Chia để trị
(Divide and Conquer)

Khoa Công Nghệ Thông Tin,
Trường Đại Học Thủy Lợi.

Ngày 7 tháng 12 năm 2017

Chia để trị

Chia để trị là một mô hình giải quyết bài toán trong đó một bài toán được tạo ra *đơn giản hơn* bằng cách 'chia nhỏ' bài toán đó thành các phần nhỏ hơn và sau đó giải quyết từng phần một

Thông thường gồm 3 bước:

1. CHIA: Phân chia bài toán thành một hoặc nhiều bài toán con nhỏ hơn - thường bằng một nửa hoặc gần một nửa
2. CHINH PHỤC (TRỊ): Giải từng bài toán con nhận được bằng đệ quy, các bài toán con bây giờ đã dễ dàng hơn
3. KẾT HỢP: Kết hợp các lời giải từ các bài toán nhỏ thành lời giải của bài toán đã cho

Các thuật toán chia để trị chuẩn

- Sắp xếp nhanh (Quicksort)
- Sắp xếp trộn (Mergesort)
- Thuật toán Karatsuba
- Thuật toán Strassen
- Nhiều thuật toán cho hình học tính toán
  - Bao lồi (Convex hull)
  - Cặp điểm gần nhất

Ứng dụng của Chia để trị

- *Giải những bài toán khó*: phân nhỏ bài toán thành các bài toán con, giải quyết những trường hợp bình thường và kết hợp những bài toán con thành bài toán ban đầu
- *Tính toán song song*: thích ứng một cách tự nhiên với việc chạy trên những máy tính đa nhân, đặc biệt những hệ thống chia sẻ bộ nhớ nơi mà sự truyền thông dữ liệu giữa các bộ xử lý không cần được lên kế hoạch trước, bởi vì những bài toán con riêng biệt có thể được thực thi trên những bộ xử lý khác nhau
- *Truy cập bộ nhớ*: một cách tự nhiên hướng tới cách sử dụng hiệu quả bộ nhớ cache. Khi một bài toán con là đủ nhỏ, nó và tất cả những bài toán con của nó, về nguyên tắc, có thể được giải quyết bên trong bộ nhớ cache mà không cần truy cập đến bộ nhớ chính chậm hơn
- *Kiểm soát việc làm tròn* (Roundoff): có thể tạo ra nhiều kết quả chính xác hơn một phương pháp lặp với phép toán làm tròn. Ví dụ, một người có thể cộng N số bằng cách sử dụng một vòng lặp đơn giản để cộng dồn mỗi mốc dữ liệu vào một biến đơn, hoặc bằng cách sử dụng một thuật toán Chia để trị (D&C) được gọi là phép cộng theo cặp mà chia nhỏ tập dữ liệu thành hai nửa, tính toán một cách đệ quy tổng của từng nửa, và sau đó cộng hai tổng. Trong khi phương pháp thứ hai thực thi cùng một số lượng phép cộng

Phân tích Thuật toán chia để trị

- Được miêu tả bằng phương trình đệ quy
- Giả sử $T(n)$ là thời gian chạy trên một bài toán kích thước n
- $$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq n_c \\ aT(n/b) + D(n) + C(n) & \text{if } n \geq n_c, \end{cases}$$

  where

  a: số lượng bài toán con
  n/b: kích thước của mỗi bài toán con
  $D(n)$: chi phí của thao tác chia nhỏ
  $C(n)$: chi phí của thao tác kết hợp

Độ phức tạp thời gian

```
1  void solve(int n) {
2      if (n == 0)
3          return;
4
5      solve(n/2);
6      solve(n/2);
7
8      for (int i = 0; i < n; i++) {
9          // some constant time operations
10     }
11 }
```

- Độ phức tạp thời gian của thuật toán chia để trị này là gì?
- Thường giúp mô hình hóa độ phức tạp thời gian như một mối quan hệ đệ quy
  - $T(n) = 2T(n/2) + n$

Độ phức tạp thời gian

- Nhưng làm sao chúng ta giải quyết sự đệ quy như vậy?
- Thông thường đơn giản nhất là sử dụng định lý Master khi có thể áp dụng
  - Nó đưa ra một lời giải cho sự đệ quy dưới dạng $T(n) = aT(n/b) + f(n)$ theo những thuật ngữ tiệm cận
  - Tất cả thuật toán chia để trị được đề cập cho đến bây giờ đều có dạng đệ quy này

- Định lý Master nói rằng $T(n) = 2T(n/2) + n$ có độ phức tạp thời gian tiệm cận $O(n \log n)$

- Phương pháp cây đệ quy cũng rất hữu ích để giải quyết dạng đệ quy này.

Giảm để trị

- Đôi khi chúng ra không thực sự chia nhỏ bài toán thành nhiều bài toán con, mà chỉ là một bài toán con nhỏ hơn
- Thường được gọi là Giảm để trị (decrease and conquer)
- Một ví dụ phổ biến nhất của phương pháp này là tìm kiếm nhị phân

Sắp xếp trộn

- **Chia:** chia dãy n phần tử thành hai bài toán con với $n/2$ phần tử
- **Chinh phục:** sắp xếp hai dãy con một cách đệ quy sử dụng sắp xếp trộn. Nếu chiều dài của một dãy là 1, không làm gì cả bởi vì nó đã theo thứ tự rồi
- **Kết hợp:** trộn hai dãy con đã được sắp xếp để tạo ra lời giải cho bài toán ban đầu

Sắp xếp trộn – Hàm trộn

- Thao tác trộn là thao tác chính (key) trong sắp xếp trộn
- Giả sử các dãy (con) được lưu trữ trong mảng A. Hơn nữa, $A[p \ldots q]$ và $A[q+1 \ldots r]$ là hai dãy con đã được sắp xếp.
- `MERGE(A,p,q,r)` sẽ trộn hai dãy con thành một dãy được sắp xếp $A[p \ldots r]$
- `MERGE(A,p,q,r)` nhận $\Theta(r-p+1)$

```
1  MERGE_SORT(A,p,r){
2      if (p<r) {
3          q=(p+r)/2
4
5          MERGE_SORT(A,p,q)
6          MERGE_SORT(A,p,r)
7          MERGE(A,p,q,r)}
8      }
```

Gọi hàm `MERGE-SORT(A,1,n)` (giả sử n=length(A))

Phân tích Sắp xếp trộn

- **Chia:** $D(n) = \Theta(1)$
- **Chinh phục:** $a = 2, b = 2$, vì vậy $2T(n/2)$
- **Kết hợp:** $C(n) = \Theta(n)$
- $T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$
- $T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$
- $T(n) = \mathcal{O}(n \log n)$ bằng Cây đệ quy hoặc Định lý Master

Tìm kiếm nhị phân

- Chúng ta có một mảng các phần tử **đã được sắp xếp**, và chúng ta muốn kiểm tra liệu mảng đó có chứa một phần tử xác định x không
- Thuật toán:
  1. Trường hợp cơ sở: mảng rỗng, trả về giá trị *false*
  2. So sánh x với phần tử ở vị trí giữa của mảng
  3. Nếu bằng, thì chúng ta đã tìm được x và trả về giá trị *true*
  4. Nếu nhỏ hơn, thì x phải ở nửa trái của mảng
     1. Tìm kiếm nhị phân phần tử này (đệ quy) ở nửa trái
  5. Nếu lớn hơn, thì x phải ở nửa phải của mảng
     1. Tìm kiếm nhị phân phần tử này (đệ quy) ở nửa phải

Tìm kiếm nhị phân

```
1  bool binary_search(const vector<int> &arr,int lo,int hi,int x){
2      if (lo > hi) {
3          return false;
4      }
5
6      int m = (lo + hi) / 2;
7      if (arr[m] == x) {
8          return true;
9      } else if (x < arr[m]) {
10         return binary_search(arr, lo, m - 1, x);
11     } else if (x > arr[m]) {
12         return binary_search(arr, m + 1, hi, x);
13     }
14 }
15
16 binary_search(arr, 0, arr.size() - 1, x);
```

- $T(n) = T(n/2) + 1$
- $O(\log n)$

Tìm kiếm nhị phân - Sử dụng vòng lặp

```cpp
bool binary_search(const vector<int> &arr, int x) {
    int lo = 0,
        hi = arr.size() - 1;

    while (lo <= hi) {
        int m = (lo + hi) / 2;
        if (arr[m] == x) {
            return true;
        } else if (x < arr[m]) {
            hi = m - 1;
        } else if (x > arr[m]) {
            lo = m + 1;
        }
    }

    return false;
}
```

Tìm kiếm nhị phân trên số nguyên

- Đây có thể là ứng dụng nổi tiếng nhất của tìm kiếm nhị phân, nhưng nó không chỉ đơn thuần là một ứng dụng
- Tổng quát hơn, chúng ta có một dãy số $p : \{0, \ldots, n-1\} \to \{T, F\}$ mà có thuộc tính: nếu $p(i) = T$, thì $p(j) = T$ với tất cả $j > i$
- Mục đích của chúng ta là tìm chỉ số nhỏ nhất j sao cho $p(j) = T$ nhanh nhất có thể

i	0	1	$\cdots$	$j-1$	j	$j+1$	$\cdots$	$n-2$	$n-1$
$p(i)$	F	F	$\cdots$	F	T	T	$\cdots$	T	T

- Chúng ta có thể làm điều này trong thời gian $O(\log(n) \times f)$, với f là chi phí của việc đánh giá dãy số p, theo cách tương tự như tìm kiếm nhị phân một mảng

Tìm kiếm nhị phân trên số nguyên

```
1  int lo = 0,
2      hi = n - 1;
3
4  while (lo < hi) {
5      int m = (lo + hi) / 2;
6
7      if (p(m)) {
8          hi = m;
9      } else {
10         lo = m + 1;
11     }
12 }
13
14 if (lo == hi && p(lo)) {
15     printf("lowest index is %d\n", lo);
16 } else {
17     printf("no such index\n");
18 }
```

Tìm kiếm nhị phân trên số nguyên

- Tìm chỉ số của x trong mảng đã được sắp xếp *arr*

```
1 bool p(int i) {
2     return arr[i] >= x;
3 }
```

- Sau đây, ta sẽ thấy cách sử dụng thuật toán này theo cách khác

Tìm kiếm nhị phân trên số thực

- Một phiên bản chung hơn của tìm kiếm nhị phân là trên các số thực
- Chúng ta có một dãy số $p : [lo, hi] \rightarrow \{T, F\}$ mà có thuộc tính là nếu $p(i) = T$, thì $p(j) = T$ với tất cả $j > i$
- Mục đích của chúng ta là tìm số thực nhỏ nhất j sao cho $p(j) = T$ nhanh nhất có thể
- Do chúng ta đang làm việc với những số thực (theo giả thiết), giá trị $[lo, hi]$ của chúng ta có thể được chia đôi vô hạn nhiều lần mà không bao giờ trở thành một số thực đơn
- Thay vào đó sẽ chấp nhận được khi tìm một số thực j' rất gần với câu trả lời đúng j, mà không sai khác quá $EPS = 2^{-30}$
- Chúng ta có thể làm điều này trong thời gian $O(\log(\frac{hi-lo}{EPS}))$ tương tự như cách chúng ta tìm kiếm nhị phân một mảng

Tìm kiếm nhị phân trên số thực

```
1  double EPS = 1e-10,
2         lo = -1000.0,
3         hi = 1000.0;
4
5  while (hi - lo > EPS) {
6      double mid = (lo + hi) / 2.0;
7
8      if (p(mid)) {
9          hi = mid;
10     } else {
11         lo = mid;
12     }
13 }
14
15 printf("%0.10lf\n", lo);
```

Tìm kiếm nhị phân trên số thực

- Điều này có rất nhiều ứng dụng toán học hay
- Tìm căn bậc hai của x

```
1 bool p(double j) {
2     return j*j >= x;
3 }
```

- Tìm nghiệm của một hàm số tăng $f(x)$

```
1 bool p(double x) {
2     return f(x) >= 0.0;
3 }
```

- Thuật toán này cũng được tham chiếu đến phương pháp Bisection

Bài tập thực hành

Cái bánh (Pie)

Tìm kiếm nhị phân câu trả lời

- Thuật toán này có thể khó tìm được trực tiếp lời giải tối ưu, như chúng ta đã thấy trong bài toán ví dụ
- Mặt khác, có thể dễ dàng kiểm tra liệu một vài x có là lời giải hay không
- Một phương pháp sử dụng tìm kiếm nhị phân để tìm lời giải nhỏ nhất hoặc lớn nhất cho một bài toán
- Chỉ có thể ứng dụng được khi bài toán có thuộc tính tìm kiếm nhị phân: nếu i là một lời giải, thì với mọi $j > i$ cũng thế
- $p(i)$ kiểm tra liệu i có là một lời giải, sau đó chúng ta đơn giản áp dụng tìm kiếm nhị phân trên p để lấy lời giải nhỏ nhất hoặc lớn nhất

Những dạng khác của Chia để trị

- Tìm kiếm nhị phân rất hữu ích, có thể được sử dụng để tìm kiếm những lời giải đơn giản và hiệu quả cho nhiều bài toán
- Nhưng tìm kiếm nhị phân chỉ là một ví dụ của chia để trị
- Hãy cùng khám phá hai ví dụ nữa

Lũy thừa nhị phân

- Chúng ta muốn tính x^n, với x, n là những số nguyên
- Giả sử chúng ta không có phương thức dựng sẵn pow
- Phương thức đơn giản

```
1  int pow(int x, int n) {
2      int res = 1;
3      for (int i = 0; i < n; i++) {
4          res = res * x;
5      }
6
7      return res;
8  }
```

- Độ phức tạp của thuật toán là $O(n)$, nhưng điều gì xảy ra nếu chúng ta muốn tính với n lớn một cách hiệu quả?

Lũy thừa nhị phân

- Hãy sử dụng chia để trị
- Chú ý 3 đặc điểm sau:
  - $x^0 = 1$
  - $x^n = x \times x^{n-1}$
  - $x^n = x^{n/2} \times x^{n/2}$
- Hoặc dưới dạng hàm của chúng ta:
  - $pow(x, 0) = 1$
  - $pow(x, n) = x \times pow(x, n-1)$
  - $pow(x, n) = pow(x, n/2) \times pow(x, n/2)$
- $pow(x, n/2)$ được sử dụng 2 lần, nhưng chúng ta chỉ cần tính nó một lần:
  - $pow(x, n) = pow(x, n/2)^2$

Lũy thừa nhị phân

- Hãy thử sử dụng những đặc điểm này để tính câu trả lời một cách đệ quy

```
1 int pow(int x, int n) {
2     if (n == 0) return 1;
3     return x * pow(x, n - 1);
4 }
```

Lũy thừa nhị phân

- Hãy thử sử dụng những đặc điểm này để tính câu trả lời một cách đệ quy

```
1 int pow(int x, int n) {
2     if (n == 0) return 1;
3     return x * pow(x, n - 1);
4 }
```

- Điều này hiệu quả như thế nào?
  - $T(n) = 1 + T(n-1)$

Lũy thừa nhị phân

- Hãy thử sử dụng những đặc điểm này để tính câu trả lời một cách đệ quy

```
1 int pow(int x, int n) {
2     if (n == 0) return 1;
3     return x * pow(x, n - 1);
4 }
```

- Điều này hiệu quả như thế nào?
  - $T(n) = 1 + T(n-1)$
  - $O(n)$

Lũy thừa nhị phân

- Hãy thử sử dụng những đặc điểm này để tính câu trả lời một cách đệ quy

```
1 int pow(int x, int n) {
2     if (n == 0) return 1;
3     return x * pow(x, n - 1);
4 }
```

- Điều này hiệu quả như thế nào?
  - $T(n) = 1 + T(n-1)$
  - $O(n)$
  - Vẫn còn chậm ...

Lũy thừa nhị phân

- Đặc điểm thứ 3 về điều gì?
  - $n/2$ không phải là số nguyên khi n là số lẻ, vì vậy chỉ sử dụng nó khi n là số chẵn

```
1  int pow(int x, int n) {
2      if (n == 0) return 1;
3      if (n % 2 != 0) return x * pow(x, n - 1);
4      int st = pow(x, n/2);
5      return st * st;
6  }
```

- Điều này hiệu quả như thế nào?

Lũy thừa nhị phân

- Đặc điểm thứ 3 về điều gì?
  - $n/2$ không phải là số nguyên khi n là số lẻ, vì vậy chỉ sử dụng nó khi n là số chẵn

```
1  int pow(int x, int n) {
2      if (n == 0) return 1;
3      if (n % 2 != 0) return x * pow(x, n - 1);
4      int st = pow(x, n/2);
5      return st * st;
6  }
```

- Điều này hiệu quả như thế nào?
  - $T(n) = 1 + T(n-1)$ nếu n là số lẻ
  - $T(n) = 1 + T(n/2)$ nếu n là số chẵn

Lũy thừa nhị phân

- Đặc điểm thứ 3 về điều gì?
  - $n/2$ không phải là số nguyên khi n là số lẻ, vì vậy chỉ sử dụng nó khi n là số chẵn

```
1  int pow(int x, int n) {
2      if (n == 0) return 1;
3      if (n % 2 != 0) return x * pow(x, n - 1);
4      int st = pow(x, n/2);
5      return st * st;
6  }
```

- Điều này hiệu quả như thế nào?
  - $T(n) = 1 + T(n-1)$ nếu n là số lẻ
  - $T(n) = 1 + T(n/2)$ nếu n là số chẵn
  - Do $n-1$ là số chẵn khi n là số lẻ:
  - $T(n) = 1 + 1 + T((n-1)/2)$ nếu n là số lẻ

Lũy thừa nhị phân

- Đặc điểm thứ 3 về điều gì?
  - $n/2$ không phải là số nguyên khi n là số lẻ, vì vậy chỉ sử dụng nó khi n là số chẵn

```
1  int pow(int x, int n) {
2      if (n == 0) return 1;
3      if (n % 2 != 0) return x * pow(x, n - 1);
4      int st = pow(x, n/2);
5      return st * st;
6  }
```

- Điều này hiệu quả như thế nào?
  - $T(n) = 1 + T(n-1)$ nếu n là số lẻ
  - $T(n) = 1 + T(n/2)$ nếu n là số chẵn
  - Do $n-1$ là số chẵn khi n là số lẻ:
  - $T(n) = 1 + 1 + T((n-1)/2)$ nếu n là số lẻ
  - $O(\log n)$
  - Nhanh!

Lũy thừa nhị phân

- Chú ý rằng x không bắt buộc phải là số nguyên, và $\star$ không bắt buộc phải là phép nhân số nguyên
- Nó cũng hoạt động với:
  - Tính x^n, khi x là một số thực, và $\star$ là phép nhân số thực
  - Tính A^n, khi A là một ma trận, và $\star$ là phép nhân ma trận
  - Tính $x^n \pmod m$, khi x là một ma trận, và $\star$ là phép nhân số nguyên đồng dư m
  - Tính $x \star x \star \cdots \star x$, khi x là phần tử bất kỳ, và $\star$ là toán tử kết hợp bất kỳ
- Tất cả những điều này có thể được thực hiện trong thời gian $O(\log(n) \times f)$, với f là chi phí để việc thực hiện một ứng dụng của toán tử $\star$

Từ Fibonnaci

- Nhớ lại rằng dãy Fibonnaci có thể được định nghĩa như sau:
  - $\mathrm{fib}_1 = 1$
  - $\mathrm{fib}_2 = 1$
  - $\mathrm{fib}_n = \mathrm{fib}_{n-2} + \mathrm{fib}_{n-1}$
- Chúng ta có dãy $1, 1, 2, 3, 5, 8, 13, 21, \ldots$

- Có rất nhiều tổng quát hóa của dãy Fibonnaci
- Một trong số chúng bắt đầu với những số khác, chẳng hạn:
  - $f_1 = 5$
  - $f_2 = 4$
  - $f_n = f_{n-2} + f_{n-1}$
- Chúng ta có dãy $5, 4, 9, 13, 22, 35, 57, \ldots$

- Điều gì xảy ra nếu chúng ta bắt đầu với những thứ khác không phải là số?

Từ Fibonnaci

- Hãy thử bắt đầu với một cặp xâu ký tự, và gọi $+$ biểu thị phép nối xâu
  - $g_1 = A$
  - $g_2 = B$
  - $g_n = g_{n-2} + g_{n-1}$
- Bây giờ chúng ta có trình tự những xâu ký tự sau:
  - A
  - B
  - AB
  - BAB
  - $ABBAB$
  - $BABABBAB$
  - $ABBABBABABBAB$
  - $BABABBABABBABBABABBAB$
  - ...

Từ Fibonnaci

- Độ dài của g_n là bao nhiêu?
  - $\text{len}(g_1) = 1$
  - $\text{len}(g_2) = 1$
  - $\text{len}(g_n) = \text{len}(g_{n-2}) + \text{len}(g_{n-1})$
- Bạn có nhìn quen?
- $\text{len}(g_n) = \text{fib}_n$
- Vì vậy những xâu ký tự nhanh chóng trở nên rất lớn
  - $\text{len}(g_{10}) = 55$
  - $\text{len}(g_{100}) = 354224848179261915075$
  - $\text{len}(g_{1000}) =$

    434665576869374564356885276750406258025646605173717804024817290895365554179490518904038798400792551692959225930803226347752096896232398733224711616429964409065331879382989696499285160037044761377951668492828875

Từ Fibonnaci

- Nhiệm vụ: Tính ký tự thứ i trong g_n

Từ Fibonnaci

- Nhiệm vụ: Tính ký tự thứ i trong g_n
- Đơn giản để thực hiện trong thời gian $O(\text{len}(n))$, nhưng điều đó trở nên cực chậm với n lớn

Từ Fibonnaci

- Nhiệm vụ: Tính ký tự thứ i trong g_n
- Đơn giản để thực hiện trong thời gian $O(\text{len}(n))$, nhưng điều đó trở nên cực chậm với n lớn
- Có thể thực hiện trong thời gian $O(n)$ bằng cách sử dụng chia để trị

Bài tập thực hành

Từ Fibonnaci