng n bit ở hệ cơ số 2.

Output:

- $c = (c_{2n-1}c_{2n-2}..c_1c_0)_2$: số c là tích hai số a và b.

Formats: $c = \text{Multiple}(a, b);$

Actions:

Bước 1. Tính tổng các tích riêng.

```
for ( j = 0; j < n; j++ ) {  
    if (bj == 1) cj = a<<j;  
    else cj = 0;  
}
```

Bước 2. Tính tổng các tích riêng.

```
p = 0;  
for ( j = 0; j < n; j++ ) p = p + cj;
```

Bước 3. Trả lại kết quả.

```
Return p;
```

EndActions.

Thuật toán nhân nhanh Karatsuba cho phép ta nhân nhanh hai số ở hệ cơ số bất kỳ với độ phức tạp là $O(n^{1.59})$. Thuật toán được thực hiện theo giải thuật chia để trị như sau:

Với mọi số tự nhiên x, y bất kỳ gồm n chữ số ở hệ cơ số B đều tồn tại số tự nhiên $m \leq n$ sao cho:

$$x = x_1 B^m + x_0 \quad y = y_1 B^m + y_0$$

Với x_0, y_0 nhỏ hơn B^m . Khi đó:

$$xy = (x_1 B^m + x_0)(y_1 B^m + y_0)$$

$$xy = z_2 B^{2m} + z_1 B^m + z_0$$

Trong đó:

$$z_2 = x_1 y_1$$

$$z_1 = x_1 y_0 + x_0 y_1$$

$$z_0 = x_0 y_0$$

Như vậy ta chỉ cần thực hiện 4 phép nhân và một số phép cộng và một số phép trừ khi phân tích các toán hạng.

Ví dụ. Ta cần nhân hai số $x = 12345$, $y = 6789$ ở hệ cơ số $B = 10$. Chọn $m=3$, khi đó:

$$12345 = 12 \cdot 1000 + 345$$

$$6789 = 6 \cdot 1000 + 789$$

Từ đó ta tính được:

$$z_2 = 12 \times 6 = 72$$

$$z_0 = 345 \times 789 = 272205$$

$$\begin{aligned} z_1 &= (12 + 345) \times (6 + 789) - z_2 - z_0 = 357 \times 795 - 72 - 272205 \\ &= 283815 - 72 - 272205 = 11538. \end{aligned}$$

$$\text{Kết quả} = 72 \cdot 1000^2 + 11538 \cdot 1000 + 272205 = 83810205.$$

Chú ý. Thời gian thuật toán sẽ thực hiện nhanh hơn khi ta lấy cơ số B cao (ví dụ $B = 1000$).

4.3. Tìm tổng dãy con liên tục lớn nhất

Bài toán: Cho dãy số nguyên bao gồm cả số âm lẫn số dương. Nhiệm vụ của ta là tìm dãy con liên tục có tổng lớn nhất.

Ví dụ. Với dãy số A = {-2, -5, 6, -2, -3, 1, 5, -6} thì tổng lớn nhất của dãy con liên tục ta nhận được là 7.

Thuật toán 1. Max-SubArray(Arr[], n): //Độ phức tạp O(n²).

Bước 1 (Khởi tạo):

Max = Arr[0]; // Chọn Max là phần tử đầu tiên.

Bước 2 (lặp):

```
for (i=1; i<n; i++) { //Duyệt từ vị trí 1 đến n-1
    S = 0; //Gọi S là tổng liên tục của / số
    for ( j =0; j<=n; j++) { //tính tổng của Arr[0],...,Arr[i].
        S = S + Arr[j];
        if (Max < S ) // Nếu Max nhỏ hơn
            Max = S;
    }
}
```

Bước 3 (Trả lại kết quả):

Return(Max).

Thuật toán 2. Devide-Conquer (Arr[], n): //Độ phức tạp O(nlog(n)).

```
int maxCrossingSum(int arr[], int l, int m, int h) {
    int sum = 0, left_sum = INT_MIN, right_sum = INT_MIN;
    for (int i = m; i >= l; i--) { //Tìm tổng dãy con từ l đến m
        sum = sum + arr[i];
        if (sum > left_sum) left_sum = sum;
    }
    sum = 0;
    for (int i = m+1; i <= h; i++) { // //Tìm tổng dãy con từ m+1 đến h
        sum = sum + arr[i];
        if (sum > right_sum) right_sum = sum;
    }
    return left_sum + right_sum; //Trả lại kết quả
}

int maxSubArraySum(int arr[], int l, int h) {
    if (l == h) return arr[l];
    int m = (l + h)/2; // Tìm điểm ở giữa
    return max(maxSubArraySum(arr, l, m),
               maxSubArraySum(arr, m+1, h),
               maxCrossingSum(arr, l, m, h));
}
```

Thuật toán 3. Hãy cho biết kết quả thực hiện chương trình //Độ phức tạp O(n).

```
#include<stdio.h>
int maxSubArraySum(int a[], int n){
    int max_so_far = 0, max_ending_here = 0;
    for(int i = 0; i < n; i++) {
        max_ending_here = max_ending_here + a[i];
        if(max_ending_here < 0)
            max_ending_here = 0;
        if(max_so_far < max_ending_here)
            max_so_far = max_ending_here;
    }
    return max_so_far;
}
int main(){
    int a[] = {-2, -3, 4, -1, -2, 1, 5, -3};
    int n = sizeof(a)/sizeof(a[0]);
    int max_sum = maxSubArraySum(a, n);
    printf("Tong day con lien tuc lon nhat: %d\n", max_sum);
    getchar();return 0;
}
```

2.5.4. Thuật toán tìm kiếm nhị phân.

Bài toán. Cho một dãy $A = (a_1, a_2, \dots, a_n)$ đã được sắp xếp theo thứ tự tăng dần. Nhiệm vụ của ta là tìm vị trí của x xuất hiện trong $A[]$.

Thuật toán Binary-Search(int Arr[], int l, int r, int x):

Input :

- $\text{Arr}[]$ mảng số đã được sắp xếp.
- Giá trị l : vị trí bên trái nhất bắt đầu tìm kiếm.
- Giá trị r : vị trí bên phải nhất bắt đầu tìm kiếm.
- Giá trị x : số cần tìm trong mảng $\text{Arr}[]$

Ouput:

- Return(mid) là vị trí của x trong khoảng l, r .
- Return(-1) nếu x không có mặt trong khoảng l, r .

Formats:

Binary-Search(Arr, l, r, x).

Độ phức tạp thuật toán: $O(\log(n))$: $n = l-r$.

4.4. Thuật toán tìm kiếm nhị phân.

```
Int Binary-Search( int Arr[], int l, int r, int x ) {  
    if (r >= l) {  
        int mid = l + (r - l)/2; //Bước chia: Chia bài toán làm hai phần  
        if (arr[mid] == x) //Trị trường hợp thứ nhất  
            return mid;  
        if (arr[mid] > x) //Trị trường hợp thứ hai  
            return binarySearch(arr, l, mid-1, x);  
        return binarySearch(arr, mid+1, r, x); //Trị trường hợp còn lại  
    }  
    return -1; //Kết luận tìm không thấy  
}
```

Ta có thể khử đệ quy theo thủ tục dưới đây:

```
int binarySearch(int arr[], int l, int r, int x) {  
    while (l <= r) {  
        int m = l + (r-l)/2; //Bước chia  
        if (arr[m] == x) return m; // Trị trường hợp 1.  
        if (arr[m] < x) l = m + 1; // Trị trường hợp 2.  
        else r = m - 1; // Trị trường hợp 3.  
    }  
    return -1; // Kết luận không tìm thấy.  
}
```

4.5. Thuật toán sắp xếp quick sort.

Thuật toán Quick Sort được thực hiện bằng cách chọn một phần tử trung tâm hay khóa (Pivot) để thực hiện bước chia. Bước chia: chia mảng thành hai đoạn. Đoạn bên trái bao gồm các phần tử nhỏ hơn khóa. Đoạn bên phải có các phần tử lớn hơn khóa. Bước trị được thực hiện bằng cách giống như bước chia nhưng cho các khóa thuộc từng đoạn bên trái và đoạn bên phải. Có nhiều Version khác nhau của thuật toán Quick Sort phụ thuộc vào phương pháp chọn khóa:

- Luôn chọn phần tử đầu tiên làm phần tử trung tâm (Private).
- Luôn chọn phần tử đầu tiên làm phần tử trung tâm (Private).
- Luôn chọn phần tử ở giữa làm khóa.
- Luôn chọn phần tử ngẫu nhiên làm khóa.

Độ phức tạp thuật toán là $O(n^2)$.

Thuật toán được mô tả sau đây được thực hiện bằng cách chọn khóa luôn là phần tử cuối cùng trong dãy. Tìm vị trí của khóa sao cho các phần tử bên trái đều nhỏ hơn khóa và các phần tử bên phải đều lớn hơn khóa.

4.5. Thuật toán sắp xếp quick sort.

```
int partition (int arr[], int l, int h) { //l là điểm đầu; h là điểm cuối
    int x = arr[h]; // Đây là phần tử khóa cần tìm vị trí
    int i = (l - 1); // chỉ số của các phần tử nhỏ hơn
    for (int j = l; j <= h- 1; j++) {
        // If current element is smaller than or equal to pivot
        if (arr[j] <= x) { // Nếu phần tử hiện tại nhỏ hơn khóa
            i++; // tăng chỉ số phần tử nhỏ hơn
            swap(&arr[i], &arr[j]); // đổi chỗ hai phần tử
        }
    }
    swap(&arr[i + 1], &arr[h]); //đổi chỗ cho khóa.
    return (i + 1);
}
```

```
Thuật toán quickSort(int arr[], int l, int h) {
    if (l < h) { //nếu l vẫn nhỏ hơn h
        int p = partition(arr, l, h); // Bước chia
        quickSort(arr, l, p - 1); // Trị đoạn bên trái.
        quickSort(arr, p + 1, h); // Trị đoạn bên phải
    }
}
```

4.6. Thuật toán sắp xếp Merge Sort.

Thuật toán Merge-Sort là thuật toán chia để trị được thực hiện theo bốn bước dưới đây.

- **Bước 1.** Chia dãy thành hai dãy con lấy điểm giữa làm trung tâm.
- **Bước 2.** Thực hiện Merge-Sort cho nửa thứ nhất.
- **Bước 3.** Thực hiện Merge-Sort cho nửa thứ hai.
- **Bước 4.** Hợp nhất hai nửa đã được sắp xếp.

Độ phức tạp thuật toán là $O(n \log n)$.

```
Thuật toán Merge- Sort(int arr[], int l, int r) {  
    if (l < r) {  
        int m = l+(r-l)/2; // Chia dãy thành hai đoạn  
        Merge-Sort(arr, l, m); // Trị đoạn thứ nhất  
        Merge-Sort(arr, m+1, r); //Trị đoạn thứ hai  
        Merge(arr, l, m, r); //Hợp nhất hai đoạn đã được sắp xếp  
    }  
}
```

4.6. Thuật toán sắp xếp Merge Sort.

Thuật toán Merge(int arr[], int l, int m, int r) {

 int i, j, k, n1 = m - l + 1, n2 = r - m;

 int L[n1], R[n2]; // Tạo hai đoạn trung gian

 // Copy vào mảng trung gian L, R

 for(i = 0; i < n1; i++) L[i] = arr[l + i];

 for(j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

 /* Hợp nhất hai mảng trung gian vào arr[l..r]*/

 i = 0; j = 0; k = l;

 while (i < n1 && j < n2) {

 if (L[i] <= R[j]) { arr[k] = L[i]; i++; }

 else { arr[k] = R[j]; j++; }

 k++;

}

 /* Copy phần còn lại của mỗi mảng nếu còn phần tử*/

 while (i < n1) {

 arr[k] = L[i]; i++; k++;

}

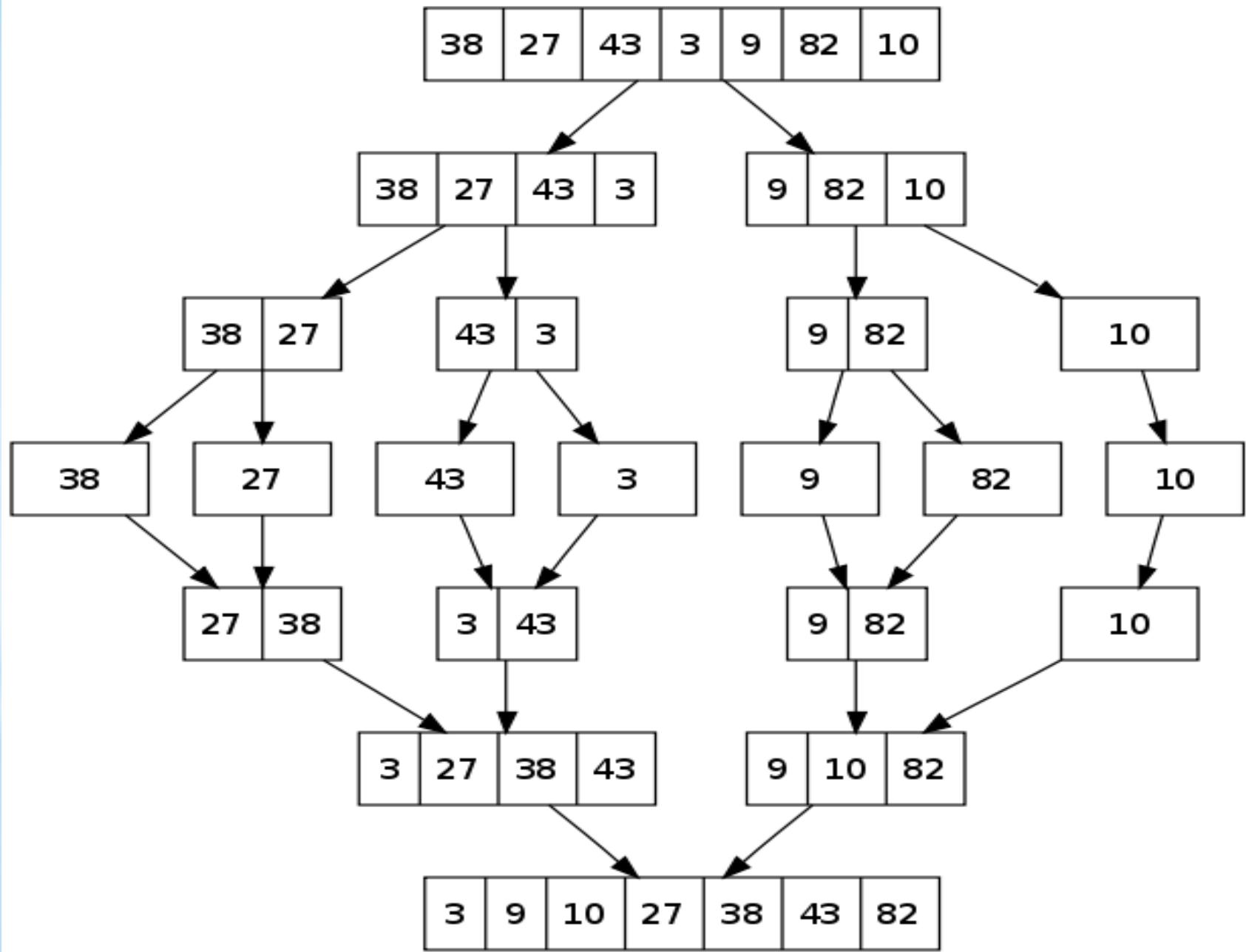
 while (j < n2) {

 arr[k] = R[j]; j++; k++;

}

}

2.5.6. Thuật toán sắp xếp Merge Sort.



4.7. CASE STUDY:

- a) Lập trình và kiểm tra lại cho các bài toán kể trên.**
- b) Tìm hiểu 10 bài toán khác nhau bằng phương pháp chia để trị..**