8 Về RMQ và LCA

Lê Minh Khánh
THÀNH VIÊN DỰ TUYỂN 2020

"Rage, rage against the dying of the light"
- Dylan Thomas -

Lời nói đầu

Các bài toán liên quan đến RMQ và LCA đóng một vai trò quan trọng trong ngành khoa học máy tính. Đến nay thì công cuộc nghiên cứu các vấn đề liên quan đến RMQ và LCA đã được xem như là hoàn thành. Tưởng chừng hai vấn đề này liên quan đến 2 lĩnh vực khác nhau nhưng cuối cùng chúng lại có một mối quan hệ mật thiết với nhau không tưởng. Trong bài viết này thì chúng ta sẽ cùng tìm hiểu về mối quan hệ giữa RMQ và LCA cũng như các ứng dụng của nó vào giải bài dựa trên bài viết của Guo Huayang – một IOIer thứ thiệt qua một bài tập nho nhỏ những không kém phần thú vị.

Một số kiến thức sẽ được đề cập

1. LCA – Lowest Common Ancestor : tổ tiên chung gần nhất
2. RMQ – Range Minimum/Maximum Query : tìm phần tử nhỏ nhất / lớn nhất trong 1 khoảng [L,R] của mảng.
3. Thuật toán Tarjan
4. Sparse Table / Segment Tree

Đầu tiên chúng ta cùng đến với bài toán của chúng ta và cùng nhau bàn luận các vấn đề xung quanh nó :

1. Problem: Director of water management (Winter Camp 2006)

a. Statement

> Công ty cấp nước cần chuyển nước từ điểm U đến điểm V mỗi ngày. Tao cần tìm một tuyến đường ống nước từ U -> V để vận chuyển nước. Tao chỉ có thể thực hiện một công việc cấp nước tại một thời điểm, và chỉ có thể thực hiện công việc tiếp theo sau khi công việc trước đó đã hoàn thành.Trước khi có thể thực hiện các công, các đường ống cần được bảo trì trước. Chúng bắt đầu bảo trì cùng 1 thời điểm nhưng vì mỗi ống lại có kích thước khác nhau dẫn đến thời gian mỗi ống bảo trì cũng khác nhau do đó

thời gian bảo trì của một tuyến đường ống nước chính là thời gian lâu nhất cần thiết để sửa một đoạn đường ống trên tuyến đường.Tao cần tìm 1 con đường luân chuyển nước sao cho thời gian cần thiết để bảo trì là ngắn nhất. Thêm nữa, do đã sử dụng nhiều năm nên một số ống nước sẽ hỏng dần theo thời gian và không sự dụng được.

b. Input

Dòng đầu tiên chứa 3 số N,M,Q – số điểm, số cạnh và số truy vấn. M dòng tiếp theo chứa 3 số (u,v,w) lần lượt là 2 đầu mút của đoạn ống nước và thời gian cần để bảo trì ống nước. Q dòng tiếp theo chính là các truy vấn chứa 3 số (k,u,v). Truy vấn có 2 loại :

1. K = 1. In ra thời gian cần bảo trì tối thiểu để chuyển nước từ u -> v.
2. K = 2. Xóa cạnh u v.

Lưu ý đây là đồ thị vô hướng.

c. Output

Đối với mỗi query K = 1 in ra đáp án trên 1 dòng riêng biệt.

d. Giới hạn

N - số node <= 1000

M - số edge <= 1e5

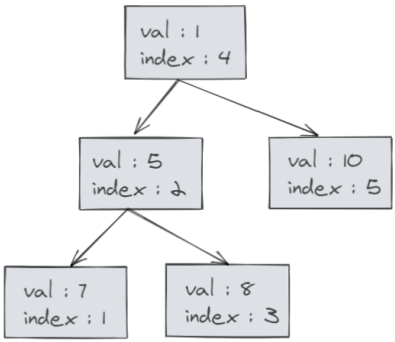
Q - số truy vấn <= 1e5

D - số cạnh xóa <= 5000

Các testcase đảm bảo truy vấn 1 luôn có đáp án.

2. Discussion

Trước hết chúng ta cần nói về mối quan hệ giữa RMQ và LCA. Qua một số ví dụ sau đây chúng ta sẽ nhận ra rằng chúng ta có thể đưa bài toán RMQ về LCA và **ngược lại**.
Cho mảng A gồm N phần tử ta dựng một cây như sau :
Giả sử phần tử nhỏ nhất trong dãy là Ak và node hiện
tại là node Tk. Đối với node trái, mình build



trên đoạn A[1..k-1] và node phải là A[k+1..N].
Ví dụ có mảng A[] = {7, 5, 8, 1, 10}

Mô tả quá trình dựng cây:

-Lấy 1 - phần tử min làm gốc

-Phần bên phải chỉ có 1 phần tử đó là 10

-Phần bên phải có 3 phần tử 7,5,8

-Lấy 5 làm gốc -> build đệ quy

-> 2 node 7,8 theo như quá trình trên

Vậy ta đã dựng xong cây. Đây được gọi là cây Cartesian. Implementation :

```
vector<int> parent(n, -1);
stack<int> s;
for (int i = 0; i < n; i++) {
    int last = -1;
    while (!s.empty() && A[s.top()] >= A[i]) {
        last = s.top();
        s.pop();
    }
    if (!s.empty())
        parent[i] = s.top();
    if (last >= 0)
        parent[last] = i;
    s.push(i);
}
```

Chi tiết bạn đọc có thể tham khảo thêm tại :

(https://cp-algorithms.com/graph/rmq_linear.html)

Nhưng chúng ta dựng cây này làm gì bước tiếp theo chính là gì ?. Các bạn có thể nhận thấy rằng phần tử min trong đoạn [L,R] chính là val của node LCA của node có index L và node có index R.

Như vậy chúng ta có thể giải quyết bài toán RMQ bằng cách ứng dụng LCA (có thể sử dụng thuật toán như thuật toán Tarjan chẳng hạn).

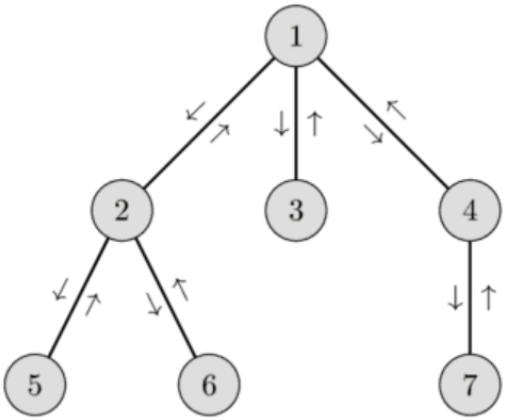
Bằng cách duyệt DFS và lưu lại các node được thăm tại các thời điểm – đây được gọi là Euler's Tour of the tree. Chúng ta nhận thấy rằng qua cách chúng ta dựng lên array

[3]

Euler's Tour thì LCA của node u và node v chính là node có độ sau – depth - thấp nhất trong các node trong subarray [ST[u]….ST[v]] (không mất tính tổng quát giả sử u được thăm trước v) trong đó ST[a] là thời điểm đầu tiên mà node a được thăm. Để giải quyết vấn đề này chúng ta có thể sử dụng RMQ (Segment tree,Sparse Table,..) để tìm node có depth lớn nhất trong range.

(Tìm hiểu thêm tại : https://cp-algorithms.com/graph/lca.html)

(Hình ảnh ví dụ của Euler's Tour of the tree)



Vertices:	1	2	5	2	6	2	1	3	1	4	7	4	1
Heights:	1	2	3	2	3	2	1	2	1	2	3	2	1

3. Solution

Chúng ta có một nhận xét rằng, để xử lý các truy vấn thì chúng ta có thể duy trì một Minimum "Spanning Forest" – vì tất cả các node không thực sự cần form được MST.

Proof : Chúng ta cần chứng minh đáp án của truy vấn có thể tìm qua được Minimum Spanning Forest

Gọi $\Phi(P)$ là số cạnh có ở trong tuyến đường P mà nhưng mà không có trong cây T.

> Nếu $\Phi(P) = 0$, thì P sẽ là 1 con đường đi men trên T
>
> Nếu $\Phi(P) > 0$, thì có cạnh $(u, v) \in P - T$

Giả sử chúng ta có một cạnh (u,v) bất kỳ.

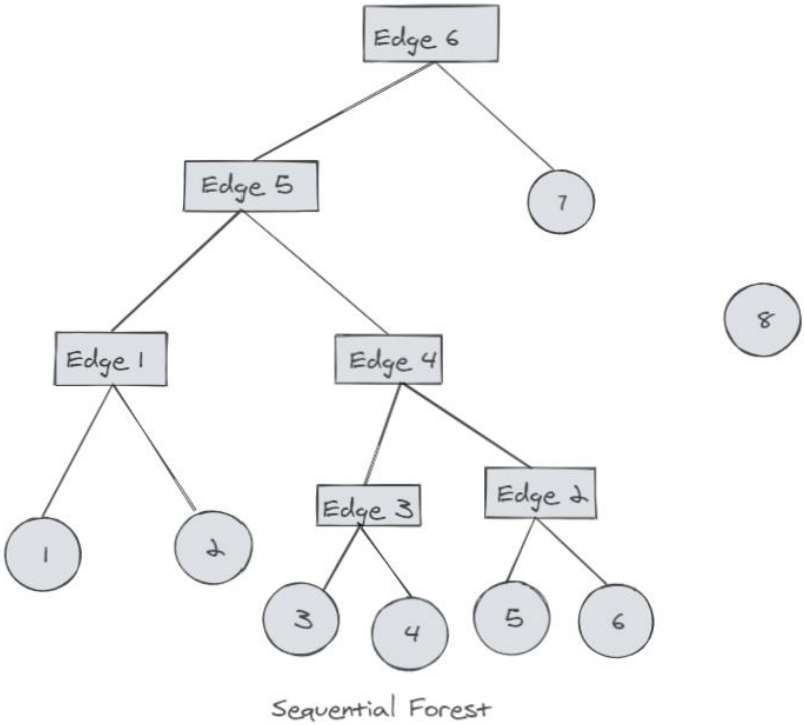
Theo tính chất của Minimum Spanning Tree, tồn tại một con đường P0 chỉ gồm các cạnh của T nối 2 điểm u và v và thời gian bảo trì không vượt quá thời gian bảo trì của cạnh (u,v). Khi đó nếu ta thay thế cạnh (u,v) bằng tuyến đường P0 thì thời gian không tăng lên mà mặt khác Φ(P) còn giảm.
Vì Φ(P) >= 0 do đó P có thể được chuyển thành một path trên T sau hữu hạn bước thay thế như trên điều này chứng minh nhận xét trên.

Do đó truy vấn 1 chúng ta có thể thực hiện trong O(N) trên mỗi query, tuy AC được nhưng thuật toán này vẫn chưa thực sự tối ưu. Thế còn query thứ 2 thì sao ?
Khi xử lý các bài toán như thế này thì để đơn giản hóa vấn đề, để giả lập quá trình xóa các cạnh chúng ta có thể xử lý các truy vấn theo một cách ngược lại, tức xử lý từ truy vấn Q về truy vấn 1 khi đó truy vấn xóa cạnh trở thành truy vấn thêm cạnh, từ đó mà bài toán được đơn giản hơn đi đôi phần. Để xử lý các truy vấn thêm cạnh này cũng như duy trì MST, người ta thường nghĩ đến dung Link-cut Tree với độ phức tạp O(log n) cho mỗi truy vấn (Bạn đọc có thể tìm hiểu thêm trên internet). Tuy nhiên để cài đặt thuật toán trên không phải dễ dàng gì, tại sao chúng ta không nghĩ thêm cách nào khác ?
Chúng ta sẽ cố giải bài toán này dựa trên thuật toán Kruskal. Chúng ta sẽ sử dụng thuật toán Kruskal để dựng nên Minimum Spanning Forrest của đồ thị trên chúng ta sẽ thêm các cạnh một cách tham lam theo trọng số của các cạnh.

> Qua đó chúng ta có nhận xét : Thời gian bảo trì của tuyến đường (u,v) chính là cạnh mà kết nối 2 component riêng biệt mà chúng nằm ở trong với nhau qua thuật toán trên.

Vậy chúng ta nên ứng dụng nhận xét trên như thế nào cho hợp lý ? Chúng ta sẽ dựng nên 1 rừng gọi là "Sequential Forest". Mỗi lá trong Sequential Forest tương ứng với một node trong đồ thị ban đầu. Node lá Vi sẽ tương ứng với node i ban đầu ở trên. Ở trong rừng thì những node không phải là lá sẽ tượng trưng cho các cạnh.

Khi một cạnh thuộc Minimum Spanning Forest được thêm, 2 connected block được connect bằng edge này phải là 2 tree trong sequential forest.



Từ đó theo nhận xét ở trên mà chúng ta có thể nhận thấy rằng thời gian bảo trì chính của tuyến ống nước (u,v) chính là trọng số của node LCA(u,v) trong Sequential Forest. Một khi chúng ta đã dựng được sequential forest thì vấn đề đã đơn giản hơn. Đến đây chúng ta có thể dựng Sparse Table và qua đó trả lời các truy vấn trong O(1) qua đó trả lời tổng thể trong O(n log n + Q). Mặt khác chúng ta vẫn có thể cải thiện được độ phức tạp thuật toán ở đây bằng cách sử dụng thuật toán Tarjan để xử lý trong $O(N. \propto +Q)$. (đọc thêm về thuật toán tìm LCA offline – Tarjan Algorithm trên CP-Algorithm để hiểu rõ hơn)

Vậy cuối cùng chúng ta có một thuật toán với độ phức tạp thuật toán như sau :

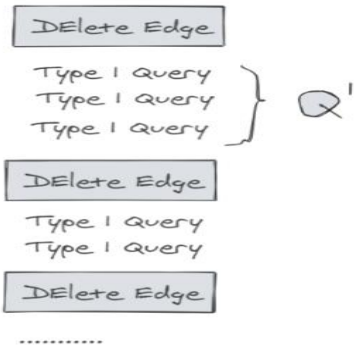
1. Dựng và duy trì Minimum Spanning Forest cũng như Sequential Forest trong $M \log M + DN \propto (n)$.
2. Trả lời truy vấn : DN + Q
   Tổng độ phức tạp : $O(M \log M + DN \propto (n) + Q)$

Qua quá trình giải bài ở trên,các bạn có thể thấy bài toán RMQ&LCA đóng một vai trò quan trọng trong việc học thuật toán cũng như ứng dụng thực tiễn như thế nào.Khi giải bài cũng như nghiên cứu thuật toán, suy nghĩ theo hướng sử dụng các bài toán cơ bản cũng như các thuật toán kinh điển sẽ giúp chúng ta cải thiện được kỹ năng giải quyết vấn đề! Đừng ngại nghĩ, chỉ có làm thì mới có ăn, bài tập trên cũng là một ví dụ điển hình như thế, mình khá chắc chắn rằng đa phần solution các bạn tìm được trên mạng sẽ sử dụng Link-cut Tree thay vì cách đơn giản như trên !

4. Implementation

Chúng ta cùng đi đến phần khó nhất của việc cài đặt thuật toán này đó chính là dựng và duy trì Minimum Spanning Forest. Chúng ta có thể xử lý phần dựng Minimum Spanning Forest bằng cách sử dụng thuật toán Kruskal và sử dụng Disjoint Set Union trong $M \log M$. Nhưng chúng ta phải duy trì nó như thế nào ? Khi chúng ta xóa một cạnh, nếu nó không phải một cạnh thuộc Minimum Spanning Forest thì chúng ta không phải làm gì cả. Còn trong trường hợp còn lại, chúng ta phải thay thế nó bằng cạnh khác, việc tìm cạnh thay thế này có thể xử lý trong O(M) một cách dễ dàng. Nhưng nếu ngược lại thì nếu, thay vì ta xóa cạnh thì nếu thêm cạnh thì mọi thứ sẽ ra sao ? Chúng ta chỉ cần tìm ra Minimum Spanning Forest mới từ MSF cũ và chỉ mỗi cạnh này thôi. Tóm lại, chúng ta cần dựng nên MSF mới với không nhiều hơn N cạnh, việc này có thể xử lý trong $O(N. \propto (n))$. Còn quá trình dựng Sequential Forest có thể được xây dựng đồng thời trong quá trình dựng nên MSF mới nên độ phức tạp tổng thể cho quá trình này không thay đổi.

Theo đề ra thì chúng ta có thể nhận thấy những truy vấn 1 sẽ được chia thành các block nhỏ như hình bên. Ở trong mỗi block đấy chúng ta có thể sử dụng thuật toán Tarjan từ đó có được độ phức tạp là O(N.Q'). Do đó thời gian tổng thể để xử lý là $O(ND + Q)$.



[7]

5. Phụ lục & Chú giải

a. Sparse Table

(https://cp-algorithms.com/data_structures/sparse-table.html)

Sparse Table chính là cấu trúc dữ liệu để xử lý các truy vấn RMQ trong O(1) nhưng trước đó phải xử lý trước trong N log N. Vấn đề duy nhất là cấu trúc dữ liệu này chỉ thích hợp để xử lý các truy vấn offline, không thử xử lý online như Segment Tree.

Ý tưởng chính của Sparse Table chính là tính RMQ trọng mọi subarray với độ dài là số mũ của 2. Chúng ta có cách cài đặt theo hướng QHD như sau :

ST[i][j] : đáp án trong đoạn [i,i+2^j-1] với độ dài 2^j.

```
for (int i = 0; i < N; i++)
    st[i][0] = f(array[i]);

for (int j = 1; j <= K; j++)
    for (int i = 0; i + (1 << j) <= N; i++)
        st[i][j] = f(st[i][j-1], st[i + (1 << (j - 1))][j - 1]);
```

(Tính tổng)

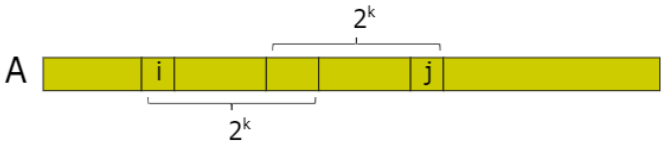
```
int st[MAXN][K + 1];

for (int i = 0; i < N; i++)
    st[i][0] = array[i];

for (int j = 1; j <= K; j++)
    for (int i = 0; i + (1 << j) <= N; i++)
        st[i][j] = min(st[i][j-1], st[i + (1 << (j - 1))][j - 1]);
```

(RMQ)

Sau khi build xong Sparse Table chúng ta trả lời truy vấn RMQ trong O(1) như sau :




```
int j = log[R - L + 1];
int minimum = min(st[L][j], st[R - (1 << j) + 1][j]);
```

```
int log[MAXN+1];
log[1] = 0;
for (int i = 2; i <= MAXN; i++)
    log[i] = log[i/2] + 1;
```

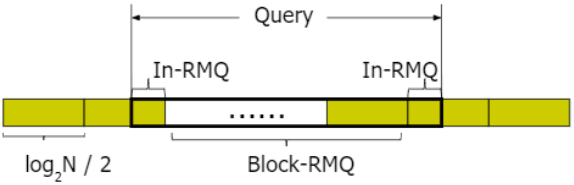
b. +- 1 RMQ

+- 1 RMQ là bài toán tìm RMQ trên mảng A sao cho hai phần tử liền kề bất kỳ có hiệu là +1 hoặc -1. Giả sử mảng A có N phần tử

Ý tưởng chính của thuật toán chính là chia mảng thành các block. Chia A thành M = N/L block trong đó L = log N / 2 – kích thước của mỗi block.

Gọi phần tử min trong block thứ k là Blockmin(k). Tổng hợp lại tất cả giá trị này của M block lại thành một dãy block. Qua ý tưởng chia block, chúng ta có thể chia truy vấn thành 2 phần :

1. Block-RMQ : giá trị min của các block liên tiếp nhau
2. In-RMQ : Giá trị min của một số phần tử từ 2 block ở 2 đầu mút.



Block-RMQ được implement bằng Sparse Table theo phần a thì độ phức tạp cho việc tính toán này là O(MlogM) – O(1)

Vì M log M < 2N / log 2N * log 2N = O(N)

In-RMQ ta nhận thấy bất kỳ 2 số liền kề trong mảng A đều có hiệu là +1 hoặc -1 => có tối đa 2^{L-1} loại block khác nhau. Mỗi loại block có nhiều nhất L^2 truy vấn.

Do đó số truy vấn tối đa là :

[9]

$$O(2^{L-1}L^2) = O(N^{0.5}\log_2^2 N) < O(N)$$

Để xử lý In-RMQ, bạn chỉ cần xử lý trước những truy vấn khác nhau và trả lời

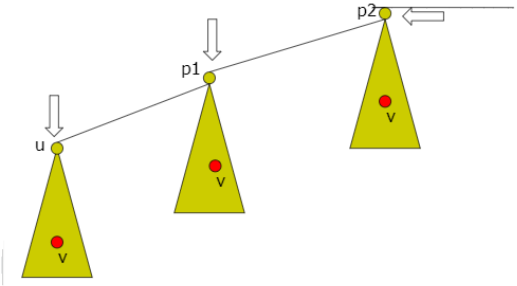
Do đó độ phức tạp thuật toán O(N) – O(1) => Có thể giải quyết bài toán LCA trong O(N) – O(1) (Nhận thấy 2 phần tử bất kỳ trong Euler's Tour of tree liên tiếp nhau đều có hiệu là 1 hoặc -1)

`c. Thuật toán Tarjan để giải LCA trong O(N+Q)`

(https://cp-algorithms.com/graph/lca_tarjan.html)

Thuật toán Tarjan sử dụng Disjoint Set Union để giải quyết mọi truy vấn LCA trong 1 lần duyệt DFS.

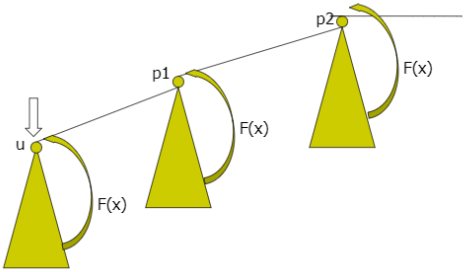
Chúng ta nhóm tất cả truy vấn liên quan đến node u lại với nhau.



TH1 : v nằm trong subtree của u => lca(u,v) = u

TH2 : v nằm trong subtree của p1 => lca(u,v) = p1

TH3 : v nằm trong subtree của p2 => lca(u,v) = 2



[10]

Đối với mỗi node X đang được thăm, chúng ta dung DSU để kết nối nó với F(X) gần nhất trên P.

Giả sử chúng ta đang ở node v, và chúng ta đã thăm node u trong query LCA(u,v) bây giờ chúng ta tìm LCA(u,v) như thế nào ?.

Để ý rằng LCA(u,v) là v hoặc là 1 tổ tiên nào đó của v mà nó cũng chính là tổ tiên của u. Nếu chúng ta cố định node v, thì những node đã được thăm sẽ phân ra thành các disjoint sets. Mỗi node p – tổ tiên của v có một set gồm p và các subtree với root là con của p sao chúng không nằm trên đường đi từ v đi đến root của tree. Set mà chứa u sẽ chính là LCA(u,v).

LCA ở đây là đại diện của một set, nó là một node nằm trên path đi từ v đi đến root tổng của tree. (Xem hình minh họa ở trên).

Từ đó chúng ta nghĩ đến việc DSU để giải quyết. Các bạn có thể tham khảo bản gốc ở CP-Algorithms.

Implementation :

```
vector<vector<int>> adj;
vector<vector<int>> queries;
vector<int> ancestor;
vector<bool> visited;

void dfs(int v)
{
    visited[v] = true;
    ancestor[v] = v;
    for (int u : adj[v]) {
        if (!visited[u]) {
            dfs(u);
            union_sets(v, u);
            ancestor[find_set(v)] = v;
        }
    }
    for (int other_node : queries[v]) {
        if (visited[other_node])
            cout << "LCA of " << v << " and " << other_node
                 << " is " << ancestor[find_set(other_node)] << ".\n";
    }
}

void compute_LCAs() {
    // initialize n, adj and DSU
    // for (each query (u, v)) {
    //    queries[u].push_back(v);
    //    queries[v].push_back(u);
    // }

    ancestor.resize(n);
    visited.assign(n, false);
    dfs(0);
}
```

6. Tổng kết

Tóm lại bài viết này giúp bạn được gì ? Mình mong rằng bài viết này đã giúp bạn có một cái nhìn tổng quát về các bài toán liên quan đến RMQ và LCA. Bài viết đã đưa ra một bài toán với đáp án chuẩn sử dụng một thuật toán phức tạp và chúng ta đã cố gắng từng bước cải thiện những thuật toán cổ điển để đưa bài toán về một lời giải đơn giản hơn, dễ cài đặt hơn và qua đó rèn luyện cũng như cải thiện kỹ năng problem solving cho bản thân! Song, bài viết này mình xin phép không đính kèm Source Code ở đây một phần là do dung lượng của bài viết, phần khác là do mình muốn các bạn hãy thử tự cài đặt thuật toán trên qua đó mới cải thiện được kỹ năng implementation. Các bạn có thể nộp bài này tại : https://www.luogu.com.cn/problem/P4172.

Bên cạnh đó mình cũng có một số bài tập luyện tập thêm về chủ đề này :

1. https://atcoder.jp/contests/JAG2013Spring/tasks/icpc2013spring_e

2.https://szkopul.edu.pl/problemset/problem/-fb7NxSJGXxkJ2Om5FvXzbil/site/?key=statement

(POI 2002 – IX – Problem kom)

Tất cả source code trong bài viết này bao gồm kiến thức cơ bản, bài tập sẽ được đăng lên cùng với giải thích trên github của mình tại đây, đừng ngại cho mình một star nhé :3

https://github.com/thisiscaau/china-paper-archive

Bài viết đến đây là kết thúc, mong các bạn đã học được thêm một số kiến thức bổ ích. May the force be with you !