Chia để trị

Giới thiệu

Bài toán mở đầu

Bài toán: Cho một mảng A gồm n phần tử khác nhau, hãy lấy ra phần tử nhỏ thứ k trong mảng đó. Một biến thể hay gặp của bài toán này chính là tìm phần tử lớn nhất, nhỏ nhất hoặc trung vị.

```
Input: A = [7, 10, 4, 3, 20, 15], k = 3
Output: 7
Input: A = [7, 10, 4, 3, 20, 15], k = 4
Output: 10
```

Bài toán mở đầu (tiếp)

Chúng ta có thể nghĩ ngay đến hai hướng giải quyết:

- Dùng định nghĩa.
- Sắp xếp lại mảng.

Dùng định nghĩa

Định nghĩa: Phần tử *nhỏ thứ k* trong một mảng gồm các *phần tử khác nhau* là phần tử lớn hơn *đúng* k-1 phần tử khác.

Với định nghĩa này, chúng ta xây dựng được mã giả như sau

Dùng định nghĩa (tiếp)

```
def tìm_phần_tử_nhỏ_thứ_k(A: array, k):
    for x in A:
        # đếm số phần tử nhỏ hơn x trong A
        c = 0
        for y in A:
            if y < x:
                c = c + 1
        # kiểm tra có phải phần tử nhỏ thứ k
        if c == k - 1:
            return x
```

Dùng định nghĩa (tiếp)

Với mã giả như trên, ta có thể tính được độ phức tạp của thuật toán là $O(n^2)$.

Sắp xếp mảng

Một hướng tiếp cận khác đó là sắp xếp mảng tăng dần. Khi đó, phần tử nhỏ thứ k sẽ nằm ở vị trí k-1 (đánh số từ 0).

Ta có mã giả

Sắp xếp mảng (tiếp)

```
def tìm_phần_tử_nhỏ_thứ_k(A: array, k):
    sắp_xếp_tăng_dần(A)
    return A[k-1]
```

Sắp xếp mảng (tiếp)

Có thể thấy độ phức tạp của hướng này chính là độ phức tạp của việc sắp xếp mảng.

Hiện nay, sắp xếp mảng có thể thực hiện trong thời gian O(n*logn).

Hướng tiếp cận khác

Tuy nhiên, có một hướng tiếp cận cho phép giải quyết bài toán trên với độ phức tạp O(n).

Hướng tiếp cận khác (tiếp)

Ý tưởng cơ bản của hướng này như sau:

- Chọn một phần tử x bất kỳ trong mảng A.
- Chia mảng A thành 2 phần:
  - Một phần gồm những phần tử nhỏ hơn x, gọi là A_x.
  - Một phần gồm những phần tử lớn hơn x, gọi là A'_x.
- Tùy thuộc vào kết quả của việc phân chia, chúng ta sẽ tiếp tục tìm phần tử nhỏ thứ k trong A_x hoặc A'_x.

Hướng tiếp cận khác (tiếp)

Quan sát A_x, chúng ta có thể thấy có 3 kết quả:

1. Số lượng phần tử trong A_x bằng k - 1.
2. Số lượng phần tử trong A_x lớn hơn k - 1.
3. Số lượng phần tử trong A_x nhỏ hơn k - 1.

Trường hợp 1 (len(A_x) = k -1)

Nếu trường hợp 1 xảy ra, thì theo định nghĩa ta có phần tử x đã chọn chính là phần tử cần tìm.

A = [7, 10, 4, 3, 20, 15], k = 3.

Chọn được x = 7, khi đó:

A_x = [4, 3]

A_x' = [10, 20, 15]

Ta có, len(A_x) = 2, vậy, x = 7 chính là phần tử nhỏ thứ 3.

Trường hợp 2 (len(A_x) > k -1)

Nếu trường hợp 2 xảy ra, thì ta có phần tử cần tìm nằm trong A_x và nó cũng là phần tử nhỏ thứ k trong A_x.

A = [7, 10, 4, 3, 20, 15], k = 3.

Chọn được x = 20, khi đó:

A_x = [7, 10, 4, 3, 15]

A_x' = []

Ta có, len(A_x) = 5 > 2, vậy, ta tiếp tục tìm phần tử nhỏ thứ 3 trong A_x.

Trường hợp 3 (len(A_x) < k -1)

Nếu trường hợp 3 xảy ra, thì ta có phần tử cần tìm nằm trong A'_x và nó là phần tử nhỏ thứ k - len(A_x) - 1 (tại sao?) trong A'_x.

A = [7, 10, 4, 3, 20, 15], k = 3.

Chọn được x = 4, khi đó:

A_x = [3]

A_x' = [7, 10, 20, 15]

Ta có, len(A_x) = 1 < 2, vậy, ta tiếp tục tìm phần tử nhỏ thứ 3 - 1 - 1 = 1 trong A'_x.

Hướng tiếp cận khác (tiếp)

Vậy, ngoại trừ trường hợp 1, chúng ta có thể gọi đệ quy thuật toán tìm kiếm (với những tham số phù hợp) để giải quyết bài toán.

Chúng ta có mã giả tương ứng.

Hướng tiếp cận khác (tiếp)

```
def tìm_phần_tử_nhỏ_thứ_k(A: array, k):
    x = phần tử ngẫu nhiên trong A
    Ax = những phần tử nhỏ hơn x
    Ap = những phần tử lớn hơn x
    if len(Ax) == k - 1:                #trường hợp 1
        return x
    elif len(Ax) > k - 1:               #trường hợp 2
        return tìm_phần_tử_nhỏ_thứ_k(Ax, k)
    else:                               #trường hợp 3
        return tìm_phần_tử_nhỏ_thứ_k(Ap, k - len(Ax) - 1)
```

Hướng tiếp cận khác (tiếp)

Hướng tiếp cận này phụ thuộc vào việc chọn phần tử x đầu tiên, tối ưu nhất là chọn được trung vị (tức là chia được thành 2 phần bằng nhau).

Tuy nhiên, việc chọn được trung vị là không phải lúc nào cũng thực hiện được.

Hướng tiếp cận khác (tiếp)

Có nhiều cách để chọn được phần tử x này, chẳng hạn:

- Chọn phần tử đứng đầu hoặc đứng cuối hoặc đứng ở giữa.
- Chọn thông qua một thuật toán nào đó.
- ...

Tuy nhiên, với cách chọn x phù hợp, ta có thể giải bài toán ban đầu với độ phức tạp O(n).

Chia để trị

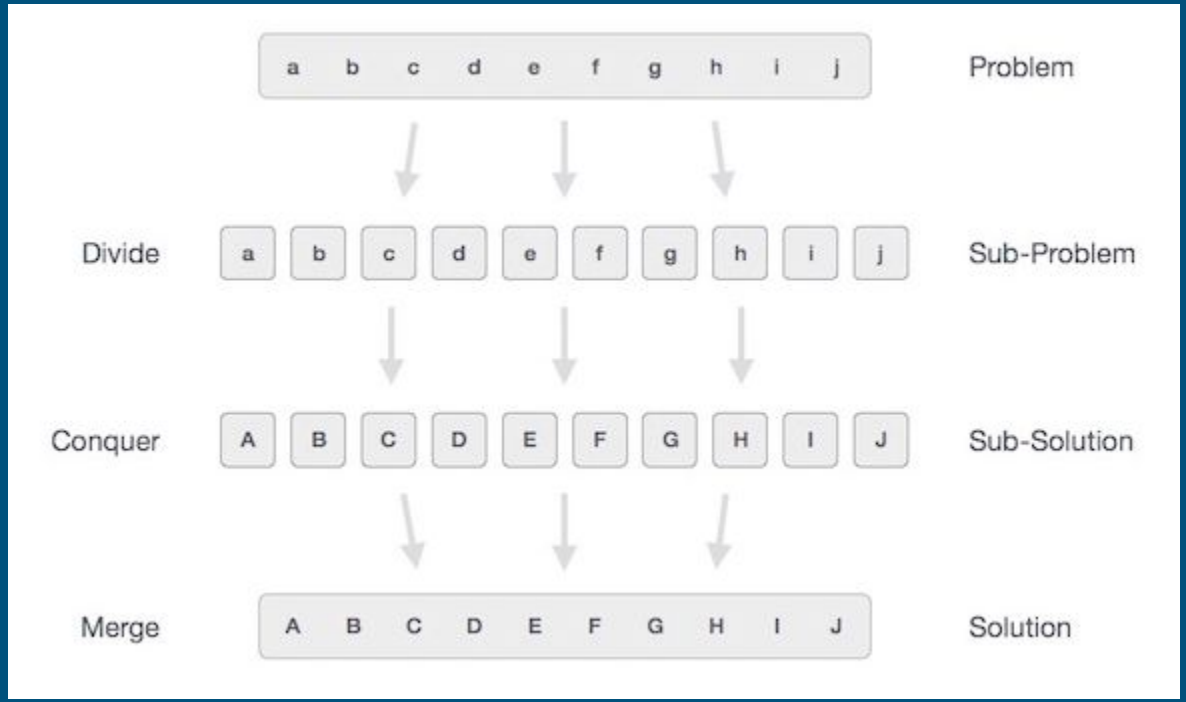
Giới thiệu

Như có thể thấy qua bài toán ban đầu, ngoài 2 cách tiếp cận cơ bàn, chúng ta có một hướng tiếp cận đó là:

- Chia bài toán thành từng phần nhỏ có cấu trúc giống bài toán ban đầu.
- Giải các bài toán nhỏ hơn đó.
- Tổng hợp lại và đưa ra kết quả cuối cùng.

Đó chính là nội dung cơ bản của phương pháp thiết kế thuật toán chia để trị (divide and conquer).

Giới thiệu (tiếp)



Giới thiệu

1. **Chia**: Bài toán ban đầu sẽ được chia thành các bài toán con cho đến khi không thể chia nhỏ được nữa.
2. **Trị**: Tìm phương án để giải quyết cho bài toán con một cách cụ thể
3. **Kết hợp**: kết hợp lời giải của các bài toán nhỏ thành lời giải của bài toán ban đầu.

Ví dụ

Xem xét bài toán tính lũy thừa a^n.

Trong bài toán này, chúng ta sẽ giả sử phép nhân có độ phức tạp là O(1) bất kể độ lớn của hai số.

Ví dụ (tiếp)

```
# Thuật toán thường dùng
def slow_power(int a, int n):
    s = 1
    for i = 0 ... n - 1:
        s = s * a
    return s
# Thuật toán này có độ phức tạp O(n)
```

Ví dụ

Chúng ta có nhận xét:

- Nếu n = 2k, $a^n = a^k * a^k$
- Nếu n = 2k + 1, $a^n = a * a^k * a^k$

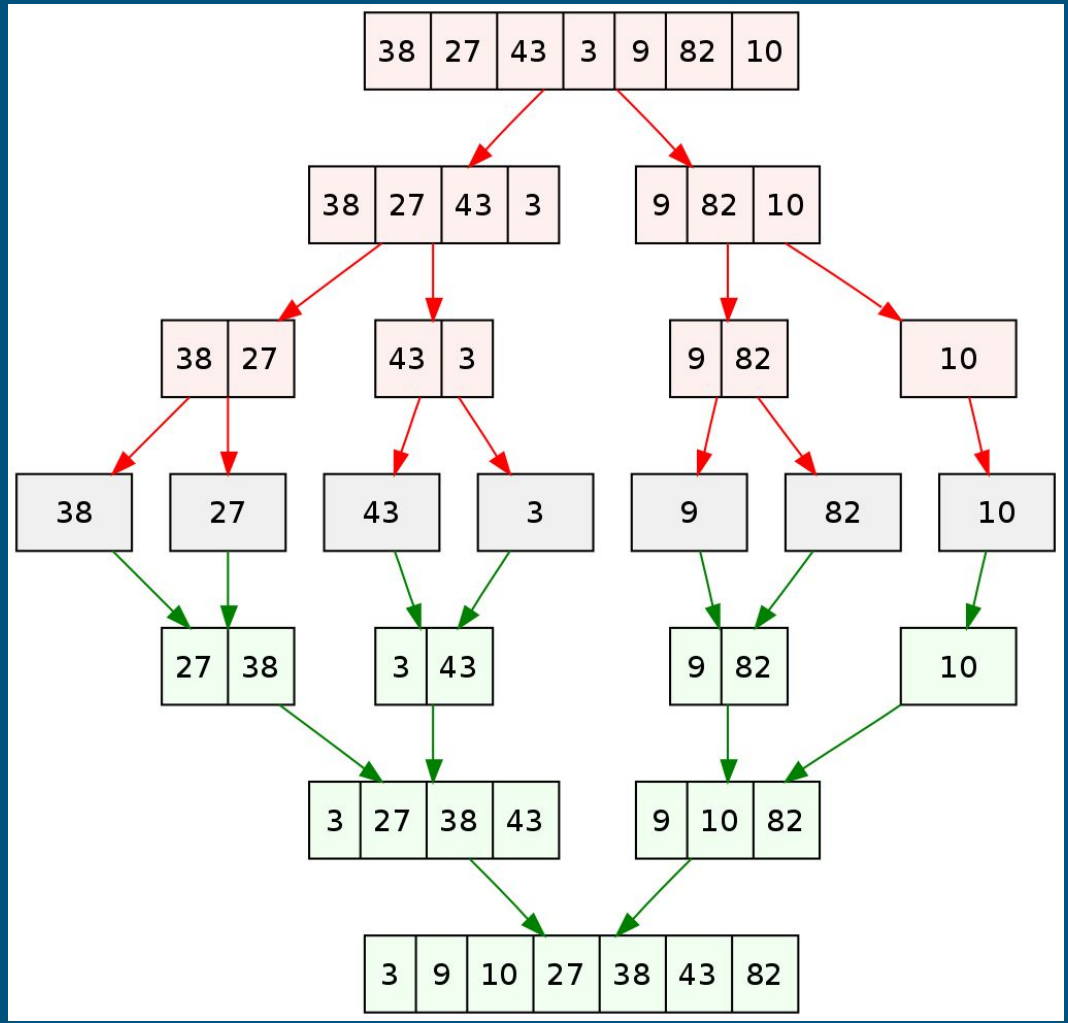
Chúng ta có thể thấy bài toán tính a^n có thể thay bằng bài toán nhỏ hơn là tính a^k.

Ví dụ (tiếp)

```
# chúng ta có thuật toán sau (độ phức tạp O(logn)):
def fast_power(int a, int n):
    if n = 0:
        return 1
    elif n % 2 == 0:      #n chẵn
        return fast_power(a, n // 2) ** 2
    else:                 # n lẻ
        return fast_power(a, n // 2) ** 2 * a
```

Ví dụ (tiếp)

Hình bên minh họa cách thực hiện merge sort (sắp xếp trộn).



Một số thuật toán

Một số thuật toán dùng phương pháp chia để trị:

- Thuật toán Euclid để tìm ước chung lớn nhất.
- Thuật toán Cooley - Tukey để tính biến đổi Fourier.
- Thuật toán nhân hai số lớn Karatsuba.
- Thuật toán sắp xếp trộn (merge sort), sắp xếp nhanh (quick sort)

Ưu điểm

- Giải quyết các vấn đề phức tạp
- Thuật toán hiệu quả
- Tính toán song song
- Tối ưu bộ nhớ
- Kiểm soát lỗi

Vấn đề khi thực hiện

Khi tiến hành cài đặt các thuật toán chia để trị, có một số vấn đề thường gặp:

- Đệ quy
- Kích thước stack
- Chọn trường hợp cơ bản
- Các bài toán con lặp lại