PC4 - Thuật toán và độ phức tạp

Ri Duan

Tháng 2/2018

1 Các phép toán trên $\mathbf{Z}/N\mathbf{Z}$

Giả sử n là một số nguyên dương. Mỗi phần tử của $\mathbf{Z}/N\mathbf{Z}$ được biểu diễn bởi một số nguyên trong khoảng từ 0 đến $N-1$. Các phép toán cộng, trừ, nhân, luỹ thừa hai phần tử a và b được thực hiện như với các số nguyên, nhưng lấy số dư khi chia kết quả cho N. Kí hiệu $n = \log_2 N$.

Chứng minh rằng:

1. *Phép cộng, trừ hai phần tử có thể được thực hiện trong thời gian $O(n)$.*
2. *Phép nhân hai phần tử có thể được thực hiện trong $O(n^2)$.*
3. *Phép luỹ thừa a^b có thể được thực hiện trong $O(n^3)$.*

Nghịch đảo của một phần tử. Một phần tử a được gọi là nguyên tố cùng nhau với N nếu với tư cách số nguyên, ước chung lớn nhất của a và N là 1. Trong trường hợp đó, tồn tại một phần tử $b \in \mathbf{Z}/N\mathbf{Z}$ sao cho $ab = 1$ trong $\mathbf{Z} \in N\mathbf{Z}$. Ta gọi nó là **nghịch đảo** của a, kí hiệu a^{-1} (dễ thấy tồn tại duy nhất trong $\mathbf{Z}/n\mathbf{Z}$).

Xét thuật toán **gcd** dưới đây:

- Input: $x, y \in \mathbf{N}, x \geq y$
- Output: d, a, b sao cho $d = \gcd(x, y)$, $ax + by = d$.

Thuật toán gcd(x, y)

- Nếu $y = 0$, trả lại $(1, 0, x)$
- Nếu không, $(a', b', d) =$ **gcd** $(y, x \mod y)$
- Trả lại $(b', a' - [x/y]b', d)$

Từ thuật toán trên, chứng minh

4. *Việc tìm nghịch đảo của một phần tử có thể được thực hiện trong $O(n^3)$. Do đó, việc thực hiện phép chia $a/b = a \cdot (b^{-1})$ trong trường hợp $(b, N) = 1$ có thể được thực hiện trong $O(n^3)$.*

2 Số nguyên tố và áp dụng trong mật mã

2.1 Thuật toán "ngây thơ"kiểm tra tính nguyên tố

Giả sử ta kiểm tra một số $x \leq N$ bất kì có nguyên tố hay không bằng cách chia x cho các số tự nhiên nhỏ hơn nó (đến một độ lớn nào đó). Chứng minh thuật toán này có thể được thực hiện trong $O(n^{1.59}\sqrt{N})$, với $n = \log_2 N$.

2.2 Kiểm tra tính nguyên tố bằng định lí nhỏ Fermat

Ta biết rằng nếu p là một số nguyên tố thì $a^{p-1} - 1 \equiv 0 \pmod{p}$ với mọi số nguyên a (định lí Fermat nhỏ).

Nếu p là hợp số thì điều này chỉ đúng với rất rất ít a (các số đặc biệt này được gọi là Carmichael, tồn tại với xác suất rất nhỏ). Do đó khi chọn a ngẫu nhiên, hầu hết sẽ không tuân thủ điều kiện Fermat nhỏ.

Từ quan sát đó, thuật toán sau được áp dụng trong thực tế để kiểm tra tính nguyên tố của một số tự nhiên x không vượt quá N:

- Lấy k số tự nhiên bất kì $a_1, \ldots, a_k$ nhỏ hơn x.
- Nếu $a_i^{x-1} \equiv 1 \pmod{x}$ với tất cả $i = 1, \ldots, k$, trả lời "nguyên tố". Nếu không, trả lời "hợp số".

Chứng minh rằng độ phức tạp của thuật toán trên là $O(n^3 \cdot k)$ với $n = \log_2 N$.

Trên thực tế, chỉ cần cố định k tương đối nhỏ (khoảng 10), độ phức tạp do đó là $O(n^3)$.

2.3 Phát sinh một số nguyên tố ngẫu nhiên

Định lí về phân bố số nguyên tố nói rằng nếu $\pi(N)$ là số số nguyên tố không vượt quá N thì

$$\lim_{N \to \infty} \frac{\pi(N)}{N/\ln N} = 1$$

Hay $\pi(N)$ sẽ xấp xỉ $x/\ln(N)$. Từ đó chứng minh được nếu lấy ngẫu nhiên một số tự nhiên không vượt quá N thì kì vọng số lần thử để số được lấy là nguyên tố là $O(\ln N)$.

Từ quan sát đó, trên thực tế người ta dùng thuật toán *thử và sai* sau đây để phát sinh (generate) một số nguyên tố không vượt quá N.:

- Lấy một số tự nhiên bất kì không vượt quá N. (Giả sử bước này được thực hiện trong $O(1)$)
- Dùng thuật toán ở phần 2.2 (với k cố định bằng $O(1)$) để kiểm tra nó có nguyên tố hay không.
- Trả lại số đã chọn nếu nó nguyên tố. Nếu không, chọn lại một số tự nhiên bất kì và lặp lại.

Chứng minh cả quá trình phát sinh số nguyên tố nói trên có thể được thực hiện trong thời gian trung bình $O(n^4)$.

2.4 Lí thuyết mật mã

Giả sử Alice muốn gửi một văn bản m cho Bob. Văn bản có thể lưu dưới dạng một số nguyên $m \leq N$. Văn bản là bí mật nên Alice và Bob không muốn một ai khác biết. Do đó Alice cần mã hoá văn bản m bằng cách tính $f(m)$ với f là một hàm nào đó. Alice và Bob thống nhất làm như sau:

- Bước 1. Bob, một cách bí mật, chọn hai số nguyên tố $p, q \leq \sqrt{N}$ ngẫu nhiên có dạng $3k + 2$, sau đó tính $S = pq$.
- Bước 2. Bob chọn $e = 3$, tính nghịch đảo d của e theo modulo $(p-1)(q-1)$. $e = 3$ được gọi là khoá công khai (public key), d được gọi là khoá bí mật.
- Bước 3. Bob gửi e và S cho Alice.
- Bước 4. Alice nhận được e và S. Alice tính $f(m) = m^e \pmod{S}$ và gửi $f(m)$ cho Bob.
- Bước 5. Bob tính $g(m) = f(m)^d \pmod{S}$, đó chính là văn bản ban đầu.

1. *Chứng minh $g(m) = m$.*

2. *Giả sử tỉ lệ số số nguyên tố có dạng $3k+2$ trên tổng số các số nguyên tố không vượt quá N lớn hơn 0.25 với N bất kì (đây là giả thuyết dựa trên quan sát trên thực tế). Chứng minh bước 1 có thể được thực hiện trong thời gian trung bình $O(n^4)$, $n = \log_2 N$.*

3. *Chứng minh mỗi bước 2, 4, 5 có thể được tính trong thời gian trung bình $O(n^3)$. Từ đó, giả sử bước 3 có thể thực hiện trong thời gian $O(1)$, chứng minh cả quá trình từ bước 1 đến 5 có thể được thực hiện trong thời gian $O(n^4)$. (Trên thực tế, nếu chọn $N = 2^{64}$ hay $N = 2^{128}$ với máy tính thông thường, việc này không tốn quá 1s.)*

4. *Giả sử trên đường truyền (bước 3, 4), một gián điệp Charlie lấy được cả e, S, $f(m)$ đồng thời biết được tất cả thoả thuận về cách thực hiện của Bob và Alice, nhưng không biết được p, q. Chứng minh nếu việc phân tích một số thành thừa số nguyên tố được thực hiện trong thời gian ngắn (ví dụ <1h) thì việc Charlie giải mã được m từ $f(m)$ cũng được thực hiện trong thời gian như vậy cộng với thời gian thực hiện bước 2, 5.*

5. *Giả sử không có thuật toán nào thực hiện hiệu quả việc phân tích S thành thừa số nguyên tố và Bob phải phân tích S bằng phương pháp thử với p từ nhỏ đến lớn xem S có chia hết cho p không, đến khi tìm được p mới tính d. Chứng minh nếu $N = 2^{128}$, mỗi phép chia S cho một số nhỏ hơn nó tốn ít nhất $10^{-8}s$, còn $p, q > \frac{\sqrt{N}}{1024}$ (tình huống rất dễ xảy ra, với xác suất >0.99), thì thậm chí mất hằng trăm năm việc giải mã của Charlie cũng không thể thực hiện được.*

Quy trình mã hoá thông qua khoá e và giải mã thông qua khoá d nêu trên được gọi là RSA (Rivest–Shamir–Adleman). Nó thuộc loại mật mã phi đối xứng vì mã hoá bằng một khoá e và giải mã bằng một khoá d khác, và việc tính d từ e không dễ dàng.

3 Các thuật toán trên array

1. *Cho một list $A = [a_1, \ldots, a_N]$. Hãy mô tả thuật toán đảo ngược thứ tự các phần tử của A, tức là biến A thành $B = [a_N, \ldots, a_1]$ trong thời gian $O(N)$.*

2. *Cho một list $A = [a_1, \ldots, a_N]$. Hãy mô tả một thuật toán tìm ra đồng thời cả giá trị lớn nhất và nhỏ nhất của các phần tử trong A, mà không dùng quá $3N/2$ phép so sánh giữa hai phần tử của A.*

3. *(*) Mô tả một thuật toán tìm k phần tử lớn nhất trong một list A có N phần tử trong thời gian $O(N \log k)$. (Bạn cần đọc phần 6, Sorting của bài giảng để làm bài này)*

4. *Cho hai list A, B đều chứa N phần tử. Mô tả một thuật toán kết luận xem có phần tử a nào thuộc cả hai list không. (Thuật toán tốt cần có độ phức tạp $O(N \log N)$)*

5. *Cho ba list A, B, C đều chứa N phần tử. Mô tả một thuật toán kết luận xem có phần tử a nào thuộc cả ba list không. (Thuật toán tốt cần có độ phức tạp $O(N \log N)$)*

6. *Cho ba list A, B, C đều chứa N số nguyên, k là một số nguyên. Mô tả một thuật toán kết luận xem có ba phần tử a, b, c nào lần lượt thuộc các list trên mà tổng của chúng bằng k. (Thuật toán tốt cần có độ phức tạp $O(N^2)$)*

4 Dynamic Array - Mảng động

4.1 Cơ chế 1

Giả sử một mảng động được tạo như sau: ban đầu có 1 ô nhớ và không lưu giá trị nào. Số ô nhớ của mảng động được gọi là size của mảng động đó. Mỗi khi mọi ô nhớ đều đã được dùng và một phép **append** được gọi, thì chương trình sẽ tăng số ô nhớ lên gấp đôi (tức là tạo ra N ô nhớ mới nếu

số ô nhớ hiện tại đang là N), và sự thêm này có thời gian chạy $\geq c_1 N$, $\leq c_2 N$ (c_1, c_2 là các hằng số). Ngược lại nếu mảng đang dùng ít hơn hoặc bằng một nửa số ô nhớ (tức số ô nhớ được dùng $\leq N/2$) mà một phép **pop** được gọi, thì chương trình sẽ xoá đi một nửa số ô nhớ trống bằng một quy trình có thời gian chạy $\geq c_1 N$, $\leq c_2 N$. Ngoài ra, nếu không phải các trường hợp này, thì các phép **append** và **pop** có thời gian chạy $\geq c_1$, $\leq c_2$.

1. *Chứng minh rằng nếu ta gọi liên tiếp nhiều phép toán, mỗi phép là **append** và **pop** theo thứ tự ngẫu nhiên bất kì, (ví dụ **append**, **append**, **pop**, **pop**, **append**, ...) thì size của mảng luôn là một luỹ thừa của 2.*

2. *Chỉ ra một ví dụ về một dãy các phép **append** và **pop** theo thứ tự bạn chọn, mà sau khi thực hiện dãy phép toán này, thời gian thực hiện trung bình một phép **append** hay **pop** không phải là $O(1)$.*

4.2 Cơ chế 2

Cũng như cơ chế 1, giả sử một mảng động được tạo như sau: ban đầu có 1 ô nhớ và không lưu giá trị nào. Số ô nhớ của mảng động được gọi là size của mảng động đó. Nhưng bây giờ, mỗi khi mọi ô nhớ đều đã được dùng và một phép **append** được gọi, thì chương trình sẽ tăng thêm $\lceil N/4 \rceil$ ô nhớ cho mảng động, và việc thêm này có thời gian chạy $\geq c_1 N$, $\leq c_2 N$ (c_1, c_2 là các hằng số). Ngược lại nếu mảng đang dùng ít hơn hoặc bằng một phần tư số ô nhớ (tức số ô nhớ được dùng $\leq N/4$) mà một phép **pop** được gọi, thì chương trình sẽ xoá đi một nửa số ô nhớ trống bằng một quy trình có thời gian chạy $\geq c_1 N$, $\leq c_2 N$. Ngoài ra, nếu không phải các trường hợp này, thì các phép **append** và **pop** có thời gian chạy là $\geq c_1$, $\leq c_2$.

*Chứng minh rằng nếu ta gọi liên tiếp nhiều phép toán, mỗi phép là **append** và **pop** theo thứ tự ngẫu nhiên bất kì, (ví dụ **append**, **append**, **pop**, **pop**, **append**, ...) thì thời gian thực hiện trung bình một phép **append** hay **pop** là $O(1)$.*

5 (*) Thiết kế kiểu dữ liệu: Sorted Map

(Chỉ làm bài tập này khi bạn đã đọc phần 6, Sorting của bài giảng).

Trong bài giảng ta đã biết kiểu dữ liệu Hash Map được xây dựng để lưu các bộ từ khoá - giá trị (k, V). Chúng được sử dụng trong các trường hợp ta thường xuyên làm việc với các phép toán tìm $d[k]$, gán $d[k] = V$, tìm tất cả các khoá $d.keys()$.

Mục tiêu của bài này là xây dựng một kiểu dữ liệu khác Hash Map để làm việc với các tình huống khác.

Giả sử ta có dữ liệu gồm các bộ (k, V) và cần làm việc với các phép toán:

- Cho k_m, phép toán $getLowerBound(k_m)$ tìm giá trị $data[k]$ ứng với khoá k nhỏ nhất thoả mãn $k \geq k_m$
- Cho k_M, phép toán $getUpperBound(k_M)$ tìm giá trị $data[k]$ ứng với khoá k lớn nhất thoả mãn $k \leq k_M$.
- Cho k_m, k_M, phép toán $getRange(k_m, k_M)$ tìm giá trị $data[k]$ ứng với tất cả các khoá k thoả mãn $k_m \leq k \leq k_M$.
- Cho k, V, phép toán $put(k, V)$ cho phép thêm (k, V) vào dữ liệu nếu từ khoá k chưa có trong dữ liệu.

1. *Xét bài toán sau đây: cho dữ liệu của một hãng hàng không gồm N bộ dạng (k, V) trong đó k là một xâu dạng ngày-tháng-năm-giờ-phút, YYYYMMDDmmHH (ví dụ 201801311513 chỉ 13h15 ngày 31/01/2018 giờ GMT), và V là tên một chuyến bay khởi hành tại thời điểm đó đi*

kèm với số hành khách đã đặt vé trên chuyến bay đó, ví dụ (VN311, 89). Giả sử ta muốn tìm tổng số lượt đặt vé trên tất cả các chuyến bay trong một chuỗi ngày liên tiếp bất kì. Thông qua hàm nào trên đây ta có thể giải quyết bài toán?

2. *Nếu lưu dữ liệu về chuyến bay trên bằng Hash Map với từ khoá k và giá trị V, chứng minh việc nhập dữ liệu gồm N bộ trên có thể được thực hiện trong thời gian $O(N)$, và việc giải bài toán ở câu 1 (tính tổng sõ lượt đặt vé giữa hai ngày bất kì) có thể được giải quyết trong thời gian $O(N)$.*

Trên thực tế, $O(N)$ là độ phức tạp trung bình tốt nhất để thực hiện các hàm $getLowerBound(k_m)$, $getUpperBound(k_M)$, $getRange(k_m, k_M)$, $put(k, V)$ nói trên. Độ phức tạp này vẫn khá lớn và ta cần tìm cách lưu dữ liệu tốt hơn.

Bây giờ ta muốn xây dựng một kiểu dữ liệu Sorted Map lưu N bộ (k, V) như sau:

- Với mỗi bộ (k, V), lưu chúng vào một địa chỉ nhớ.
- Dùng một mảng động (dynamic array) để lưu N địa chỉ nhớ đó theo quy tắc k tăng dần. Mỗi lần thêm một phần tử vào mảng động, cần tuân thủ thứ tự của k.

Sử dụng Sorted Map, chứng minh:

3. *Việc nhập dữ liệu gồm N bộ trên có thể được thực hiện trong thời gian $O(N \log N)$.*
4. *$getLowerBound(k_m)$, $getUpperBound(k_M)$ có thể được thực hiện trong $O(\log N)$.*
5. *Việc cập nhật $put(k, V)$ khi k đã có trong dữ liệu có thể được thực hiện trong $O(\log N)$. Nếu chưa tồn tại, trường hợp worst case của $put(k, V)$ được thực hiện trong $O(N)$.*
6. *$getRange(k_m, k_M)$ được thực hiện trong $O(\log N + S)$, trong đó S là số cặp (k, V) trong kết quả (tức $k_m \leq k \leq k_M$). Đây cũng là độ phức tạp của bài toán tính tổng lượt vé được đặt nêu trên.*