Chia để trị

THUẬT TOÁN ỨNG DỤNG

Đỗ Phan Thuận
thuandp.sinhvien@gmail.com

Bộ môn Khoa Học Máy Tính, Viện CNTT & TT,
Trường Đại Học Bách Khoa Hà Nội.

Ngày 23 tháng 3 năm 2020

Các mô hình giải bài cơ bản

Các phương pháp căn bản xây dựng lời giải cho từng dạng bài toán

- Duyệt toàn bộ
- Chia để trị
- Quy hoạch động
- Tham lam

Mỗi mô hình ứng dụng cho nhiều loại bài toán khác nhau

Chia để trị

Chia để trị là một mô hình giải bài theo hướng làm dễ bài toán đi bằng cách chia thành các phần nhỏ hơn và xử lý từng phần một

Thông thường làm theo 3 bước chính:

1. CHIA: chia bài toán thành một hay nhiều bài toán con - thường hay chia một nửa hoặc gần một nửa
2. XỬ LÝ: giải đệ quy mỗi bài toán con - mỗi bài toán cần giải trở nên dễ hơn
3. KẾT HỢP: kết hợp lời giải các bài toán con thành lời giải bài toán ban đầu

Một số bài toán chia để trị cơ bản

- Sắp xếp nhanh (Quick sort)
- Sắp xếp trộn (Merge sort)
- Thuật toán Karatsuba nhân nhanh số lớn
- Thuật toán Strassen nhân ma trận
- Rất nhiều thuật toán trong tính toán hình học
  - Bao lồi (Convex hull)
  - Cặp điểm gần nhất (Closest pair of points)

Ứng dụng của thuật toán chia để trị

- Giải các bài toán khó: bằng cách chia nhỏ thành cách bài toán nhỏ dễ giải hơn và kết hợp các lời giải bài toán nhỏ lại thành lời giải bài toán ban đầu
- Tính toán song song: tính toán trên nhiều máy tính, nhiều vi xử lý, tính toán trên dàn/lưới máy tính. Trong trường hợp này độ phức tạp chi phí truyền thông giữa các phần tính toán là rất quan trọng
- Truy cập bộ nhớ: bài toán được chia nhỏ đến khi có thể giải trực tiếp trên bộ nhớ đệm sẽ cho thời gian thực hiện nhanh hơn nhiều so với việc truy cập sử dụng bộ nhớ chính
- Xử lý dữ liệu: dữ liệu lớn được chia thành cách phần nhỏ để lưu trữ và xử lý dữ liệu
- ...

Phân tích độ phức tạp thuật toán chia để trị

- Được mô tả bởi một công thức truy hồi
- Gọi $T(n)$ là thời gian tính toán của bài toán kích thước n
- $$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq n_c \\ aT(n/b) + D(n) + C(n) & \text{if } n \geq n_c, \end{cases}$$

  với

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

Ví dụ

```
void solve(int n) {
    if (n == 0)
        return;

    solve(n/2);
    solve(n/2);

    for (int i = 0; i < n; i++) {
        // mot so cau lenh don
    }
}
```

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

Chia để trị: Độ phức tạp thuật toán

- Nhưng làm thế nào để giải được công thức truy hồi này?
- Thường đơn giản nhất là sử dụng định lý thợ để giải
  - Định lý thợ cho phép đưa ra lời giải cho công thức đệ quy dạng $T(n) = aT(n/b) + f(n)$ theo ký pháp hàm tiệm cận
  - Đa phần các thuật toán chia để trị thông dụng có công thức truy hồi theo mẫu này
- Định lý thợ cho biết $T(n) = 2T(n/2) + n$ có thời gian tính $O(n \log n)$
- Nên thuộc định lý thợ
- Phương pháp cây đệ quy cũng rất hữu ích để giải công thức truy hồi

Định lý thợ rút gọn

$T(n) = aT(n/b) + cn^k$, với a, b, c, k là các hằng số dương, và $a \geq 1, b \geq 2$, ta có

$T(n) =$

- $O(n^{\log_b a})$, nếu $a > b^k$,
- $O(n^k \log n)$, nếu $a = b^k$,
- $O(n^k)$, nếu $a < b^k$,

Sắp xếp trộn

- **CHIA:** chia dãy n phần tử thành 2 dãy con mỗi dãy $n/2$ phần tử
- **XỬ LÝ:** sắp xếp mỗi dãy con sử dụng lời gọi đệ qui thuật toán sắp xếp trộn, đến khi độ dài dãy là 1 thì dừng
- **KẾT HỢP** trộn 2 dãy con đã được sắp xếp lại thành dãy kết quả

Hàm trộn

- Hàm trộn là là hàm thiết yếu trong thuật toán sắp xếp trộn
- Giả sử các dãy con được lưu trữ trong mảng A. Hai dãy con $A[p \ldots q]$ và $A[q+1 \ldots r]$ đã được sắp xếp
- `MERGE(A,p,q,r)` sẽ trộn 2 dãy con thành dãy kết quả $A[p \ldots r]$
- `MERGE(A,p,q,r)` tốn thời gian $\Theta(r-p+1)$

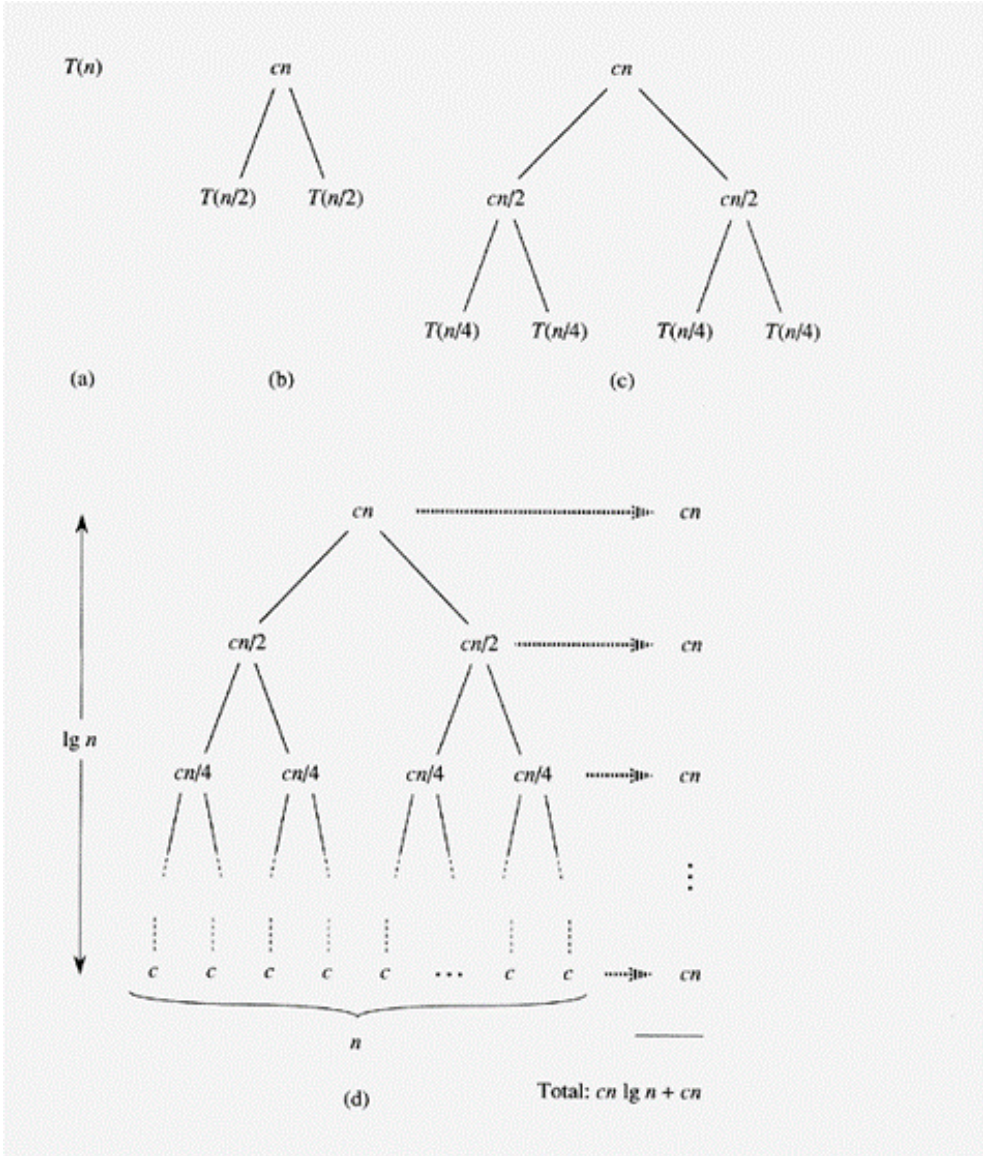
```
1  MERGE_SORT(A,p,r){
2      if (p<r) {
3          q=(p+r)/2
4
5          MERGE_SORT(A,p,q)
6          MERGE_SORT(A,q+1,r)
7          MERGE(A,p,q,r)}
8      }
```

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

Phân tích độ phức tạp thuật toán Sắp xếp trộn

- **CHIA:** $D(n) = \Theta(1)$
- **XỬ LÝ:** $a = 2, b = 2$, so $2T(n/2)$
- **KẾT HỢP:** $C(n) = \Theta(n)$
- $T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$
- $T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$
- $T(n) = \mathcal{O}(n \log n)$ theo Định lý thợ hoặc Phương pháp Cây đệ qui

Cây đệ qui phân tích độ phức tạp MERGE-SORT



Giảm để trị (Decrease and conquer)

- Đôi khi không cần chia bài toàn thành nhiều bài toán con, mà chỉ giảm về một bài toán con kích thước nhỏ hơn
- Thường gọi là Giảm để trị
- Ví dụ thông dụng nhất là Tìm kiếm nhị phân

Tìm kiếm nhị phân

- Cho một mảng các phần tử **đã được sắp xếp**, hãy kiểm tra xem mảng có chứa phần tử x không
- Thuật toán:
  1. Trường hợp biên: mảng rỗng, trả lời KHÔNG
  2. So sánh x với phần tử ở vị trí giữa mảng
  3. Nếu bằng, tìm thấy x và trả lời CÓ
  4. Nếu nhỏ hơn, x chắc chắn nằm bên nửa trái mảng
     1. Tìm kiếm nhị phân (đệ quy) tiếp nửa trái mảng
  5. Nếu lớn hơn, x chắc chắn nằm bên nửa phải mảng
     1. Tìm kiếm nhị phân (đệ quy) tiếp nửa phải mảng

Tìm kiếm nhị phân

```cpp
bool binary_search(const vector<int> &arr,int lo,int hi,int x){
    if (lo > hi) {
        return false;
    }

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

binary_search(arr, 0, arr.size() - 1, x);
```

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

Tìm kiếm nhị phân trên các số nguyên

- Đây có lẽ là ứng dụng phổ biến nhất của tìm kiếm nhị phân
- Cụ thể, cho hàm $p : \{0, \ldots, n-1\} \to \{T, F\}$ thỏa mãn nếu $p(i) = T$, thì $p(j) = T$ với mọi $j > i$
- Yêu cầu tìm chỉ số j nhỏ nhất sao cho $p(j) = T$ càng nhanh càng tốt

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

- Có thể thực hiện trong $O(\log(n) \times f)$, với f là giá của việc đánh giá hàm p

Tìm kiếm nhị phân trên các 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("chi so nho nhat tim duoc la %d\n", lo);
16 } else {
17     printf("khong ton tai phan tu %d \n", x);
18 }
```

Tìm kiếm nhị phân trên các số nguyên

- Tìm vị trí của x trong mảng đã sắp xếp *arr*

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

- Cách sử dụng khác ở phần sau

Tìm kiếm nhị phân trên các số thực

- Đây là phiên bản tổng quát hơn của tìm kiếm nhị phân
- Cho hàm $p : [lo, hi] \to \{T, F\}$ thỏa mãn nếu $p(i) = T$, thì $p(j) = T$ với mọi $j > i$
- Yêu cầu tìm số thực nhỏ nhất j sao cho $p(j) = T$ càng nhanh càng tốt
- Do làm việc với số thực, khoảng $[lo, hi]$ có thể bị chia vô hạn lần mà không dừng ở một số thực cụ thể
- Thay vào đó có thể tìm một số thực j' rất sát với lời giải đúng j, sai số trong khoảng $EPS = 2^{-30}$
- Có thể làm được trong thời gian $O(\log(\frac{hi-lo}{EPS}))$ tương tự cách làm tìm kiếm nhị phân trên mảng

Tìm kiếm nhị phân trên các 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 các số thực

- Có nhiều ứng dụng thú vị
- Tìm căn bậc hai của x

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

- Tìm nghiệm của hàm $f(x)$

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

- Đây cũng được gọi là phương pháp chia đôi trong phương pháp tính (Bisection method)

Bài toán ví dụ

- Problem C from NWERC 2006:

Tìm kiếm nhị phân câu trả lời

- Một số bài toán có thể khó tìm ra lời giải tối ưu một cách trực tiếp,
- Mặt khác, dễ dàng kiểm tra một số x nào đó có phải là lời giải không
- Phương pháp sử dụng tìm kiếm nhị phân để tìm lời giải nhỏ nhất hoặc lớn nhất của một bài toán
- Chỉ áp dụng được khi bài toán có tính chất tìm kiếm nhị phân: nếu i là một lời giải, thì tất cả $j > i$ cũng là lời giải
- $p(i)$ kiểm tra nếu i là một lời giải, thì có thể áp dụng một cách đơn giản tìm kiếm nhị phân trên p để nhận được lời giải nhỏ nhất hoặc lớn nhất

Các loại chia để trị khác

- Tìm kiếm nhị phân rất hữu ích, có thể dùng để xây dựng các bài giải đơn giản và hiệu quả
- Tuy nhiên tìm kiếm nhị phân là chỉ là một ví dụ của chia để trị
- Hãy xem tiếp 2 ví dụ nữa

Nhị phân hàm mũ (Binary exponentiation)

- Yêu cầu tính x^n, với x, n là các số nguyên
- Giả thiết ta không có phương thức pow
- Phương pháp trực tiếp:

```
int pow(int x, int n) {
    int res = 1;
    for (int i = 0; i < n; i++) {
        res = res * x;
    }

    return res;
}
```

- Độ phức tạp $O(n)$, tuy nhiên với n lớn thì sao?

Nhị phân hàm mũ

- Hãy sử dụng chia để trị
- Để ý 3 đẳng thức sau:
  - $x^0 = 1$
  - $x^n = x \times x^{n-1}$
  - $x^n = x^{n/2} \times x^{n/2}$
- Hoặc theo ngôn ngữ hàm:
  - $pow(x, 0) = 1$
  - $pow(x, n) = x \times pow(x, n-1)$
  - $pow(x, n) = pow(x, n/2) \times pow(x, n/2)$
- $pow(x, n/2)$ được sử dụng 2 lần, nhưng ta chỉ cần tính 1 lần:
  - $pow(x, n) = pow(x, n/2)^2$

Nhị phân hàm mũ

- Hãy sử dụng các đẳng thức đó để tìm câu trả lời theo cách đệ quy

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

Nhị phân hàm mũ

- Hãy sử dụng các đẳng thức đó để tìm câu trả lời theo cách đệ quy

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

- Độ phức tạp?
  - $T(n) = 1 + T(n-1)$

Nhị phân hàm mũ

- Hãy sử dụng các đẳng thức đó để tìm câu trả lời theo cách đệ quy

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

- Độ phức tạp?
  - $T(n) = 1 + T(n-1)$
  - $O(n)$

Nhị phân hàm mũ

- Hãy sử dụng các đẳng thức đó để tìm câu trả lời theo cách đệ quy

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

- Độ phức tạp?
  - $T(n) = 1 + T(n-1)$
  - $O(n)$
  - Vẫn chậm như trước...

Nhị phân hàm mũ

- Để ý đẳng thức thứ 3:
  - $n/2$ không là số nguyên khi n lẻ, vì vậy chỉ sử dụng nó khi n 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 }
```

- Độ phức tạp?

Nhị phân hàm mũ

- Để ý đẳng thức thứ 3:
  - $n/2$ không là số nguyên khi n lẻ, vì vậy chỉ sử dụng nó khi n 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  }
```

- Độ phức tạp?
  - $T(n) = 1 + T(n-1)$ nếu n lẻ
  - $T(n) = 1 + T(n/2)$ nếu n chẵn

Nhị phân hàm mũ

- Để ý đẳng thức thứ 3:
  - $n/2$ không là số nguyên khi n lẻ, vì vậy chỉ sử dụng nó khi n 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  }
```

- Độ phức tạp?
  - $T(n) = 1 + T(n-1)$ nếu n lẻ
  - $T(n) = 1 + T(n/2)$ nếu n chẵn
  - Do $n-1$ chẵn khi n lẻ:
  - $T(n) = 1 + 1 + T((n-1)/2)$ nếu n lẻ

Nhị phân hàm mũ

- Để ý đẳng thức thứ 3:
  - $n/2$ không là số nguyên khi n lẻ, vì vậy chỉ sử dụng nó khi n 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  }
```

- Độ phức tạp?
  - $T(n) = 1 + T(n-1)$ nếu n lẻ
  - $T(n) = 1 + T(n/2)$ nếu n chẵn
  - Do $n-1$ chẵn khi n lẻ:
  - $T(n) = 1 + 1 + T((n-1)/2)$ nếu n lẻ
  - $O(\log n)$
  - Nhanh!

Nhị phân hàm mũ

- Để ý là x không nhất thiết là số nguyên và $\star$ không nhất thiết là phép nhân số nguyên...
- Cũng dùng được cho:
  - Tính x^n, với x là số thực và $\star$ là phép nhân số thưc
  - Tính A^n, với A là một ma trận và $\star$ là phép nhân ma trận
  - Tính $x^n \pmod m$, với x là một số nguyên và $\star$ là phép nhân số nguyên lấy mod m
  - Tính $x \star x \star \cdots \star x$, với x là bất kỳ loại phần tử gì và $\star$ là một toán tử phù hợp
- Tất cả có thể giải trong $O(\log(n) \times f)$, với f là giá để thực hiện một toán tử $\star$

Số Fibonacci

- Nhắc lại dãy Fibonacci được định nghĩa như sau:
  - $\text{fib}_1 = 1$
  - $\text{fib}_2 = 1$
  - $\text{fib}_n = \text{fib}_{n-2} + \text{fib}_{n-1}$
- Ta có dãy $1, 1, 2, 3, 5, 8, 13, 21, \ldots$

- Có rất nhiều biến thể của dãy Fibonacci
- Một kiểu là cùng công thức nhưng bắt đầu bởi các số khác, ví dụ:
  - $f_1 = 5$
  - $f_2 = 4$
  - $f_n = f_{n-2} + f_{n-1}$
- Ta có dãy $5, 4, 9, 13, 22, 35, 57, \ldots$

- Với những loại phần tử không phải số thì sao?

Số Fibonacci

- Thử với một cặp xâu, và đặt $+$ là phép toán ghép xâu:
  - $g_1 = A$
  - $g_2 = B$
  - $g_n = g_{n-2} + g_{n-1}$
- Ta thu được dãy các xâu:
  - A
  - B
  - AB
  - BAB
  - $ABBAB$
  - $BABABBAB$
  - $ABBABBABABBAB$
  - $BABABBABABBABBABABBAB$
  - ...

Số Fibonacci

- g_n dài bao nhiêu?
  - $\text{len}(g_1) = 1$
  - $\text{len}(g_2) = 1$
  - $\text{len}(g_n) = \text{len}(g_{n-2}) + \text{len}(g_{n-1})$
- Trông quen thuộc?
- $\text{len}(g_n) = \text{fib}_n$
- Vì vậy các xâu trở nên rất lớn rất nhanh
  - $\text{len}(g_{10}) = 55$
  - $\text{len}(g_{100}) = 354224848179261915075$
  - $\text{len}(g_{1000}) =$

    434665576869374564356885276750406258025646605173717804024817290895365554179490518904038798400792551692959225930803226347752096896232398733224711616429964409065331879382989696499285160037044761377951668492828875

Số Fibonacci

- Nhiệm vụ: Hãy tính ký tự thứ i trong g_n

Số Fibonacci

- Nhiệm vụ: Hãy tính ký tự thứ i trong g_n
- Dễ dàng thực hiện trong $O(\text{len}(n))$, nhưng sẽ cực kỳ chậm với n lớn

Số Fibonacci

- Nhiệm vụ: Hãy tính ký tự thứ i trong g_n
- Dễ dàng thực hiện trong $O(\text{len}(n))$, nhưng sẽ cực kỳ chậm với n lớn
- Có thể giải trong $O(n)$ sử dụng chia để trị