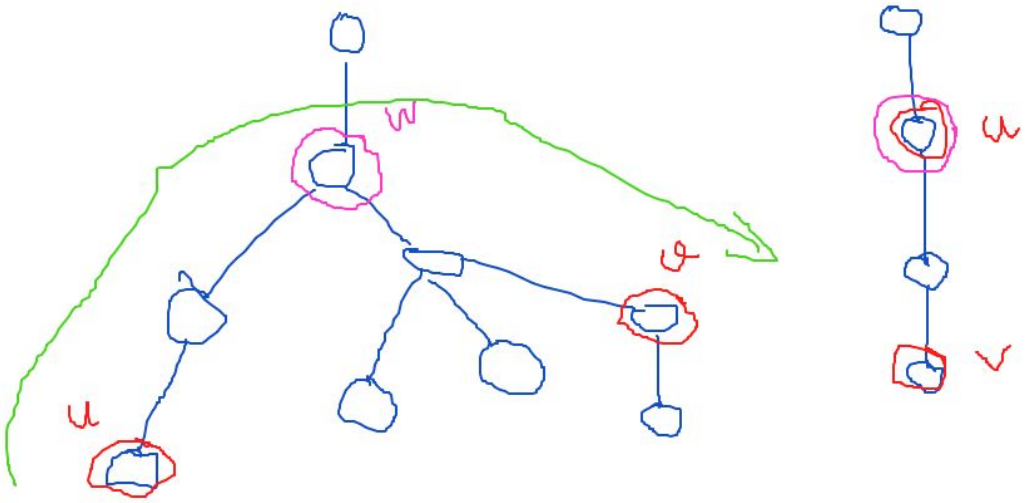
Tổ tiên chung gần nhất

Cho một cây (cố định gốc là 1), và hai nút (u, v); tìm tổ tiên chung gần nhất (nút w gần u và v nhất) sao cho w vừa là tổ tiên của u vừa là tổ tiên của v.



Khi nói đến tổ tiên chung gần nhất, ta chú ý có hai trường hợp:
+ Trường hợp 1: Hai nút u và v không có quan hệ tổ tiên: Khi đó, tổ tiên chung gần nhất w là một nút thứ ba khác u và v. Đồng thời, khi xét các nhánh con trong cây con gốc w, u và v thuộc hai nhánh khác nhau.
+ Trường hợp 2: Hai nút u và v có quan hệ tổ tiên. Không mất tổng quát, ta giả sử u là tổ tiên của v. Khi đó, tổ tiên chung gần nhất của u và v chính là u (là nút gần gốc hơn).

Thuật toán duyệt

Để tìm tổ tiên chung gần nhất giữa hai đỉnh u và v, ta thực hiện theo hai bước như sau. Trước tiên, không mất tổng quát, ta giả sử độ sâu của u lớn hơn hoặc bằng độ sâu của v (độ sâu của một nút là số cạnh từ nút đó đến gốc). Thuật toán tìm tổ tiên chung gần nhất diễn ra trong hai bước:
+ Bước 1: Cân bằng độ sâu: Di chuyển nút u theo hướng về phía gốc cho đến khi độ sâu của nút u bằng với độ sâu của nút v. Như đã giả sử, ban đầu độ sau của nút u lớn hơn hoặc bằng độ sâu của nút v. Nếu độ sâu của nút u lớn hơn, ta cần di chuyển lên cho độ sâu hai nút bằng nhau.
+ Bước 2: Khi độ sâu của u và v đã cân bằng, ta di chuyển hai nút đồng thời từng bước một, cho đến khi hai nút gặp nhau.

```
int numNode;
vector<int> adj[MAX]; // mảng danh sách kề
int par[MAX]; // mảng lưu cha của các nút
int high[MAX]; // mảng lưu độ sâu của các nút (gốc cây có độ sâu là 0)

void dfs(int u) {
      for (int v : adj[u]) if (v != par[u])) {
            par[v] = u;
```

```
            high[v] = high[u] + 1;
            dfs(v);
      }
}
dfs(1);

int lcaSlow(int u, int v) { // tìm tổ tiên chung gần nhất giữa u và v
      // nếu v có độ sâu lớn hơn u, đổi chỗ u và v để ta luôn có thể giả sử u có độ sâu lớn
hơn hoặc bằng v.
      if (high[v] > high[u]) return lcaSlow(v, u);

      // cân bằng độ sâu: di chuyển nút u lên trên cho đến khi có độ sâu bằng v
      while (high[u] > high[v]) u = par[u];

      // di chuyển u và v cùng lúc cho đến khi hai nút gặp nhau
      while (u != v) {
            u = par[u];
            v = par[v];
      }

      return u; // tổ tiên chung gần nhất chính là điểm gặp nhau đầu tiên
}
```

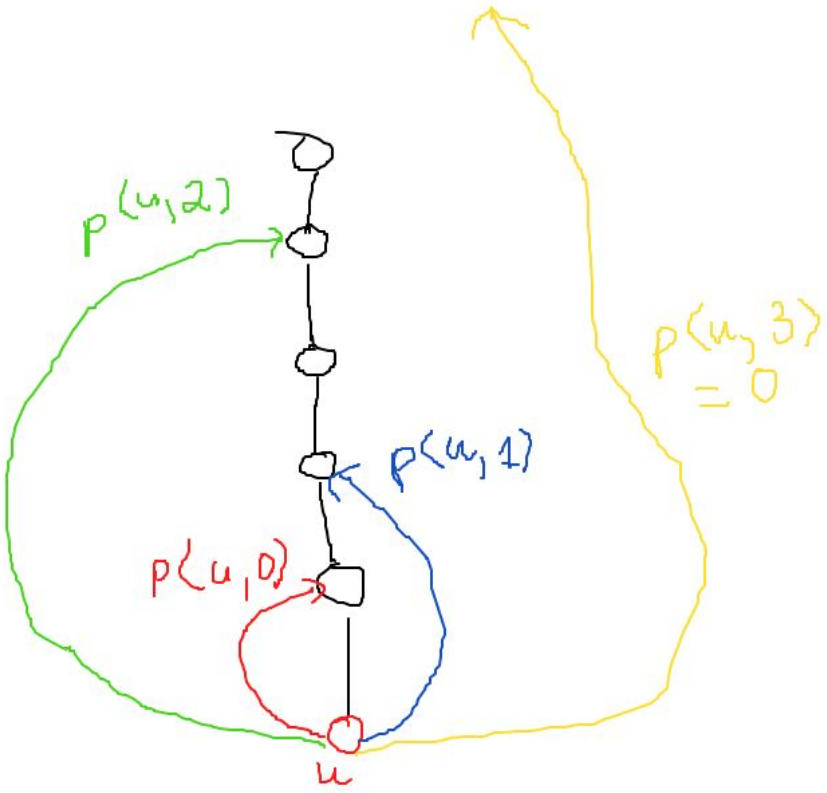
Thuật toán nhanh

Thuật toán tìm tổ tiên chung gần nhất trên cây với độ phức tạp O(logn) cho mỗi truy vấn diễn ra với cùng một ý tưởng như thuật toán chậm. Tuy nhiên, có ý tưởng đột phá chính: Ở thuật toán duyệt, mỗi bước ta chỉ di chuyển mỗi đỉnh lên cha trực tiếp (giảm độ sâu đi 1). Như vậy, nếu khoảng cách từ tổ tiên chung gần nhất tới hai nút rất xa, thuật toán có thể có thời gian chạy lên tới O(n). Vì vậy, để giảm độ phức tạp của thuật toán, ta cần phải tìm cách sao cho mỗi bước có thể nhảy các nút lên nhiều nấc cùng một lúc.

Ở đây, thuật toán nhanh sử dụng ý tưởng nhảy theo các bước có độ dài là lũy thừa của hai. Trước tiên, ta chú ý đến các điểm giống và khác trong tư tưởng của thuật toán nhanh.
+ Giống như thuật toán chậm, thuật toán nhanh cũng cần tiền xử lý để giả sử u có độ sâu lớn hơn v.
+ Ở bước 1, thuật toán nhanh cũng có mục đích là cho nút u có độ sâu cân bằng với nút v.
+ Sau bước 1, ở thuật toán nhanh cần thêm tiền xử lý: Nếu u và v hiện nay đang là cùng một nút, kết luận luôn tổ tiên chung gần nhất của u và v là vị trí hiện tại (bản chất, ở trường hợp này, v là tổ tiên của u, và tổ tiên chung gần nhất của u và v chính là v).
+ Ở bước 2, thuật toán nhanh cũng di chuyển u và v từng bước với các bước có cùng độ dài, tuy nhiên mục tiêu của bước 2 trong thuật toán nhanh là chuyển u và v tới điểm gần nhất khi hai nút vẫn chưa gặp nhau. Cụ thể, sau bước này hai nút u và v vẫn có cùng độ sâu, nhưng chúng chỉ cách tổ tiên chung gần nhất thực sự của chúng đúng 1 bước và chỉ cần thêm một bước nữa là tới LCA.
+ Ở bước 3, đưa u và v thêm một bước nữa, tới vị trí hai nút gặp nhau và kết luận điểm gặp nhau này là LCA.

Thuật toán nhanh tăng tốc bằng ý tưởng là mỗi lần lặp của vòng lặp while, các nút thay vì chỉ đi một bước (chỉ giảm độ sâu đi 1), thì giờ sẽ giảm độ sâu một lượng bằng 2^k

Trước tiên, ta cần chuẩn bị trước mảng cha: Gọi par(u, j) là vị trí của nút u sau khi đi lên trên 2^j bước. Nếu độ sâu của nút u nhỏ hơn 2^j, ta coi par(u, j) = 0. Chú ý, các đỉnh được đánh số từ 1 đến n.

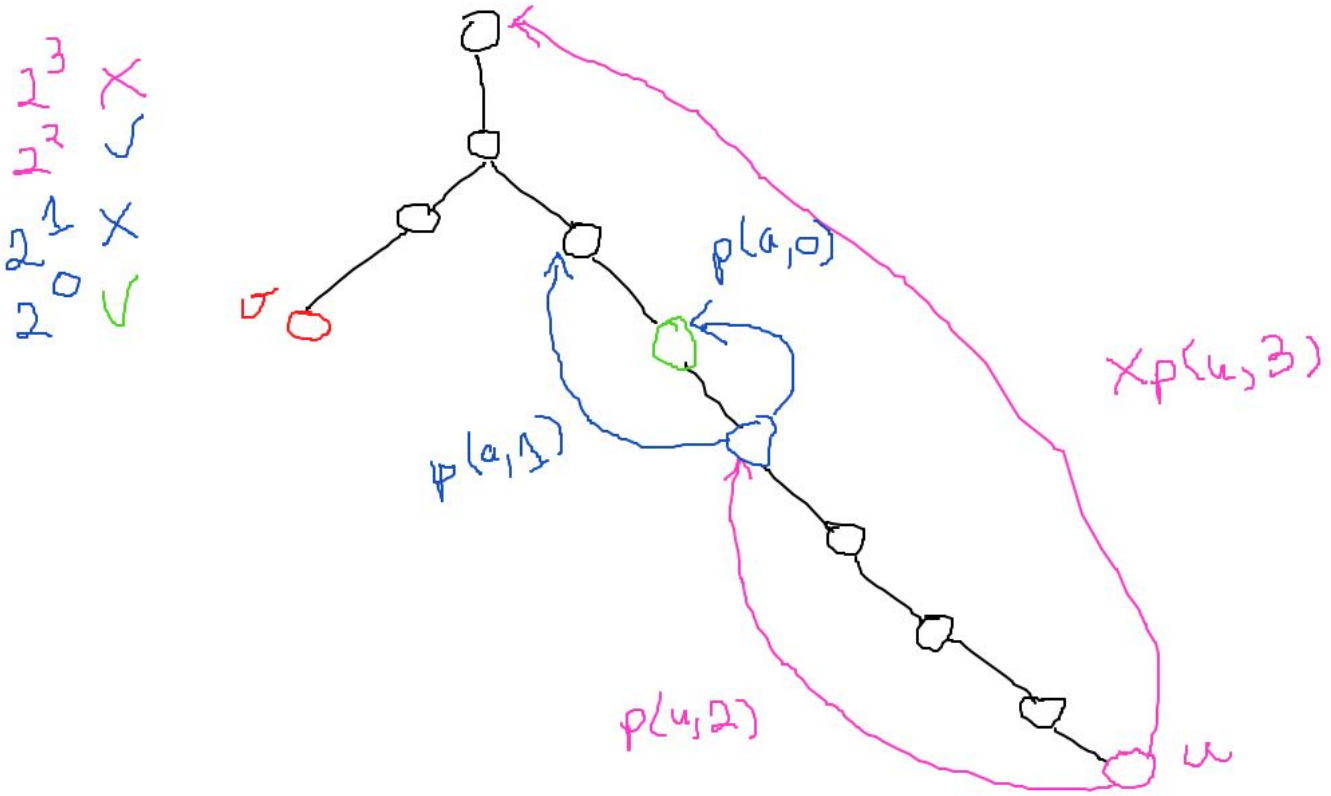


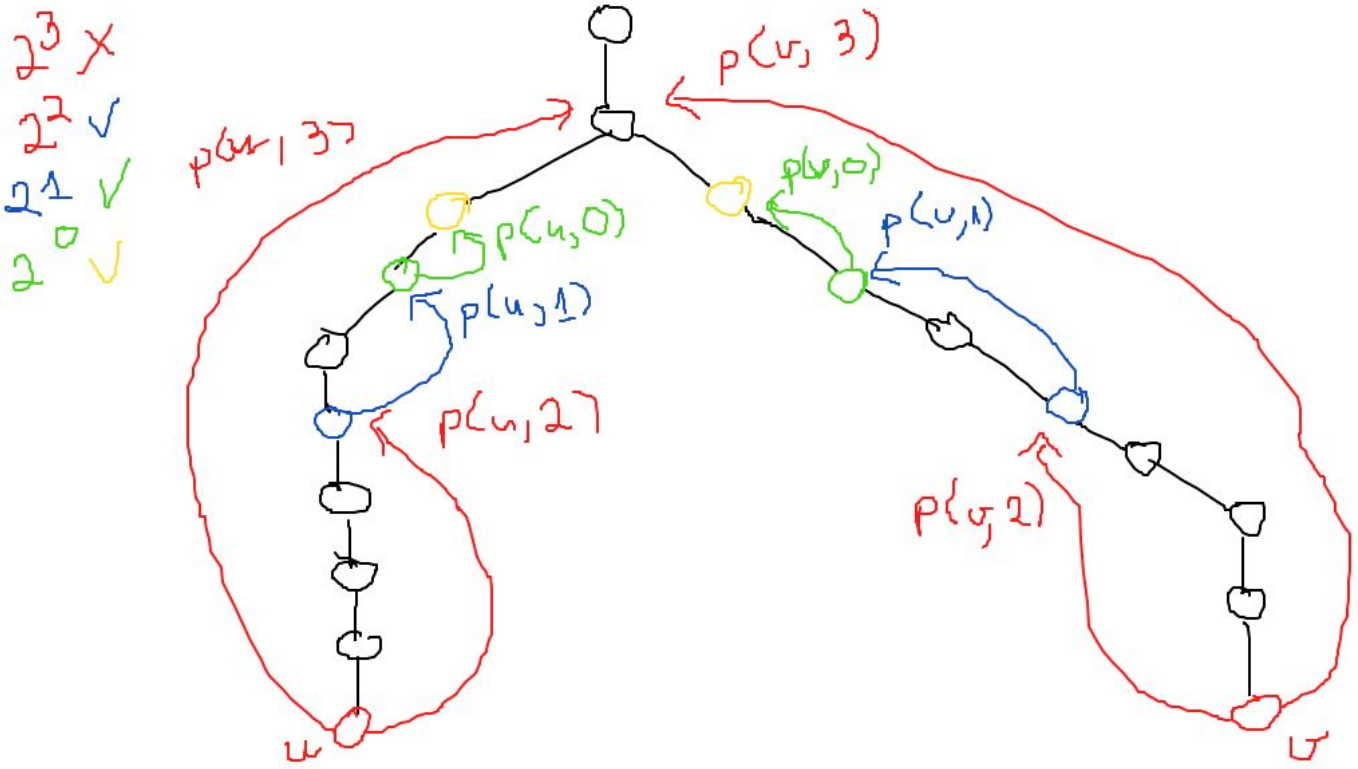
Việc tính mảng par này như sau: par(u, 0) chính là cha trực tiếp của nút u. Còn lại, để nhảy từ u lên trên 2^j bước, ta chia làm hai lần nhảy, mỗi lần nhảy 2^(j-1), Do đó, giả sử v = par(u, j -1) thì par(u, j) = par(v, j - 1). Hay nói cách khác, ta có par(u, j) = par(par(u, j -1), j-1).

```
void dfs(int u) {
	for (int v : adj[u]) if (v != par[u])) {
		par[v][0] = u;
		high[v] = high[u] + 1;
		dfs(v);
	}
}

dfs(1);
for (j = 1; j <= LOG; j++) for (int i = 1; i <= n; i++)
	par[i][j] = par[par[i][j - 1]][j - 1];
```

Ý tưởng của việc nhảy nhanh trong 2 bước 1 và 2 của thuật toán nhanh được gói gọn theo phương châm: "cứ nhảy được thì nhảy". Cụ thể, ta duyệt qua các lũy thừa của 2, từ lũy thừa lớn nhất (2^log(n)) tới lũy thừa nhỏ nhất (2^0). Khi duyệt tới lũy thừa 2^i, ta kiểm tra, nếu việc nhảy vẫn còn được thì cứ nhảy không thì thôi.





Hình ảnh này mô phỏng quá trình nhảy ở giai đoạn 2. Bản chất ở trên ta thấy tổng quãng đường cần nhảy là 7. Cách làm trên thực chất mô phỏng quá trình phân tích số 7 ra hệ nhị phân (viết lại 7 dưới dạng tổng các lũy thừa của 2). Cụ thể như sau:

+ Xét số 2^3, do 2^3 > 7, nên bit thứ 3 là 0, không nhảy

+ Xét số 2^2, do 2^2 <= 7, nên bit thứ 2 là 1, nhảy tiếp 2^2 đơn vị. quãng đường còn lại là 3
+ Xét số 2^1, do 2^1 <= 3, nên bit thứ 1 là 1, nhảy tếp 2^1 đơn vị, quảng đường còn lại là 1.

Cài đặt hoàn chỉnh

```
#define MAX   100100
#define LOG   17
int numNode;
vector<int> adj[MAX]; // danh sách kề
int par[MAX][LOG + 1]; // mảng cha
int high[MAX]; // số cạnh từ một đỉnh tới gốc (độ sâu)

void dfs(int u) {
      for (int v : adj[u]) if (v != par[u][0]) {
            par[v][0] = u; // cha trực tiếp của v là u
            high[v] = high[u] + 1;
            dfs(v);
      }
}

// chuẩn bị ban đầu
high[0] = -1; // rất quan trọng, không thể bỏ sót
dfs(1);
for (int j = 1; j <= LOG; j++) for (int i = 1; i <= numNode; i++)
      par[i][j] = par[par[i][j - 1]][j - 1];

int lca(int u, int v) {
      // nếu v có độ sâu lớn hơn u, đổi chỗ u và v
      if (high[v] > high[u]) return lca(v, u);

      // bước 1: nhảy u lên để u có độ sâu bằng v
      for (int j = LOG; j >= 0; j--) if (par[u][j] >= par[v]) u = par[u][j];

      // nếu sau bước 1, hai đỉnh u và v đã bằng nhau rồi, kết luận đây là lca
      if (u == v) return u;

      // bước 2: nhảy đồng thời u và v đến vị trí xa nhất mà chưa gặp nhau
      for (int j = LOG; j >= 0; j--) if (par[u][j] != par[v][j]) {
            u = par[u][j]; v = par[v][j];
      }

      // nhảy thêm một bước nữa là tới lca
      return par[u][0];
}
```

Một số lưu ý đối với cách cài đặt trên
- Cách cài đặt này có tính tiện lợi và ngắn gọn cao, tuy nhiên, dựa trên giả thiết rằng các đỉnh được đánh số từ 1 đến n. Nếu đánh số các đỉnh của cây từ 0, cách cài đặt có rất nhiều chỗ cần phải thay đổi.

- Cách cài đặt này giả thiết rằng mảng par đã được khởi gán là 0. Nếu ban đầu mảng par có các phần tử khác 0, cần phải memset (khởi tạo) lại để bằng 0.
- Lệnh high[0] = -1 là rất quan trọng và không thể thiếu với cách cài đặt này.
- Nên sử dụng một hằng số LOG là logarit cơ số 2 của giá trị max của n. Không nhất thiết phải tính log_n phụ thuộc vào số n thực tế. Thêm nữa, tuyệt đối không sử dụng hàm log để tính log cơ số 2 vì hàm này có thể gây ra sai số rất lớn.
- Giới hạn mảng par phải là LOG + 1, không phải là LOG.
- Để test nhanh tính đúng đắn của việc cài LCA, có thể thử ba trường hợp sau:
+ Kiểm tra LCA giữa hai nút không có quan hệ tổ tiên
+ Kiểm tra LCA giữa một nút là tổ tiên của nút còn lại (kết quả phải là nút tổ tiên)
+ KIểm tra LCA giữa gốc và một nút bất kỳ trên cây.

Practice: https://vn.spoj.com/problems/LCA

```
int jump(int u, int k) { // nhảy từ nút u lên trên k bậc
      FORD(i, LOG, 0) if (BIT(k, i) == 1) u = par[u][i];
      return u;
}

// để kiểm tra xem u có phải tổ tiên của v hay không: v thuộc cây con gốc u (u là tổ tiên
của v) khi và chỉ khi high[v] >= high[u] && jump(v, high[v] - high[u]) == u.

// để tìm nút con trực tiếp w của u sao cho v thuộc cây con gốc w:
w = jump(v, high[v] - high[u] - 1);
```