ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

THUẬT TOÁN ỨNG DỤNG

THUẬT TOÁN CHIA ĐỂ TRỊ

ONE LOVE. ONE FUTURE.

NỘI DUNG

- Chia để trị
- Giảm để trị
- Một số loại chia để trị thông dụng khác

NỘI DUNG

- **Chia để trị**
- Giảm để trị
- Một số loại chia để trị thông dụng khác

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

Mô hình chung của chia để trị

```
void DC(int n) {
    if (n <= n0) {
        [Giai_bai_toan_con_mot_cach_truc_tiep];
    } else {
        [Chia_Bai_Toan_Thanh_a_Bai_Toan_Con_Kich_Thuoc_n/b];
        [foreach Moi_Bai_Toan_Trong_a_Bai_Toan_Con] {
            call DC(n/b);
        }
        [Tong_Hop_Loi_Giai_Cua_a_Bai_Toan_Con];
        return solution;
    }
}
```

- n0 là kích thước nhỏ nhất của bài toán con (bước neo đệ qui), được giải trực tiếp
- a là số lượng bài toán con cần giải
- b liên quan đến kích thước của bài toán con được chia

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

Phân tích độ phức tạp thuật toán chia để trị

```
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
    }
}
```

- Công thức: 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 qui 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 độ phức tạp $O(n\log n)$
- Nên thuộc định lý thợ
- Phương pháp cây đệ qui cũng rất hữu ích để giải công thức truy hồi

Định lý thợ rút gọn

$T(n) = aT(n/b) + n^k$ 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 hai dãy, mỗi dãy gồm 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...q] và A[q+1...r] đã được sắp xếp
- Merge(A, p, q, r) sẽ trộn 2 dãy con thành dãy kết quả
- Merge(A, p, q, r) tốn thời gian O(r – p + 1)

```
MergeSort(A, p, r) {
    if (p < r) {
        q = (p + r)/2;
        MergeSort(A, p, q);
        MergeSort(A, q + 1, r);
        Merge(A, p, q, r);
    }
}
```

- Gọi hàm MergeSort(A, 1, n), với n = length(A)

Phân tích độ phức tạp thuật toán Sắp xếp trộn

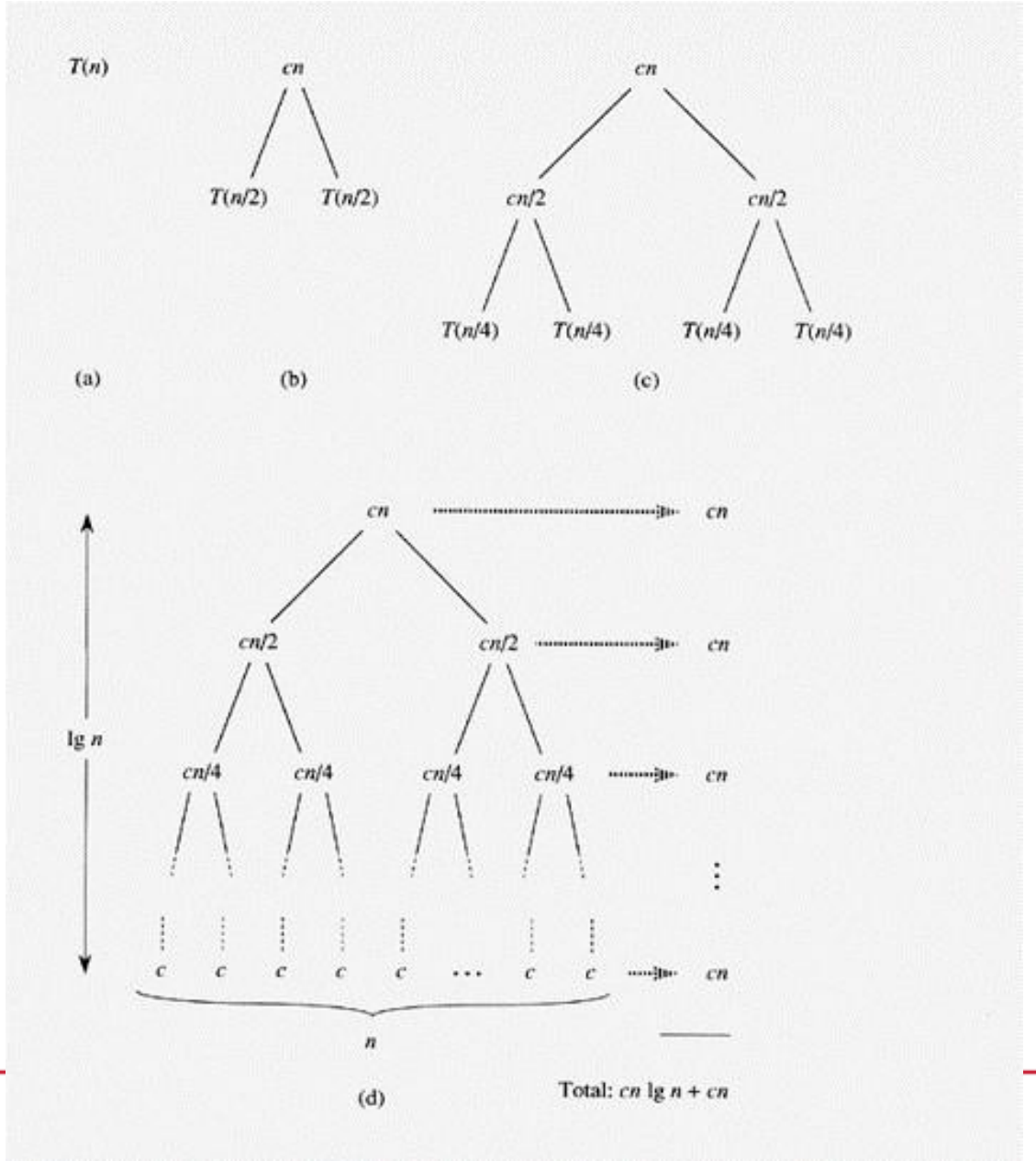
- Chia: $D(\mathrm{n}) = \theta(1)$
- Xử lý: $\mathrm{a} = 2, \mathrm{b} = 2$, nên là $2\mathrm{T}(\mathrm{n}/2)$
- Kết hợp: $\mathrm{C}(\mathrm{n}) = \theta(\mathrm{n})$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

- $\mathrm{T}(\mathrm{n}) = \mathrm{O}(\mathrm{nlogn})$ theo định lý thợ hoặc cây đệ quy

Cây đệ qui phân tích độ phức tạp Merge_Sort



Đoạn con có tổng lớn nhất

- Cho một mảng số nguyên A[1], A[2], ..., A[n]. Hãy tìm một đoạn trong mảng có trọng số lớn nhất, nghĩa là tổng các số trong đoạn là lớn nhất.

-16	7	-3	0	-1	5	-4

- Tổng của đoạn có trọng số lớn nhất trong mảng là 8
- Cách giải thế nào?
  - Phương pháp trực tiếp thử tất cả gần n^2 khoảng, và tính trọng số mỗi đoạn, cho độ phức tạp $O(n^3)$
  - Ta có thể xử lý kỹ thuật bởi một "mẹo" lưu trữ cố định trong vòng lặp để giảm độ phức tạp về $O(n^2)$
  - Liệu có thể làm tốt hơn với phương pháp Chia để trị?

Đoạn con có tổng lớn nhất

- Chia: Chia dãy n phần tử thành 2 dãy con tại điểm giữa mid = floor((n + 1)/2), ký hiệu là AL và AR
- Xử lý: Tính đoạn con có tổng lớn nhất của mỗi nửa một cách đệ qui. Gọi wL và wR là trọng số của các đoạn con có tổng lớn nhất trong AL và AR tương ứng.
- Kết hợp: ký hiệu trọng số của đoạn con lớn nhất mà nằm đè lên điểm chia ở giữa là wM. Kết quả cần tìm sẽ là max(wL, wR, wM)
  - wM được tính bằng tổng độ dài đoạn con có tổng lớn nhất nửa bên trái mà kết thúc tại mid và độ dài đoạn con có tổng lớn nhất nửa bên phải mà bắt đầu tại mid + 1

Đoạn con có tổng lớn nhất: Cài đặt

```
int SubSeqMax(int i, int j) {
    if (i == j) return a[i];
    int m = (i + j) / 2;
    int wL = SubSeqMax(i, m);
    int wR = SubSeqMax(m + 1, j);

    int wLM = MaxLeft(i, m);
    int wRM = MaxRight(m + 1, j);
    int wM = wLM + wRM;

    return max(max(wL, wR), wM);
}
```

- Gọi SubSeqMax(1, n);
- Độ phức tạp O(nlogn), giống bài toán sắp xếp trộn

Đoạn con có tổng lớn nhất: Cài đặt

```
int MaxLeftMid(int i, int m) {
    int ans = a[m], sum = 0;
    for (int k = m; k >= i; k--) {
        sum += a[k];
        ans = max(ans, sum);
    }
    return ans;
}

int MaxRightMid(int m, int j) {
    int ans = a[m], sum = 0;
    for (int k = m; k <= j; k++) {
        sum += a[k];
        ans = max(ans, sum);
    }
    return ans;
}
```

NỘI DUNG

- Chia để trị
- **Giảm để trị** chặt nhị phân
  - Tìm kiếm nhị phân
  - Tìm kiếm nhị phân trên các số nguyên
  - Tìm kiếm nhị phân trên các số thực
  - Tìm kiếm nhị phân câu trả lời
- Một số loại chia để trị thông dụng khác

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

Tìm kiếm nhị phân

```
bool Binary_Search(const vector<int> &A, int lo, int hi, int x) {
    if (lo > hi) return false;
    int mid = (lo + hi) / 2;
    if A[mid] == x return true;
    if (x < A[mid])
        return Binary_Search(A, lo, mid -1, x);
    if (x > A[mid])
        return Binary_Search(A, mid + 1, hi, x);
}
```

- Gọi `Binary_Search(A, 1,n, x);`
- Độ phức tạp:
  - 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, 1, ..., n-1} -> {TRUE, FALSE} thoả mãn nếu P(i) = TRUE thì P(j) = TRUE với mọi j > i
- Yêu cầu tìm chỉ số j nhỏ nhất sao cho P(j) = TRUE

i	0	1	$\cdots$	$j-1$	j	$j+1$	$\cdots$	$n-2$	$n-1$
$P(i)$	*FALSE*	*FALSE*	$\cdots$	*FALSE*	*TRUE*	*TRUE*	$\cdots$	*TRUE*	*TRUE*

- Có thể thực hiện trong O(log(n) x f), với f là giá là giá của việc đánh giá hàm P

Tìm kiếm nhị phân trên các số nguyên

```
int lo = 0, hi = n - 1;
while (lo < hi) {
    int mid = (lo + hi) / 2;
    if (P(mid)) {
        hi = mid;
    } else {
        lo = mid + 1;
    }
}

if (lo == hi && P(lo)) {
    cout << "The minimum index where " << x << " can be found is " << lo << endl;
} else {
    cout << "Cannot find " << x << endl;
}
```

Tìm kiếm nhị phân trên các số nguyên

- Tìm vị trí của x trong mảng đã sắp xếp A

```
bool P(int i) {
    return A[i] >= x;
}
```

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] -> {TRUE, FALSE} thoả mãn nếu P(i) = TRUE thì P(j) = TRUE với mọi j > i
- Yêu cầu tìm số thực nhỏ nhất j sao cho P(j) = TRUE
- 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((hi – lo) / EPS)) tương tự cách làm tìm kiếm nhị phân trên mảng

Tìm kiếm nhị phân trên các số thực

```
double EPS = 1e-10, lo = -1000.0, hi = 1000.0;

while (hi -lo >= EPS) {
    double mid = (hi + lo) / 2.0;
    if (P(mid)) {
        hi = mid;
    } else {
        lo = mid;
    }
}

cout << lo;
```

Tìm kiếm nhị phân trên các số thực

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

```
bool P(double j) {
    return j*j >= x;
}
```

- Tìm nghiệm của hàm F(x)

```
bool P(double x) {
    return F(x) >= 0.0;
}
```

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

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

NỘI DUNG

- Chia để trị
- Giảm để trị
- **Một số loại chia để trị thông dụng khác**
  - Nhị phân hàm mũ
  - Chuỗi Fibonacci

Một số loại chia để trị thông dụng 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 theo dõi 2 ví dụ sau đây

Nhị phân hàm mũ (Binary exponentiation)

- Yêu cầu tính x^n với x, n là các số nguyên
- Giả thiết ta không biết phương thức pow trong thư viện
- 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 (Binary exponentiation)

- 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 (Binary exponentiation)

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

```
int Pow(int x, int n) {
    if (n == 0) return 1;
    return x * Pow(x, n - 1);
}
```

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

Nhị phân hàm mũ (Binary exponentiation)

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

```
int Pow(int x, int n) {
    if (n == 0) return 1;
    if (n % 2 != 0) return x * pow(x, n - 1);
    int res = Pow(x, n/2);
    return res * res;
}
```

- 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) ➔ Thuật toán tối ưu

Nhị phân hàm mũ

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

Số Fibonacci

- Nhắc lại dãy Fibonacci được định nghĩa như sau:

$$\begin{aligned}\mathrm{Fib}_1 &= 1\\ \mathrm{Fib}_2 &= 1\\ \mathrm{Fib}_n &= \mathrm{Fib}_{n-2} + \mathrm{Fib}_{n-1}\end{aligned}$$

- Ta có dãy 1, 1, 2, 3, 5, 8, 13, 21...
- Có rất nhiều biến thể của dãy Fibonacci
- Một kiểu là cùng công thức nhưng bắt đầu bởi các số khác, ví dụ:

$$\begin{aligned}F_1 &= 5\\ F_2 &= 4\\ F_n &= F_{n-2} + F_{n-1}\end{aligned}$$

- Ta có dãy 5, 4, 9, 13, 22, 35, 57 ...
- 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?

$$\text{len}(G_1) = 1$$
$$\text{len}(G_2) = 1$$
$$\text{len}(G_n) = \text{len}(G_{n-2}) + \text{len}(G_{n-1})$$

- $\text{len}(G_n) = \text{Fib}_n$
- Vì vậy các xâu trở nên rất lớn rất nhanh

$$\text{len}(G_{10}) = 55$$
$$\text{len}(G_{100}) = 354224848179261915075$$
$$\text{len}(G_{1000}) =$$
$$43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209689623239873322471161642996409065331879382989696499285160037044761377951668492$$
$$28875$$

Số Fibonacci

- Nhiệm vụ: Hãy tính ký tự thứ i trong G_n
- Dễ dàng thực hiện trong O(len(G_n) nhưng sẽ cực kỳ chậm nếu n lớn
- Có thể giải trong O(n) sử dụng chia để trị

HUST

hust.edu.vn fb.com/dhbkhn

THANK YOU !