Vietnam Northern Provincial 2018 Editorial

Overview

Bài	First AC	#AC	Writer
A	71 FPTU Reserved 2	8	Nguyen Diep Xuan Quang HCMUS
B	3 Send Bob to Alice	244	Nguyễn Thị Việt Hà Illumina
C	49 Send Bob to Alice	58	Hoàng Dương Illumina
D	38 UIT.BakaCF	205	Lê Minh Phúc Illumina
E	166 Pandamiao	13	Nguyễn Tiến Trung Kiên Pandamiao
F		0	Trần Tấn Phát Send Bob to Alice
G	35 HCMUS-TheCows	4	Phạm Bá Thái P_not_equal_NP
H	123 Ams-Fusion	5	Trần Tấn Phát Send Bob to Alice
I	118 bitset	5	Phạm Thắng bitset
J	147 Send Bob to Alice	5	Lê Quang Tuấn Send Bob to Alice
K	9 Amazingbamboo_with_coccoc & map	187	Lê Minh Phúc Illumina
L	259 Send Bob to Alice	1	Nguyễn Đinh Quang Minh Send Bob to Alice

A

Lời giải: Nguyễn Diệp Xuân Quang

Trước hết, ta xét một tập con S của N người ban đầu. Gọi K = |S|. Giả sử có cách để cho mọi người trong tập S rời khỏi hố. Ta cần tìm thứ tự rời hồ tối ưu dành cho những người trong tập S (tức là sắp xếp K người lại theo một thứ tự mới, sao cho người có thứ tự nhỏ hơn sẽ rời hố trước người có thứ tự lớn hơn).

Ta tưởng tượng quá trình ngược lại: thay vì lần lượt K người rời hố, giả sử ban đầu, trong hố gồm N - K người không thuộc tập S. Ta cần tìm một thứ tự để thả K người này rơi xuống hố, sao cho tại thời điểm người thứ i rơi xuống hố thì người này có khả năng rời khỏi hố ngay lúc đó (tức là $H_i + L_i$ + tổng H của những người trong hố lớn hơn hoặc bằng d).

Khi đó, ta có thể thấy rằng, ở mọi thời điểm, ta nên thả người có $H_i + L_i$ lớn nhất rơi xuống hố, vì người này có khả năng rời hố cao nhất trong thời điểm này. Sau khi ta thả một người nào đó vào hố, tổng H của những người trong hố sẽ chỉ tăng lên chứ không giảm đi. Do đó, khả năng rời hố sau khi bị thả vào của những người còn lại sẽ không trở nên tồi đi.

Nói cách khác, ta sắp xếp N người ban đầu theo thứ tự $H_i + L_i$ tăng dần. Khi đó, với hai chỉ số i, j sao cho i < j thì người thứ i nên rời hố trước người thứ j.

Đến đây, ta đã có thể dùng quy hoạch động để giải quyết bài toán. Gọi dp[i][k] là tổng chiều cao lớn nhất của i - k người còn lại trong hố trong số, khi đã có k người ra khỏi hố (chỉ tính trong i người đầu tiên). Khi đó, với một trạng thái (i, k), ta cần cập nhật theo hai trường hợp:

- Người thứ i + 1 rời hố. Trường hợp này chỉ xảy ra nếu H[i + 1] + L[i + 1] + dp[i][k] >= d. Khi đó thì dp[i+1][k+1] = max(dp[i+1][k+1], dp[i][k]).
- Người thứ i + 1 ở lại hố. Khi đó thì dp[i+1][k] = max(dp[i+1][k], dp[i][k] + H[i]).

Độ phức tạp: $O(N^2)$

B

Lời giải: Nguyễn Thị Việt Hà

Sau khi chạm vào đồ vật có giá trị v, tất cả các món đồ có giá trị v trong dãy sẽ bị xóa khỏi dãy, không được chọn vào kết quả. Với test ví dụ, cách chọn tối ưu là bỏ món đồ thứ 3 ($w_3 = 2$) và chọn các món đồ còn lại có tổng giá trị bằng $1 + 6 + 1 + 1 = 9$

Thao tác này chỉ thực hiện duy nhất 1 lần, do đó bài toán quy về, tìm

$$max\ \{\ \sum_{i=1}^{n} w_i\ -\ w_j\ *\ occurrences[w_j]\}$$

với

$$occurrences[w_j]\ =$$ số lần xuất hiện của giá trị w_j trong dãy

Có thể dùng hashtable để tính và truy vấn nhanh số lần xuất hiện của một giá trị (Độ phức tạp: O(1) cho mỗi cập nhật/truy vấn).

Độ phức tạp: O(n)

C

Lời giải: Hoàng Dương

Để giảm thiểu sai sót và tiết kiệm thời gian code, ta có thể chỉ viết **2 hàm cơ bản nhất** và dùng 2 hàm này để thực hiện 4 thao tác, thay vì việc viết 4 hàm riêng biệt cho 4 thao tác.

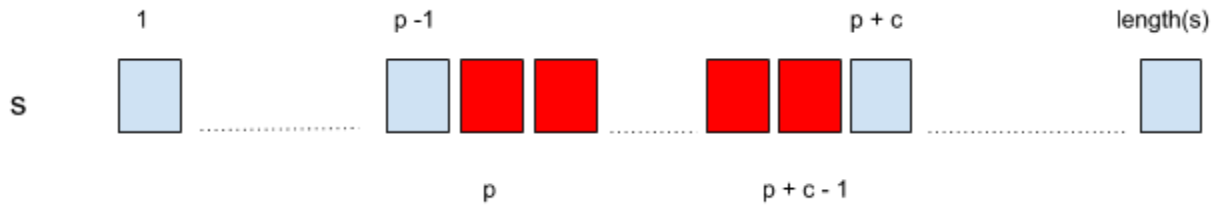
2 hàm cơ bản là:

- **`concatenate(string s, string t)`**: Ghép xâu t vào sau xâu s (**thao tác 1** trong đề bài) (Chi tiết cài đặt phía dưới, độ phức tạp O(L) với L là độ dài xâu rút gọn s, t)

- **`getSubstring(string s, int l, int r)`**: Lấy xâu con từ vị trí **l** tới vị trí **r** của xâu **s** (**s[l...r]**) (**thao tác 3** trong đề bài có thể chuyển về thao tác này bằng cách gán **l = p**, **r = p + c - 1**) (Chi tiết cài đặt phía dưới, độ phức tạp O(L) với L là độ dài xâu rút gọn s)

Để thực hiện thao tác 2 và thao tác 4 với 2 hàm cơ bản trên, ta thực hiện như sau:

- **Với thao tác 2:** Xóa c kí tự, từ vị trí p, trong xâu s. Thao tác này tương đương với việc xóa xâu con từ vị trí p tới vị trí p + c - 1 của xâu s (s[p...p + c - 1]).



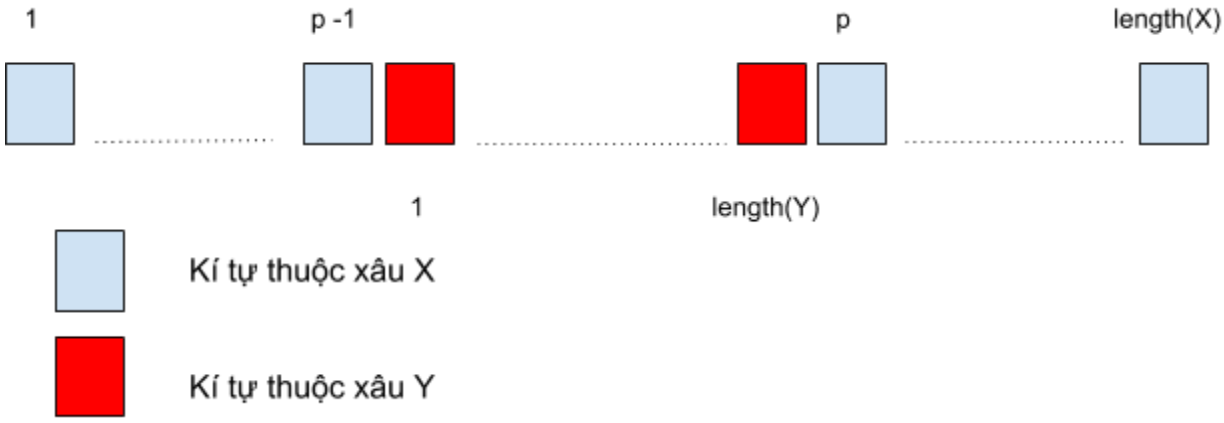
Thay vì việc xóa trực tiếp, ta có thể lấy xâu con của s từ vị trí 1 tới vị trí p - 1 (s[1...p - 1]) ghép với xâu con từ vị trí p + c tới vị trí length(s) (s[p + c...length(s)]) (length(s) = độ dài xâu s).

Ta có đoạn code sau để lấy kết quả thao tác 2:

```
prefix = getSubstring(s, 1, p - 1);
suffix = getSubstring(s, p + c, length(s));
return concatenate(prefix, suffix);
```

Lưu ý: Kết quả có thể là xâu rỗng!

Với thao tác 4: Chèn xâu Y vào vị trí p của xâu X. Giống ý tưởng trên với thao tác 2, ta có thể lấy xâu con từ vị trí 1 tới vị trí p - 1 của xâu X (X[1...p - 1]), xâu Y và xâu con từ vị trí p tới vị trí (length(X)) (X[p...length(X)]) (length(X) = độ dài xâu X). Sau đó nối 3 xâu theo thứ tự trên để có kết quả.



Ta có đoạn code sau để lấy kết quả thao tác 4:

```
prefix = getSubstring(X, 1, p - 1);
suffix = getSubstring(X, p, length(s));
return concatenate(concatenate(prefix, Y), suffix);
```

Độ phức tạp: O(Q * L) với Q = số truy vấn, L = độ dài xâu rút gọn

---

Chi tiết cài đặt với 2 hàm cơ bản:

- **`concatenate(string s, string t)`**:
  Để cài đặt, ghép nhóm kí tự cuối của xâu s với nhóm kí tự đầu của xâu t nếu chúng giống nhau, các nhóm kí tự khác không thay đổi. Ví dụ trong đề bài, để ghép **X = 'a1b10**' và **Y = 'b3c9'**, ta giữ nguyên **'a1'**, ghép **'b10'** và **'b3'** thành **'b13'** và giữ nguyên **'c9'**, để có kết quả là **Z = 'a1b13c9'**.

  Độ phức tạp của hàm này là **O(L)** với L là độ dài xâu rút gọn (L <= 1000).
  Lưu ý: Xử lý trường hợp **s** hoặc/và **t** rỗng (xảy ra khi dùng hàm trên cho các thao tác khác).

- **`getSubstring(string s, int l, int r)`**:
  Để cài đặt, cần xác định vị trí **l** và **r** thuộc nhóm kí tự nào trong xâu. Ví dụ, với xâu **s = 'a10b10c20'**, **l = 2**, **r = 21**, l thuộc nhóm kí tự **'a10'**, r thuộc nhóm kí tự **'c20'**.
  - Nếu **l** và **r** thuộc cùng 1 nhóm kí tự, tính số kí tự trong đoạn **[l, r]** và trả kết quả.
  - Nếu **l** và **r** không thuộc cùng 1 nhóm kí tự, tính số kí tự trong đoạn **[l, r]** cùng nhóm kí tự với **l** và **r**, và lấy tất cả các nhóm kí tự nằm trọn trong đoạn **[l, r]**. Với ví dụ trên, có 9 kí tự **'a'** trong đoạn **[2, 21]** thuộc cùng nhóm với **l**, nhóm kí tự **'b10'** nằm trọn trong đoạn **[2, 21]**, 1 kí tự **'c'** trong đoạn **[2, 21]** thuộc cùng nhóm với **r**. Do đó, kết quả là **'a9b10c1'**.

  Độ phức tạp của hàm này là **O(L)** với L là độ dài xâu rút gọn (L <= 1000).
  Lưu ý: Xử lý trường hợp **s** rỗng hoặc/và kết quả rỗng (**l > r**) (xảy ra khi dùng hàm trên cho các thao tác khác).

D

Lời giải: Lê Minh Phúc

Vì BTC đã cho kết quả số tam giác trong hình khi bỏ đi 1 cạnh AB, ta đếm số tam giác khi bỏ đi 0, 2 và 3 cạnh như sau:

- 0 cạnh: lấy (số tam giác khi bỏ cạnh AB) + (số tam giác có cạnh nằm trên AB).
- 2 cạnh: Đếm số tam giác có cạnh nằm trên AC sau khi đã bỏ AB, rồi lấy 35 trừ đi.
- 3 cạnh: Đếm số tam giác có cạnh nằm trên BC sau khi đã bỏ AB và AC, rồi lấy kết quả phần 2 cạnh trừ đi.

E

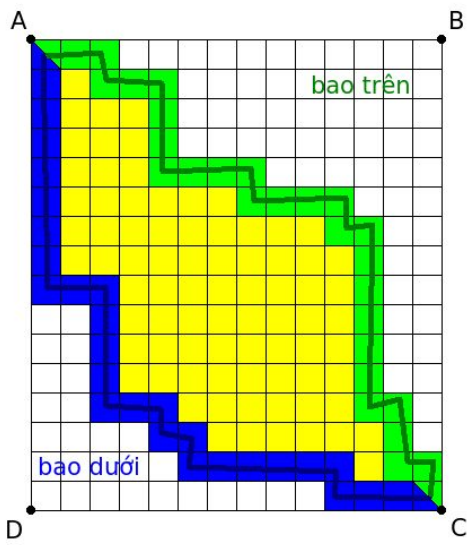
Lời giải: Nguyễn Tiến Trung Kiên

Định nghĩa

Đường đi hợp lệ là một đường đi từ ô A (1, 1) đến ô C (m, n) mà chỉ sử dụng các bước đi sang phải hoặc đi xuống dưới.
Đường đi tối ưu là một đường đi hợp lệ có tổng lớn nhất.

Nhận xét

1. Trong tất cả các đường đi tối ưu từ A đến C, có một đường đi gọi là bao trên P và một đường đi gọi là bao dưới Q:
    - Bao trên P: đường đi tối ưu "gần B nhất".
    - Bao dưới Q: đường đi tối ưu "gần D nhất".
2. Mọi đường đi tối ưu đều nằm trong khoảng được giới hạn bởi bao trên P và bao dưới Q.
3. Một ma trận con R chặn được đường đi S nếu như giữa S và R có ít nhất một điểm chung.
4. Hai mệnh đề sau là tương đương:
    - Ma trận con R chặn được bao trên P và bao dưới Q.
    - Ma trận con R chặn được tất cả các đường đi tối ưu.



Thuật toán

1. Tìm bao trên P và bao dưới Q.
2. Thử tất cả các ma trận con R có một đỉnh nằm trên bao trên P và đỉnh đối diện nằm trên bao dưới Q. Tìm ma trận R có diện tích bé nhất mà R không chặn tất cả các đường đi từ A đến C.

Độ phức tạp

Bước 1: O(n^2)
Bước 2: O(n^2)
Tổng cộng: O(n^2)

F

Lời giải : Trần Tấn Phát
Nhận xét : Chỉ có k + 1 số điểm khác nhau người chơi x có thể đạt được.

Giả sử người chơi x thắng e trận, gọi f[i][j][l] = số trường hợp có thể để người chơi i thắng j trận , có l người trong bảng hiện tại của i hơn điểm người chơi x. Với mỗi j , cần xử lý cùng lúc 1 block 2^j người liên tiếp để tính được f[i][j][l] trong O(1).
Đpt : O((2 ^ k) * k * k) .
Cài đặt: http://acmicpc-vietnam.github.io/2018/northern/codes/f.cpp

G

Lời giải: Phạm Bá Thái
Đây có lẽ là một bài dễ về thuật toán, chắc hẳn BTC đang muốn kiểm tra khả năng tối ưu code của các team

Thuật toán 1:

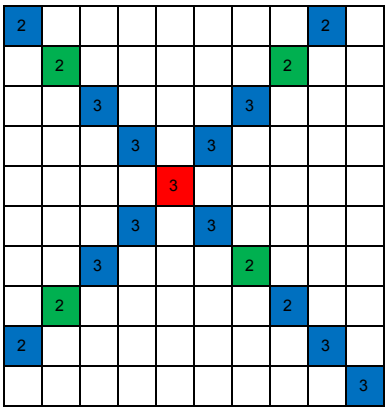
Dựng đồ thị với mỗi ô trên lưới là một đỉnh, và mỗi cách di chuyển hợp lệ là một cạnh. Ta tìm đường đi ngắn nhất bằng thuật toán BFS. Độ phức tạp của BFS là O(số cạnh). Tuy nhiên số cạnh của đồ thị này cỡ n^3 nên cách này không tốt

Thuật toán 2:

Dựng đồ thị với tập đỉnh (x,y,h) là tọa độ và hướng đang chạy của người đó trên lưới. Các đỉnh (x,y,h) được nối với (x',y',h) bởi một cạnh có trọng số bằng 0, và được nối với (x',y',h') bởi một cạnh có trọng số bằng 1. Ở đây x',y' là một ô có chung đúng một đỉnh với (x,y), và h' khác h. Đồ thị này có số đỉnh là n*m*2 (chỉ cần 2 hướng thay vì 4), số cạnh bằng cỡ 4 lần số đỉnh. Do chỉ có cạnh 0 1 nên có thể loang theo lớp để tìm đường đi ngắn nhất, độ phức tạp cũng là O(số cạnh)

Tối ưu thuật toán 1:

Thuật 2 có vẻ có độ phức tạp tốt hơn, thực tế team mình cũng cài thuật 2 trước vì biết chắc thuật 1 không tốt. Sau khi biết time quá chặt, team mình chuyển sang tối ưu thuật toán 1 bằng 1 trick: Khi for cách cạnh kề với ô (x,y), nếu gặp cây, đi quá biên, hoặc gặp một ô (x',y') đã thăm mà f(x',y') <= f(x,y) thì break luôn ở đây (f(x,y) là khoảng cách của ô (x,y) đến ô xuất phát). Thuật toán này đúng vì những ô bị bỏ qua bởi lệnh break sẽ được xét tiếp khi ta thăm đến ô (x',y'). Với trick này, mỗi ô sẽ không có quá 4 ô xét được đến nó, đó là 4 ô gần nhất về 4 hướng mà có f nhỏ hơn nó, các ô mà sẽ chuyển nhãn cho nó.



Như ở hình trên, tất cả các ô có ghi số đều là ô láng giềng của ô đỏ trên đồ thị mà chúng ta dựng. Tuy nhiên chỉ có các ô xanh lá, khi được thăm, sẽ for được đến tận ô đỏ mà không bị break, các ô xanh dương sẽ không for đến được. Như vậy nếu cộng tổng số lần được chuyển nhãn của các ô đỏ lại thì ta được đpt là 4*n*m. Với đpt tiệm cận như vậy, cần có thêm một vài trick như dùng luôn mảng f để đánh dấu, dùng mảng thay cho queue, ,fast I/O,… tùy vào khả năng tối ưu của các team

H

Lời giải : Trần Tấn Phát
Đầu tiên , preprocess các xâu input trên trie, lưu lại mỗi node có bao nhiêu xâu đi qua. Sau khi làm xong, với mỗi nút sẽ dựng được 1 cây BIT/ Segment tree để đếm số lượng xâu xuất hiện i lần có prefix tương ứng với nút hiện tại. Nhờ đó , ta đã giải quyết xong việc đếm số lượng xâu xuất hiện nhiều hơn xâu hiện tại đang xét.

Với các xâu xuất hiện ngang số lần có thứ tự từ điển nhỏ hơn, để ý rằng ta có thể xét theo độ dài prefix chung giảm dần .Ta sẽ đếm bằng cách đi xuống các nhánh tương ứng với chữ cái nhỏ hơn chữ cái tiếp theo rồi sử dụng ctdl như trên. Chú ý trường hợp một xâu là prefix của xâu hiện tại và xuất hiện nhiều lần hơn.

Độ phức tạp: O(len * 26 * log(len))
Cài đặt: http://acmicpc-vietnam.github.io/2018/northern/codes/h.cpp

I

Lời giải: Phạm Thắng

Ta dễ dàng có thuật toán độ phức tạp O(N * N) bằng việc sử dụng quy hoạch động f(i) là kết quả tối ưu nếu ta chỉ xét dãy w(1), w(2), …, w(i).

Nhận thấy f(i) = min(f(j) + max(j + 1 -> i) - min(j + 1 -> i)) (với 1 <= j <= i - 2).

Để tối ưu độ phức tạp, giả sử tại vị trí i, ta xét các vị trí j <= i - 2, ta có g(j) = f(j) + max(j + 1 -> i) - min(j + 1 -> i).

Vậy nếu ta muốn cập nhật mảng g cho vị trí i + 1 thì ta nên làm thế nào?

Nhận thấy từ vị trí i -> i + 1, ta thay đổi max và min của một đoạn liên tiếp, hay tồn tại 1 đoạn (j', …, i) mà max(j' + 1 -> i + 1) = a(i + 1), tương tự với min.

Xét trường hợp ta thay đổi giá trị max của 1 đoạn l, r nào đó thành a(i + 1) với max(l + 1 -> i) = … = max(r + 1 -> i). Khi đó mọi giá trị g(j) với l <= j <= r sẽ tăng thêm 1 lượng là a(i + 1) - max(l + 1 -> i).

Vậy lúc này, ta có thể nghĩ đến việc sử dụng một cây phân đoạn để tăng giá trị g trong 1 đoạn và sử dụng cây trên để tính hàm f.

Ta giải tương tự với trường hợp min, việc tìm đoạn l, r có thể dùng stack để lưu các khoảng.

Vậy độ phức tạp khoảng O(N * LOG2(N)).

J

Lời giải: Lê Quang Tuấn

Tóm tắt đề bài : Cho một dãy **A** gồm ***N*** phần tử. Thực hiện **Q** truy vấn, mỗi truy vấn có dạng ***(l ,r)***, yêu cầu thay thế đoạn con các phần tử liên tiếp từ ***l*** đến ***r*** trên dãy bằng min(**A[l..r]**) hoặc max(**A[l..r]**). Cuối cùng, hãy đếm số dãy kết quả khác nhau có thể tạo được.

Lời giải :

Nhận xét : Sau một số lần thực hiện truy vấn, một vị trí ***i*** có thể mang một số giá trị nào đó, và 2 vị trí ***i*** và ***j*** bất kì khác nhau thì tập giá trị có thể của ***i*** độc lập với tập giá trị có thể của ***j.*** Điều này đúng vì các truy vấn yêu cầu **thay thế** các phần tử trong đoạn **[l, r]** bằng một phần tử ***mới*** có giá trị bằng **max** hoặc **min** của các phần tử trong đoạn **[l, r],** cho nên tại một thời điểm nào đó thì hai phần tử khác nhau thì không liên quan gì đến nhau.

Thuật toán :

- Sau mỗi truy vấn, mỗi phần tử ***i*** lưu lại một **set** các giá trị mà nó có thể mang.
- Để thực hiện truy vấn **(l, r),** cần tìm **set** các giá trị mà phần tử mới này có thể mang.
- Gọi **S[i]** là set các giá trị mà phần tử thứ **i** có thể mang, **min(S[i])** là giá trị bé nhất thuộc set **S[i]**, **max(S[i])** là giá trị lớn nhất thuộc set **S[i].**
- Xét một giá trị **x** nào có thuộc một tập **S[i]** nào đó **(l <= i <= r).**
    - Nếu ở truy vấn này ta chọn phép thay thế cả đoạn bằng giá trị **x** sử dụng truy vấn **min**, thì các vị trí **j** khác trong đoạn, set giá trị có thể của nó phải chứa một phần tử có giá trị lớn hơn hoặc bằng **x** (vì khi đó **x** mới có thể là **min** của đoạn), Hay nói cách khác **x <= max(S[j])** với mọi **(l <= j <= r)** và **(j** khác **i)**. Nhưng hiển nhiên **x <= max(S[i])** nên điều kiện trên tương đương với **x <= max(S[j])** trong đó (**l <= j <= r**). (bỏ đi điều kiện **j** khác **i**).
    - Tương tự ta có điều kiện **x >= min(S[j])** với mọi **(l <= j <= r)**, nếu ở truy vấn này ta thực hiện phép lấy **max** và giá trị cuối cùng của đoạn bằng **x.**
- Bây giờ, ta chỉ cần gộp tất cả các set **S[i]** lại với nhau thành set các giá trị của phần tử mới, đồng thời xoá đi các giá trị không thoả mãn hai điều kiện trên.
- Để thực hiện việc gộp các set với nhau, ta cần tìm set có số phần tử lớn nhất, rồi **insert** tất cả các phần tử của các set khác vào set lớn nhất này. Độ phức tạp của việc gộp set này là tổng số lần một phần tử được chuyển từ set này sang set khác. Rõ ràng, một phần tử khi được chuyển từ set A sang set B, thì **|(B ∪ A)| >= 2 * |A|.** Mà các set thì có lực lượng lớn nhất có thể là **N**. Cho nên một phần tử chỉ được chuyển qua chuyển lại tối đa **log(N)** lần.
- Bước cuối cùng chỉ cần lấy tích của tất cả lực lượng các set còn lại trong dãy.
- Một chi tiết nhỏ là ta cần dùng một cây BIT/IT để lưu lại những phần tử nào chưa bị xoá cũng như là thực hiện việc xoá các phần tử, tìm xem phần tử nào đứng thứ l, phần tử nào đứng thứ r ở trong dãy hiện tại.

Tổng độ phức tạp. **O(N x log(N) x log(N))**. (N log (N) lần chuyển các phần tử giữa các set, mỗi lần chuyển mất O(log(N)).
Code mẫu: http://acmicpc-vietnam.github.io/2018/northern/codes/j.cpp

K

Lời giải: Lê Minh Phúc

Chú ý trên tử số và mẫu số chỉ có 1000 số, nên ta có thể tính:
P = tích các số trên tử
Q = tích các số trên mẫu
mà ko sợ TLE.
Sau đó tìm R = gcd(P,Q) với độ phức tạp O(log(max{P,Q})) xấp xỉ 50,000 vì P và Q max cũng chỉ 10^15,000 (tầm 2^50,000) và in ra (P/R)%M và (Q/R)%M.
Bài này giá trị số lớn, I/O không cần quá nhanh nên có thể dùng Java BigInteger vì có sẵn hàm gcd và modulo.

L

Lời giải: Nguyễn Đinh Quang Minh (team Send Bobs to Alice)

Giả sử các số A_1, A_2, …, A_K thỏa mãn $T = A_1A_2...A_K$ là một số chính phương. Ta thấy T là số chính phương khi và chỉ khi với mọi số nguyên tố p, số mũ của p trong phân tích thừa số nguyên tố của T phải là số chẵn. Mặt khác, số mũ của p trong phân tích của T chính bằng tổng số mũ của p trong các phân tích của A_1, A_2, …, A_K. Do vậy, nếu ta gọi B(i, p) là tính chẵn lẻ của số mũ của p trong phân tích thừa số nguyên tố của A_i, thì $B(1, p) \oplus B(2, p) \oplus \ldots \oplus B(k, p) = 0$ với mọi p, trong đó $\oplus$ là phép XOR, hay phép cộng trên modulo 2.

Quay lại với bài toán, nếu ta dựng ma trận B với ý nghĩa như trên, và p lên tới $4*10^6$, thì kết quả bài toán là số tập con các dòng thỏa mãn tổng của chúng bằng (0, 0, …, 0) (điều kiện (*)).

Ta có một số nhận xét sau:

+ Nếu ma trận B có 0 dòng, số tập con các dòng thỏa mãn điều kiện trên là 1 (tập rỗng).
+ Ngược lại, nếu ma trận B có N > 0 dòng, xét N-1 dòng đầu của ma trận B. Giả sử có S tập con của N-1 dòng đầu thỏa mãn (*). Khi đó:
    a) Nếu dòng thứ N độc lập tuyến tính với các dòng trên thì số tập con thỏa mãn điều kiện (*) vẫn là S.
    b) Ngược lại, số tập con thỏa mãn (*) sẽ là 2S.

Hiển nhiên nhận xét a đúng, vì không tồn tại tập con nào chứa dòng N thỏa mãn điều kiện (*) - nếu không, dòng N sẽ viết được dưới dạng tổng modulo 2 của các dòng còn lại trong tập con, do đó phụ thuộc tuyến tính.
Với nhận xét b, xét một biểu diễn bất kì của dòng N dưới dạng tổ hợp tuyến tính của các dòng khác, nó cho ta một tập D thỏa mãn (*) và có chứa dòng N. Khi đó, mọi tập con thỏa mãn (*) không chứa dòng N sẽ ánh xạ 1-1 với một tập con thỏa mãn (*) có chứa dòng N, bằng cách XOR tập con đó với D. Do đó, số tập con thỏa mãn (*) sẽ bằng 2 lần số tập con thỏa mãn (*) mà không chứa dòng N, tức là bằng 2S.

Tóm lại, số tập con các dòng thỏa mãn điều kiện (*) sẽ bằng $2^{N - rank(B)}$, với rank(B) là số dòng độc lập tuyến tính của B. Giá trị này có thể tính bằng thuật toán khử Gauss. Tuy nhiên, số dòng và số cột của ma trận B là lớn, do vậy không thể áp dụng khử Gauss thông thường. Đến đây, ta sử dụng một kết quả quen thuộc: với mỗi số không lớn hơn $4*10^6$, tồn tại không quá một thừa số nguyên tố lớn hơn $sqrt(4*10^6) = 2000$. Điều đó kéo theo việc với mỗi hàng của ma trận B, tồn tại không quá một ô bằng 1 ứng với số nguyên tố lớn hơn 2000. Như vậy, thuật toán khử Gauss có thể thực hiện như sau:

+ Với mỗi số nguyên tố p > 2000, xét tất cả các dòng i có B(i, p) = 1. Nếu không tồn tại thì bỏ qua p.

+ Thực hiện “khử” các hàng này. Phép tổ hợp tuyến tính trên modulo 2 thực chất chỉ là phép XOR 2 hàng với nhau, do đó sau khi khử, các hàng này đều không còn thừa số p > 2000 nữa.
+ Sau khi xét tất cả các số p > 2000, các hàng giờ chỉ còn các thừa số nhỏ hơn 2000. Số các số nguyên tố đến 2000 là M = 303, do đó ta có thể khử Gauss như bình thường với độ phức tạp $O(N * M^2)$.

Tất nhiên, $O(N * M^2)$ khá lớn với N = 200000, M = 303, nhưng các phép “cộng” các hàng chỉ là phép XOR, do đó ta có thể coi mỗi hàng là 5 số 64-bit, chi phí cho phép XOR sẽ giảm 64 lần. Với các ngôn ngữ hỗ trợ kiểu dữ liệu dạng bitset, ví dụ std::bitset<303> trong C++, việc cài đặt sẽ đơn giản hơn nhiều.

Code AC trong giờ của team Send Bobs to Alice:
http://acmicpc-vietnam.github.io/2018/northern/codes/I.cpp