Heavy-Light Decomposition
(тежко-леко разлагане)

Какво е Heavy-Light Decomposition (HLD)?

Чували сте за задачи, които звучат по следния начин:
„Дадено претеглено дърво ..., заявки за модифициране на ребра и заявки за намиране на минимума на ребрата между два върха.“

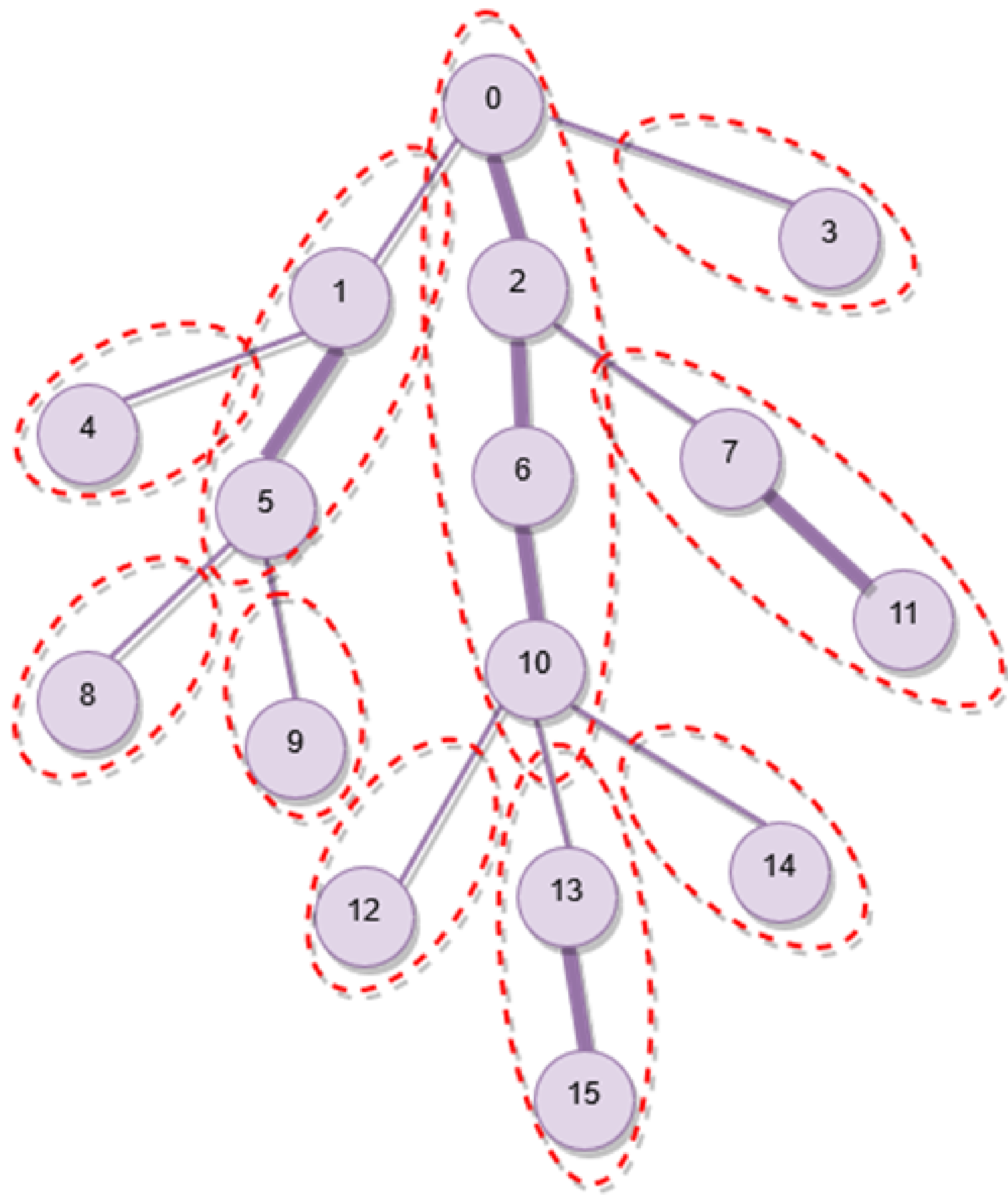
Решавали сте задача, подобна на тази, използвайки сегментни дървета. Но проблемът тук е, че ни е дадено дърво. Докато сегментно дърво може да бъде изградено с помощта на масив. *Така че, ако можем да намалим даденото дърво и да го представим с масиви, проблемът е решен!* За да направим това, може да използваме техника за данни, наречена „**heavy-light decomposition**“.

HDL прави точно това.

Какво е Heavy-Light Decomposition (HLD)? (2)

Нека ни е дадено дърво G, съдържащо n върха с произволен корен. HLD е техника за разлагане на кореново дърво в множество от пътища.

Същността на това разлагане на дървото е да се **раздели дървото на няколко пътя/вериги**, за да се достигне до коренния връх от всяко **v** чрез преминаване на най-много log n пътища.
Освен това нито един от тези пътища не трябва да се пресича с друг път. Пътят е поредица от възли, свързани един след друг. Може да се разглежда като прост масив от възли (числа). Можем да извършваме много операции върху масив от елементи със сложност O(log n), използвайки сегментно дърво и други структури от данни.


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Защо се нуждаем от HDL?

Нека имаме задача, в която се казва: Дадено ви е небалансирано дърво от n възела и трябва да извършите действия, за да отговорите на q заявки. Всяка заявка може да бъде от два вида:

a) **update(p,q)** - актуализиране на стойността на възела със стойност p със стойността на възела q.

b) **maxEdge(p,q)** - в пътя от p до q изведете възела с максимална стойност.

За решаването на тази задача може да се намери решение, което преминава през цялото дърво веднъж при всяка заявка. Това решение ще има сложност по време **O(q*n)**, тъй като преминаването през дървото отнема време O(n). *Това не е ефикасно решение*.

Можем да решим задачата за логаритмично време, като използваме HDL.

Алгоритъм - основна идея

Разделяме дървото на вериги, включващи несвързани върхове (което означава, че две вериги нямат общ възел) по такъв начин, че за да преминем от който и да е връх в дървото към корeновия връх, ще трябва да променим най-много log n вериги.
Или пътят от всеки връх до корена може да бъде разделен на части, така че всеки елемент да принадлежи само на една верига, тогава няма да имаме повече от log n части.

Защо log n?

Балансирано двоично дърво от n върха има височина log n. Трябва да посетим повечето log n възли, за да достигнем до корена от всеки друг връх. По този начин, дори ако променим веригата на всяко ниво на височина, ще трябва да променим не повече от log n вериги най-много.

Използвани термини

- **Тежко дете**: Тежко дете на възел е детето с най-голям размер на поддърво, вкоренено в детето. Всеки възел, който не е лист има точно едно тежко дете.
- **Тежко ребро**: За всеки възел, който не е лист, реброто, свързващо възела с неговото тежко дете, се счита за тежко ребро. Всички останали ребра са означени като леки.
- **Леко дете** на възел е всяко дете, което не е тежко дете.
- **Леко ребро** свързва възел с което и да е от неговите леки деца.
- **Тежък път** е път, образуван от поредица от тежки ребра.
- **Лек път** е пътят, образуван от последователност от леки ребра.

Алгоритъм

1. Първо, изчисляваме размера на поддървото ***s(v)***, за всеки връх ***v***.
2. Разглеждаме всички ребра, водещи до децата на връх ***v***.

Наричаме ребро тежко, ако води до връх ***c***, така че:

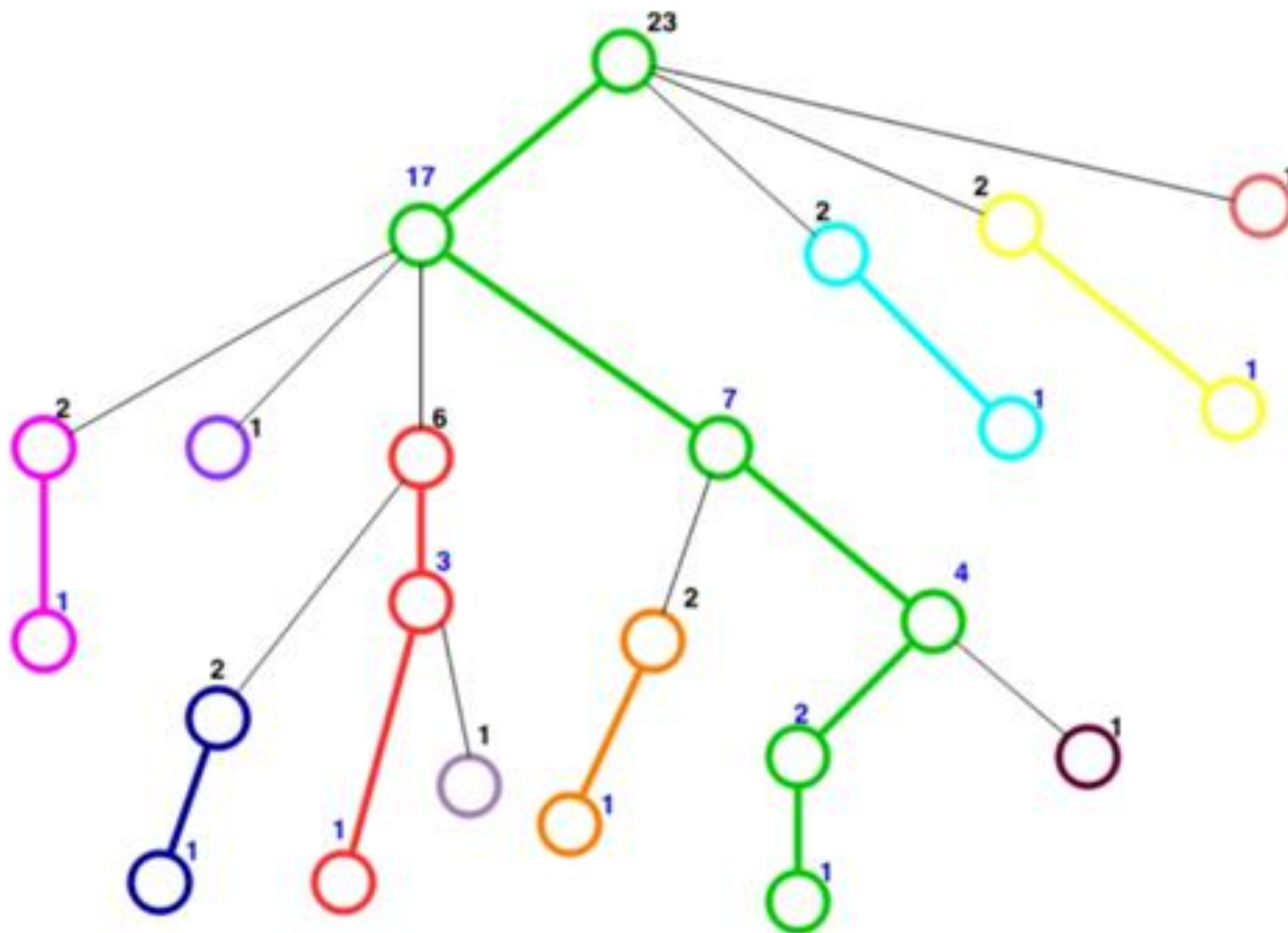
s(c)≥s(v)/2 ⟺ реброто (v,c) е тежко.

Всички останали ребра са означени като леки.

3. Разлагаме дървото на несвързани пътища.

Да разгледаме всички върхове, от които не слизат тежки ребра. Ще се изкачваме от всеки такъв връх, докато стигнем до корена на дървото или преминем през леко ребро. В резултат на това ще получим няколко пътя, които са съставени от нула или повече тежки ребра плюс едно леко ребро. Пътят, който има край в основата, е изключение и няма да има леко ребро. Нека ги наречем **тежки пътища** – това са желаните пътища на HLD.

Примерно дърво с готово HLD



Всяка верига е представена с различен цвят. Черните ребра свързват две вериги

Как да решаваме задачи с помощта на HLD?

update(p,q)

1. Над всеки тежък път изграждаме сегментно дърво, което ще ни позволи да търсим връх с максималната зададена стойност в посочения сегмент на дефинирания тежък път за време O(log n).

2. Извършваме актуализацията, като използваме актуализацията на сегментните дървета.

3. Следователно времевата сложност на операцията за актуализиране ще бъде O(log n).

Как да решаваме задачи с помощта на HLD? (2)

maxEdge(p,q)

1. Над всеки тежък път изграждаме сегментно дърво, което ще ни позволи да търсим връх с максималната присвоена стойност в посочения сегмент от посочения тежък път за време O(log n).

2. За да отговорим на заявката query (a, b), намираме най-малкия общ предшественик на a и b (LCA) или по някакъв друг метод.

A) Сега задачата е намалена до две заявки (a, l) и (b, l), за всяка от които можем да направим следното: да намерим тежкия път, в който се намира долния връх, да изпълним заявка по този път, придвижваме се до върха на този път, отново определяме на кой тежък път сме и правим заявка по него и така нататък, докато стигнем до пътя, съдържащ l.

Как да решаваме задачи с помощта на HLD? (3)

B) Трябва да се внимава със случая, когато например a и l са на един и същ тежък път - тогава максималната заявка по този път трябва да се направи не на който и да е префикс, а на вътрешния участък между a и l.

C) Отговарянето на подзаявките (a, l) и (b, l) изисква преминаване през $O(\log n)$ тежки пътя и за всеки път се прави максимална заявка за някакъв участък от пътя, което отново изисква $O(\log n)$ операции в сегментното дърво. Следователно една заявка (a, b) отнема $O(\log^2 n)$ време.

Реализация

За да конструираме HLD, трябва да изпълним 2 стъпки.

- Да изчислим размера на поддървото за всеки възел
- Да разложим дървото на несвързани вериги

Изчисляване на размера на поддървото за всеки възел

За тази цел можем да изолзваме dfs.

Размерът на поддървото на всеки възел е сумата от *(размерите на поддървото на всички негови деца)+1*.

Освен това размерът на поддървото на всички листови възли е 1, тъй като нямат деца.

```
int subtree_size[N] = {0};
// subtree_size[i] размер на поддървото на i-тия възел.
int dfs(int curr,int parent) {
// размерът на поддървото на даден възел = размерите на
// поддървото на всички негови деца +1.
  int ans=1;
  for(auto x:adj[curr]){
    // Добавяне размерите на поддървото на децата
    if(x!=parent)ans+=dfs(x,curr);
  }
// Актуализиране subtree_size на текущия възел с ans и
// връщане
 return subtree_size[curr] = ans;
}
```

Разлагане на дървото на несвързани вериги

Създаваме функция HLD, на която предаваме възела и текущата верига. Във функцията:

- Добавяме текущия възел към текущата верига.
- Ако текущият възел е листов възел, просто прекратяваме процеса и се връщаме.
- В противен случай преминаваме през всеки дъщерен възел на текущия възел и намираме тежкото му дете.
- Извикваме функцията HLD за тежкия възел със същата верига.
- За леките деца на текущия възел извикваме функцията, но с нова празна верига, тъй като старата верига няма да продължи след това.

Горните стъпки реализират правилно HLD, но в кода ще ни липсва информация, от която имаме нужда.

Трябва да намерим отговора на въпросите:

1. Към коя верига принадлежи даден възел?
2. Каква е позицията на даден възел във веригата му?
3. Кое е началото на дадена е верига, кой е първият му възел?
4. Каква е дължината на дадена е верига?

За да отговорим на тези въпроси, ще използваме променливите:

`chainNo` - указва текущия номер на веригата, на която се намираме. (инициализираме го с 0, тъй като в началото няма да има вериги)
`chainHead[N]` - указва началото на веригата, към която i-тият възел принадлежи. (Инициализираме всеки възел с -1, тъй като първоначално няма верига, така че няма начало)
`chainPos[N]` - указва позицията на i-тия възел във веригата.
`chainInd[N]` - указва индекса на веригата, към която i-тият възел принадлежи.
`chainSize[N]` - указва размера на веригата, към която принадлежи i-ят възел. (инициализираме го с 0, тъй като първоначално всички размери ще бъдат 0)

Във функцията HLD, за текущия възел:

- Проверяваме дали началото на веригата на текущия номер на веригата е -1 или не. Ако не е -1, това означава, че не се образува нова верига от този възел и той е част само от по-ранната верига. Така че началото остава същото. В противен случай сме започнали нова верига от текущия възел и по този начин **`chainHead`** на текущия номер на веригата ще бъде този възел.

- Актуализираме стойността на **`chainIn`** на възела с текущия номер на веригата. Актуализираме стойността на позицията на веригата на текущия възел с размера на веригата, към която принадлежи. Увеличаваме размера на текущата верига с 1.

- Ако текущият възел не е листо, намераме неговия тежък дъщерен елемент, който има максимален размер на поддървото.

- Ако имаме тежко дете, викаме HLD за тежкото дете, като запазим същата верига.

- За по-леките деца, викаме HLD, като увеличаваме номера на веригата с 1 всеки път, тъй като ще има ново образуване на верига от всяко такова дете.

```
void hld(int cur) {
/*
 проверка дали началото на веригата на текущия номер на
 веригата е -1 или не. Ако не е -1, това означава, че не се
 образува нова верига от този възел и той е част само от
 по-ранната верига. Така че началния възел остава същият.
 В противен случай трябва да започнем нова верига от текущия
 възел и по този начин chainHead на текущия номер на
 веригата ще бъде този възел.
*/
  if(chainHead[chainNo] == -1) chainHead[chainNo]=cur;

//Актуализиране стойността на chainIn на възела с текущия
//номер на веригата.
    chainInd[cur] = chainNo;
```

```
//Актуализиране стойността на позицията на веригата на
//текущия възел с размера на веригата, към която принадлежи.
    chainPos[cur] = chainSize[chainNo];

//Увеличаване размера на текущата верига с 1.
    chainSize[chainNo]++;

// Намиране на тежко дете
    int heavy_child = -1,mx = -1;
    for(int i = 0; i < adj[cur].sz; i++) {
        if(subtree_size[ adj[cur][i] ] > mx) {
            mx = subtree_size[ adj[cur][i] ];
            heavy_child = i;
        }
    }
```

```
//Ако имаме тежко дете, извикваме HLD за тежкото дете,
//като запазваме същата верига.
    if(heavy_child >= 0) hld( adj[cur][heavy_child] );

//За леките деца, извикваме HLD, като увеличаваме номера
//на веригата с 1 всеки път, тъй като ще има нова верига,
//образувана от всяко такова дете.
    for(int i = 0; i < adj[cur].sz; i++) {
        if(i != heavy_child) {
            chainNo++;
            hld( adj[cur][i] );
        }
    }
}
```

Задача

Дадено е дърво, състоящо се от n възела. Възлите са номерирани 1, 2,…, n. Всеки възел има зададена стойност. Вашата задача е да обработвате следните видове заявки:

- променете стойността на възел s на x;
- намерете максималната стойност в пътя между възли a и b.

Вход

Първият ред на стандартния вход съдържа две цели числа n и q: броят на възлите и заявките. Възлите са номерирани 1, 2,…, n. Следващият ред има n цели числа $v_1,v_2,\ldots,v_n$: стойността на всеки възел. След това има n-1 реда, описващи ребрата. Всеки ред съдържа две цели числа a и b: между възлите a и b има ребро. И накрая, има q реда, описващи заявките. Всяка заявка е във формата "1 s x" или "2 a b".

Изход

Отпечатайте отговора на всяка заявка от тип 2 на отделен ред.

Ограничения

$1 \le n, q \le 2.10^5$

$1 \le a, b, s \le n$

$1 \le v_i, x \le 10^9$

Пример

Вход:

```
5 3
2 4 1 3 3
1 2
1 3
2 4
2 5
2 3 5
1 2 2
2 3 5
```

Изход:

```
4 3
```

```
#include <bits/stdc++.h>
using namespace std;
#define fi first
#define se second
#define pb push_back
const int mxN = (int)2e5+9;
const int mxLg = 18;
const int INF = (int)1e9;
vector<int> adj[mxN];
int n, q, tim=1, a[mxN], st[mxN], h[mxN], p[mxN];
int sub[mxN], dep[mxN], segTree[mxN*2], jmp[mxLg][mxN];

void update(int i, int v){
    segTree[i+=n]=v;
    for(i/=2;i;i/=2)
segTree[i]=max(segTree[2*i],segTree[2*i+1]);
}
```

```
int query(int i, int j){
    int ra=0,rb=0;
    for(i+=n,j+=n+1; i<j; i/=2,j/=2){
        if(i&1) ra=max(ra,segTree[i++]);
        if(j&1) rb=max(rb,segTree[--j]);
    }
    return max(ra,rb);
}
void dfs(int s, int par){
    sub[s] = 1; jmp[0][s]=p[s]=par;
    if(par) dep[s] = dep[par]+1;
    for(auto &u : adj[s]){
          if(u==par) continue;
          int x = adj[s][0]; dfs(u,s); sub[s]+=sub[u];
        if(x==par or sub[x]<sub[u]) swap(adj[s][0],u);
    }
}
```

```
 void Hld(int s, int head){
   st[s]=tim++; h[s] = head;
   for(auto u : adj[s]) if(u!=p[s])Hld(u,u==adj[s][0]?head:u);
}
int getPath(int x, int k){
     for(int i = 0; i < mxLg; i++)if((k>>i)&1) x = jmp[i][x];
     return x;
}
int lca(int a, int b){
 if(a==b) return a;
 if(jmp[0][a]==jmp[0][b]) return jmp[0][a];
 if(dep[a] > dep[b]) swap(a,b);
 b = getPath(b,dep[b]-dep[a]);
 for(int i = mxLg-1; i>=0; i--)
  if(jmp[i][a]!=jmp[i][b] and jmp[i][a])a=jmp[i][a],b=jmp[i][b];
     return lca(a,b);
}
```

```
int calc(int a, int b){
 int ans = 0, ok=1;
 while(ok){
  if(h[a]!=h[b]) {ans = max(ans, query(st[h[a]],st[a]));
                   a = p[h[a]];}
  else {ans = max(ans, query(st[b],st[a])); ok=0;} }
 return ans;
}
int32_t main(){
    ios_base::sync_with_stdio(false); cin.tie(0);
    cin >> n >> q;
    for(int i = 1; i <= n; i++) cin >> a[i];
    for(int i = 1; i < n; i++){int a, b; cin >> a >> b;
        adj[a].pb(b), adj[b].pb(a);
    }
    dfs(1,0); Hld(1,1);
```

```
     for(int i = 1; i < mxLg; i++)
          for(int j = 1; j <= n; j++)
               jmp[i][j] = jmp[i-1][jmp[i-1][j]];
    for(int i = 1; i <= n; i++) update(st[i],a[i]);
    while(q--){
        int t, x, y, L;
        cin >> t >> x >> y;
        if(t==1) update(st[x],y);
        else {
          L=lca(x,y),
          cout << max(calc(x,L),calc(y,L)) << endl;
        }
    }
}
```

Задачи

https://usaco.guide/plat/hld?lang=cpp