pbds 1

1.1 前言

這份講義要介紹的是一個 GNU C++ 中的函式庫－－ pbds(policy-based data structure)，俗稱平板電視。他跟 STL 有一點類似，但是他其中的某些功能是 STL 無法達到的。大家應該都知道 STL 在競賽中的重要性，因此 pbds 也絕對是幫助你邁向程式設計競賽成功的好幫手！話不多說，我們就趕快來看看怎麼用 pbds 吧！

1.2 pbds 介紹

接下來就向各位介紹 pbds 的用法及其中的資料結構。首先，無論是使用 pbds 中的人和資料結構，都需要打上這兩行：

```
#include <ext/pb_ds/detail/standard_policies.hpp>
using namespace __gnu_pbds;//there are two _ before gnu, but only one _ bet
```

以下就開始正式開始介紹 pbds 中的資料結構囉！

Hash Table

就跟 STL 中的unordered_map幾乎一樣 (可用中括號隨機存取、insert、delete 、clear、find、查看 size、自定 hash function，但不能count)，它分成gp_hash_table和 cc_hash_table，使用並無差異，只是處理碰撞的方式不同。
而使用時還要加上：#include<ext/pb_ds/assoc_container.hpp>。讓我們直接來看一下範例程式碼吧！

```
#include<bits/stdc++.h>
#include<ext/pb_ds/detail/standard_policies.hpp>
#include<ext/pb_ds/assoc_container.hpp>
using namespace std;
using namespace __gnu_pbds;
int main()
```

```
{
  gp_hash_table<int,int> r;
  for(int i = 0;i < 5000000;i++)
  {
    r.insert({i,i+1});
  }
  cout << (r.find(7))->second << endl;
  for(int i = 0;i < 5000000;i++)
  {
    r.erase(i);
  }
  if(r.find(7) == r.end())
  {
    cout << "No!" << endl;
  }
}
```

之後我又將cc_hash_table換成gp_hash_table和unordered_map。跑一樣的程式碼，cc_hash_table跑 2.42 秒左右，gp_hash_table跑 2.53 秒左右，unordered_map則跑了 5.06 秒。因此若覺得unordered_map常數太大的人可以考慮用用看 pbds 的 hash table 喔！

Priority Queue

對，這也跟 STL 的priority_queue差不多，STL 的priority_queue支援的操作它都支援，而為了防止與 STL 中的priority_queue搞混，在使用上要特別注意在宣告的時候要加上__gnu_pbds::，以確保用到的是 pbds 中的priority_queue。宣告方式也有些微差異，可在底下範例中看到。相較於 STL 的priority_queue只有一種，pbds 的priority_queue可以在宣告時加上一個 tag，以決定priority_queue的實作方式。使用不同的實作方式也就會導致不同操作有不同複雜度，簡單整理如下：

	push	pop	modify	erase	join
std::priority_queue	最差 $\Theta(n)$ 均攤 $\Theta(\log n)$	最差 $\Theta(\log n)$	最差 $\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$

其中的 pairing heap 和 binomial heap 較常用到。或許會有人好奇，要怎麼修改priority_queue內部的元素呢？首先，pbds 的priority_queue的 iterator 宣告如下：

```
__gnu_pbds::priority_queue<int>::point_iterator it;
```

此外，將元素push到 pbds 的priority_queue的時候會有回傳值，即為該元素所在的 iterator，只要在push的時候紀錄一下，就可以利用modify函式和 iterator 來進行修改囉！
使用時別忘了加#include<ext/ pb_ds/ priority_queue.hpp>。給個範例程式：

```cpp
#include<bits/stdc++.h>
#include<ext/pb_ds/detail/standard_policies.hpp>
#include<ext/pb_ds/priority_queue.hpp>
using namespace std;
using namespace __gnu_pbds;
int main()
{
  __gnu_pbds::priority_queue<int,less<int>,binomial_heap_tag> pq;//型態 比較
  __gnu_pbds::priority_queue<int,less<int>,binomial_heap_tag>::point_iterat
  for(int i = 0;i < 10;i++)
  {
    it[i] = pq.push(i);
  }
  pq.modify(it[0],7122);
  cout << pq.top() <<endl;
}
```

有沒有覺得寫 dijkstra 的時候會派上用場呢？

接著示範合併 (join)，注意被合併的priority_queue的大小會歸零。

```cpp
#include<bits/stdc++.h>
#include<ext/pb_ds/detail/standard_policies.hpp>
#include<ext/pb_ds/priority_queue.hpp>
using namespace std;
using namespace __gnu_pbds;
int main()
{
  __gnu_pbds::priority_queue<int,less<int>,pairing_heap_tag> a,b;
  for(int i = 0;i < 5;i++)
  {
    a.push(i);
    b.push(i+5);
  }
  a.join(b);
  cout << a.size() << " " << b.size() <<endl;
  cout << a.top() <<endl;
}
```

tree

接下來講的就是大家常常抱怨的set和map的進化版了啊，就是 pbds 中的平衡二元樹。它除了insert、erase、lower_bound、upper_bound等set已經有的操作之外，還可以找 k 小值 (find_by_order，回傳 iterator(0-based)) 和求排名 (order_of_key，回傳一整數 (0-based))，還支援類似 merge-split treap 的join(join的時候兩棵樹的類型要相同，沒重複元素，且 key 值要左小右大) 還有split。

它的宣告長這樣：

```
tree<int,null_type,less<int>,rb_tree_tag,tree_order_statistics_node_update>
```

前三格就是 STL map中的 key、對應的資料型態，和比較函式。第四格是各種樹的 tag，有rb_tree_tag、splay_tree_tag、ov_tree_tag，其中的rb_tree_tag較為常用。其他兩個的用處我不是很了解，但它在一般使用上的常數較高，盡量避免使用。最後就是更新方式了，不知道要打甚麼就照抄吧！它其實還有一些更進階的用法，像是自行定義點的更新方式，不過我也不會，而且蛋餅說還不如自己刻一顆 treap，所以這邊我就不多說大家有興趣的自行研究喔！ 也記得要加上#include<ext/ pb_ds/ assoc_container.hpp>。嗯，一樣來看範例程式。

```
#include<iostream>
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/tree_policy.hpp>
using namespace std;
using namespace __gnu_pbds;
int main()
{
  tree<int,null_type,less<int>,rb_tree_tag,tree_order_statistics_node_updat
  for(int i = 0;i < 10;i++)
  {
    a.insert(i*2);
  }
  a.erase(18); // tr = {0,2,4,6,8,10,12,14,16}
  cout << *a.lower_bound(11) << endl; // 原本set和map支援的操作都有
  cout << *a.find_by_order(0) << endl;
  cout << *a.find_by_order(5) << endl;
  cout << a.order_of_key(9) << endl;
  cout << a.order_of_key(10) << endl;
  a.split(6,b);
  for(auto i : a)
  {
    cout << i << " ";
  }
  cout << endl;
  for(auto i : b)
  {
    cout << i << " ";
  }
  cout << endl;
  a.join(b);
  cout << a.size() << endl;
}
```

Trie

大家可能會覺得剛剛的資料結構或許很厲害，但其實都只是 STL 之中不同資料結構的變形。但接下來要介紹的資料結構，STL 之中就沒有了呢！它，就是大名鼎鼎的 Trie！

首先先介紹宣告方式，它的宣告方式比較複雜，我盡量想辦法簡單一點。它完整的宣告長這樣：

```
1 trie<string,null_type, trie_string_access_traits<>, pat_trie_tag, trie_pref
```

第一個參數一定要是字串型態，第二格可以是其他的資料型態 (類似map的感覺)，不用的話就用null_type，第三格是比較函式，第四格是 trie 的種類，最後則是更新方式。一般使用時後三格可以照打，有興趣者可自行研究。那你可以對它做啥操作呢？

首先就是最基本的insert、delete、find 字串，也可以進行合併 (join) 和分割 (split)(要注意的是，join的兩個 trie 一定要第一個字典序比較小，第二個比較大)。它還有一個很特別的操作，就是prefix_range的這個操作，它會回傳一個 pair，代表以某個前綴開頭的字串的開頭的 iterator 和結尾的 iterator。這樣講不太清楚，不如直接看範例吧！要記得加上#include<ext/pb_ds/assoc_container.hpp>。

```
1  #include <iostream>
2  #include <ext/pb_ds/assoc_container.hpp>
3  #include <ext/pb_ds/detail/standard_policies.hpp>
4  using namespace std;
5  using namespace __gnu_pbds;
6  int main()
7  {
8    trie<string, null_type, trie_string_access_traits<>,pat_trie_tag,trie_pre
9    a.insert("her");
10   a.insert("hers");
11   b.insert("she");
12   b.insert("his");
13   a.join(b);
14   auto temp = a.prefix_range("h");
15   for(auto i = temp.first;i != temp.second;i++)
16   {
17     cout << *i << " ";
18   }
19   cout <<endl;
20   a.delete("hers");
21   a.split("r",b);
22   cout << a.size() << " " << b.size() <<endl;
23 }
```

1.3 習題

其實 pbds 只是一個方便使用的工具，裡面很多的資料結構都可以自己手刻出來，因此我也不知道要放甚麼習題呢！大家好好思考下面的題目吧！

習題 1.3.1: 忍者調度問題 (TIOJ4544 1429)

給定一個 N 個節點的有根樹，每個節點有兩個值 (C 和 L)，再給一個整數 M，你要找到任意個節點，使的他們 C 值的和小於 M，且選定的節點數乘上所有選定節點的某個共同祖先的 L 值最大，輸出該最大值。($N \leq 10^5$，$M \leq 10^9$)

LCA 和他們的產地 2

2.1 前言

LCA（Lowest Common Ancestor），最低共同祖先，到底是甚麼呢？當你把一堆生物的親緣關係樹畫出來的時候，你會很想要知道兩種生物的親緣關係接不接近，而最直接的想法就是找出他們的共同祖先嘛！但是他們可能有很多共同祖先，要找哪個好呢？如果找的是很早很早的祖先，那所有的生物都是他的後代，意義並不大，要找的話就是找出他們最低的共同祖先，也就是 LCA，才會最有意義。我們可以說，LCA 越低的兩個生物，親緣關係越近。

那 LCA 在 CP 中有甚麼用武之地呢 ?? 嗯，他跟上面所舉的例子一樣，專門用來處理有關於樹的題目，但處理樹的工具千奇百怪，包括樹鍊剖分啦、重心剖分啦、樹分治、LCT（Link Cut Tree）啦、ETT（Euler Tour Tree）啦、樹壓平啦，LCA 可以說是比較基礎的一個主題。而 LCA 的作法也是各式各樣，接下來都會介紹，而且每個都有每個的好處，千萬不要因為會了其中一個，就對其他的不屑一顧喔！

2.2 暴力做 LCA

如果要暴力去做 LCA，相信大家都會：每次詢問兩個節點 A 和 B 的 LCA 時，紀錄從根到 A、B 的路徑成兩個序列，然後找到最前面的節點使得序列不同，這個的前一個就是了（如果都一樣就是根了）。這樣子每次詢問 DFS 兩次 $O(N)$，然後找尋 LCA，$O(N)$ 掃過去即可。

另外一個想法就是：DFS 一次紀錄每一個點的父節點和深度，然後每次詢問的時候，不斷把深度小的（一樣就隨便）往上走一次，直到走到一樣為止，這樣複雜度進步為 $O(N)$。每次處理都紀錄所有算到的東東就可以大大大減少其複雜度，具體的實作方法：

$$\mathrm{LCA}(x,y)=\begin{cases} x, & \text{for } x=y \\ x, & \text{for } \mathrm{parent}[y]=x \\ y, & \text{for } \mathrm{parent}[x]=y \\ \mathrm{LCA}(\mathrm{parent}[y],\,x), & \text{for } \mathrm{depth}[x]\le\mathrm{depth}[y] \\ \mathrm{LCA}(\mathrm{parent}[x],\,y), & \text{for } \mathrm{depth}[y]<\mathrm{depth}[x] \end{cases}$$

當然，暴力永遠（大部分的時候）不是答案，所以在這裡要介紹各種常見找 LCA 的方法。

2.3 樹壓平取 LCA

RMQ 問題

在講 LCA 之前，首先看個區間的問題：

習題 2.3.1: RMQ（區間極值）

給定 N 個數字 $a_1, a_2, a_3, \cdots, a_N$，和 Q 個詢問 $[l_i, r_i]$，請對於每一個詢問輸出 $[l_i, r_i]$ 內的最大值。

相信大家都有辦法在不裝弱的前提下容易的寫出 $O(NQ)$ 的解答（寫不出來可能要去面壁思過了），所以通常題目也不會出那麼簡單。這裏具體實作方法不寫（請洽其他章節 !），不過用 Sparse Table 或線段樹等資料結構可以變成 $O(N + Q \log N)$ 或 $O(N \log N + Q)$。

歐拉遍歷（Euler Tour）

要如何把一個樹變成一個序列呢？一個簡單的想法就是 DFS，然後遇到每一個點進去和離開的時候都push_back進去。這個序列會有 $1 + 2(N - 1) = 2N - 1$（根先進去，然後每一個邊都加兩次）個元素，並且紀錄每一個點進來和出去的時間戳。這個有什麼性質呢？可以發現，如果 u 是 v 的祖先，則 u 會先放進去，然後 v 進來出來了，u 才會放第二次。除此之外，還需要紀錄一些額外的東西，待會一一介紹。所以呢，**對於一個點，其所有子節點的子樹在序列的位置會被包在那個點在序列中的位置兩邊。**如果感覺有一點抽象，那就看個例子吧！(Source: GeeksForGeeks)

會需要維護四個陣列：

1. 在序列裡面每一個節點的深度 $\mathrm{dep}[x]$
2. 原本的歐拉遍歷序列 $\mathrm{S}[x]$
3. 每一個節點在序列中第一次出現的位置 $\mathrm{first}[x]$（$= \arg\min_i(\mathrm{S}[i] = x)$）
4. 在遍歷序列中，每一個節點所對應到的深度序列 $\mathrm{SDep}[x]$（$\mathrm{SDep}[i] = [\mathrm{S}[i]]$）

當每次要詢問 a 和 b 的 LCA 時，找到 SDep 內的區間 $(\mathrm{first}[a], \mathrm{first}[b])$ 的最小深度所在，假設在 x，則 $\mathrm{S}[x]$ 為答案。用以上的圖解釋，如果想要找 4 和 9 的 LCA，則發現 $\mathrm{first}[3] = 2$（第一次出現在位置為 2 的地方，在 1，2 前面）、$\mathrm{first}[9] = 7$。則可以發現，在 SDep 中（圖中下面的序列），位置為 3 的有最小深度 1，所以回去看 S 中位置為 3 的是誰，發現是 2（1，2，4，**2**），所以回傳 2。

為什麼這樣會對？跑的夠快嗎？

當想要計算 a 和 b ($\text{first}[a] < \text{first}[b]$) 的 LCA 時，從 $\text{S}[\text{first}[a], \cdots, \text{first}[b]]$ 的序列是進入 a 了，然後出去 b 了，然後往上遞迴，每到他的一個祖先就會遞迴到那個祖先的所有子樹，直到進入了 b 為止。顯然，一旦回溯到了 $\text{LCA}(a, b)$，即會遞迴到 b，而到了區間的結尾。所以呢，$\text{LCA}(a, b)$ 在區間裡面，而且它一定是深度最下面的那一個。

至於複雜度，第一次 DFS 的複雜度是 $O(V)$，而如果用線段樹或 Sparse Table 的話，複雜度差不多是 $O(V \log V)$。

2.4 Doubling 倍增法

Doubling，顧名思義，就是倍增法，聽到倍增法，就可以知道跟二的冪次有關。

Doubling 可以說是 LCA 的各種做法中最受高中競賽選手歡迎的一種做法。它的好處是 code 短好刻，又可以在求 LCA 的時候順便紀錄某一些性質，以便做到某些神奇的事。

話不多說，就直接講做法囉！首先介紹預處理的部分。預處理非常的簡單明瞭，只要記錄兩個簡單的東西：節點的深度，還有每個節點的第 2^k 輩祖先。點的深度相信大家都會，我也不多費唇舌了。至於每個節點的第 2^k 輩祖先，則可以用以下的 DP 做法簡單求得：

```
anc[root][0] = -1; //根的祖先不存在！
anc[x][0] = pa[0];//第一輩祖先就是父節點
for(int i = 0;i + 1 < MaxLog; i++)
{
    anc[x][i+1] = anc[anc[x][i]][i]; //anc[i][j]表示第i個節點的第$2^j $輩祖
}
```

這樣子看來，預處理的複雜度是 $O(N \log N)$(N 是節點數)。

那詢問要如何回答呢？

第一步，想辦法讓兩個要求 LCA 的節點等高。具體做法就是找出兩者的深度差 (稱為 Δh)，並讓較低的節點慢慢上升（每次提升 Δh 的二進位中為一的位數所代表的二的冪次，也就是每次都盡量移動多），複雜度 $O(\log N)$。Code 長這樣：

```
if(dep[a] > dep[b])
    swap(a,b);
int k  = 0;
for(int i = dep[b]-dep[a]; i > 0; i/=2)
{
    if(i % 2 == 1)
        b = anc[b][k];
    k++;
}
```

接下來，基本上就是二分搜了，我們可以知道，若兩個節點深度相同，要找出兩節點的 LCA 等於找出一個 t，使兩者的第 t 輩祖先恰好相同，我們二分搜的目標

就是這個 t。更詳細地說，就是從大的二的冪次開始，若兩者的第 2^i 祖先不同，就將兩者轉換為兩者原先的第 2^i 祖先然後繼續。

```
for(int i = MaxLog-1;i >= 0;i--)
{
    if(anc[a][i] != anc[b][i])
    {
        a = anc[a][i];
        b = anc[b][i];
    }
}
```

最後的 LCA 就是 anc[a][0]。

一開始有說到可以在倍增法的時候紀錄某些性質，是甚麼意思呢？讓我們直接來看看一個例題：

習題 2.4.1: 經典題目

給你一個有 N 個節點的樹，其中每條邊都有邊權，並有 Q 筆詢問，每次詢問問你兩個點之間的路徑上所有邊權的最大值。$(N, Q \leq 10^5)$

我看到題目的第一個想法就是紀錄從跟開始到每個節點的路徑上的邊權的大值。可是稍微想了一想之後，就會發現這樣做好像沒辦法解決問題（因為 max 沒辦法好好合併）！

接著，可能就要請出 LCA 了，能否用 doubling 呢？然後你就會發現真的可以喔！具體做法就是在預處理的時候順便紀錄每個節點走到他 2^k 輩祖先所經過的路徑中的邊權最大值，之後在處理詢問的時候，就可以在向上跳躍的過程中順便找出跳躍過的這一大堆路徑最大值中的最大值囉！

2.5 Tarjan 的離線 LCA 演算法

演算法敘述

可能聰明的讀者有聽說過 Tarjan 這個人的名字！他對演算法學的貢獻不只在這裏，他有更為人知的求割點的演算法，但是這裏要介紹的是他的離線 LCA 求法。顧名思義，他的演算法雖然很快（幾乎線性 !），但是必須離線，也就是需要事先知道需要查詢哪些點對。

他的作法運用到了並查集（Disjoint Sets），所以會用到其基本的操作（你們應該會並查集，放在這裡只是複習用）：

定理 2.5.1: 並查集所支持的操作

1. Find(x)：找到 x 的代表
2. Union(x, y) ：將 x、y 合併成為同一個集合

那假設想要詢問的 LCA 是 $Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_n, v_n)\}$，那進行以下演算法：從根開始先 DFS 一次，DFS 完一個子節點即將子節點的代表設定為自己，而當遞迴完所有的

對於目前節點 u 的每一個子節點 v 先遞迴下去，然後將 v 的代表變成 u。然後將 u 的顏色（自己設定的值，類似visited）設定為 1，也就是黑色（一開始全部都是 0，白色）。可以發現，**一個節點為黑色若且唯若其所有的子樹都被遞迴完了**。

兩個奇怪的操作

可是，為什麼要做設定代表和判斷顏色這兩個奇怪的操作呢？首先，設定 v 的代表的意思就是：每搜到一個 v 的祖先 x，則會將Find(v)設定為 x，代表搜到其他 x 的子樹內，如果有問到 v 而且沒有在之前就就回答到了，那個詢問的答案就是 x，而且 x 的子樹也都一樣，這個操作利用並查集剛好可以快速實作。

第二，顏色的問題就比較簡單了，就是想要比較晚的來問而已，晚來的會問早來的，而首次（也就是要回答的那一次）問的時候就是被他們的 LCA 遞迴的時候，此時 $dsu[$ 早來的 $]$ 就會是他們的 LCA。

程式碼

看了那麼多，可能你眼花撩亂；那就看簡短的程式碼理解吧（並查集的操作省略）！

```cpp
vector<vector<int> > children; //各個節點的子節點
vector<vector<int> > queries //各個詢問有那個節點的資料，Ex. 詢問(2, 3)會同
vector<int> dsu; //並查集
void tarjanLCA(int u){//u為目前的節點
    for(int v : children[u]){
        tarjanLCA(v);
        Union(u, v);
        anc[Find(u)] = u;
    }
    colour[u] = 1;
    for(int v : queries[u]){
        if(colour[v] == 1){
            Find(v);
            //LCA(u, v) = anc[Find(v)]
        }
    }
}
```

複雜度

這個在一次 DFS 即可完成，而且每一個 query 會被弄到兩次，所以會很接近線性（超線性）的複雜度（會多一個艾克曼的反函數，一般用途不會超過 4）的 $O((N+Q)\cdot\alpha(N))$，此處 $\alpha(N)$ 是反艾克曼函數。

2.6 樹鍊剖分

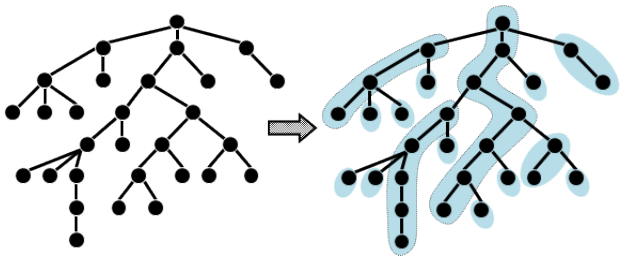
咦！樹鍊剖分，聽起來就很恐怖，到底是啥啊？不用怕，他真的沒有想像中難，他只不過是一種 LCA 的求法而已嘛！當然，他能做的事當然不只 LCA。

何謂樹鍊剖分

樹鍊剖分，又稱輕重鍊剖分 (Heavy-Light Decomposition)，也是用來處理樹上路徑相關問題的好工具。他是一種把序列上的操作搞到樹上路徑的方法，線段樹可以對序列做的事，他幾乎都可以把它搬到樹上來做 (路徑查詢啦、點權更改等等)，真的非常有趣喔！

定義 2.6.1: 重鍊定義

對於一棵樹的每一個節點 u，令其子樹的大小為 $\mathrm{sz}(u)$，則若存在一個 u 的子節點 v 使得 $\frac{\mathrm{sz}(u)}{2} < \mathrm{sz}(v)$，那 u 和 v 就存在一條**重邊**（圖中粗邊為重邊，細邊為輕邊）。



由圖看出，那些重邊會形成一條鍊，而那些鍊就是所謂的**重鍊**！

直接講樹鍊剖分沒什麼意思，讓我們直接來看例題吧！

習題 2.6.1: UVa 12424

給你一棵 N 個節點的樹，每個節點有一種顏色，和 Q 個操作，每一個操作都是以下兩個的其中之一：

1. 將某個點的顏色改掉
2. 查詢兩個節點之間的簡單路徑哪個顏色出現最多次，並且輸出

$(N, Q \leq 10^5$，顏色數 $\leq 10)$

看到這個題目，不知道大家有甚麼想法呢？首先，套一句老話，如果對這個題目沒有想法，就先思考一下簡單的版本：

習題 2.6.2: 上一題的簡單辦

給你一個序列，你需要處理兩種操作：

1. 將某個位置的數字改掉
2. 查詢兩個位置之間哪種數字最多

(序列長度、操作數 $\leq 10^5$，數字種類 ≤ 10)

這個問題好像就不那麼難，只要對每一種顏色開一顆支援單點修改，區間找和的資料分結構 (BIT、線段樹等)，每次修改次單點修改，查詢就把 10 種顏色都區間查詢一遍就好了。

回到原本的題目，我們的目標就是**構造一個節點的序列，使得樹上的一條簡單路徑經過的節點分布在序列中盡量少個連續的區塊**。

若是把樹轉換成一個序列，最直覺的想法就是所謂的**樹壓平**，也就是**任意選定一個根後，依照樹的前序遍歷順序（也就是進入時間）將所有節點由小到大排成一個序列**。這樣做的好處就是，一個子樹會落在連續的一段序列中，所以只要知道是哪些序列，然後再對那些序列詢問，就可以化簡成上面的問題了！因此，若這個問題問的是每個子樹最多的顏色，那用樹壓平的方法可以在 $O(N + Q \log N)$ 的複雜度做完。

很可惜的，這題的主要重點是路徑而非子樹，所以不能用樹壓平解決。

通常，**把無根樹轉為有根樹會有助於解題**，而這麼做之後，兩點之間的簡單路徑就是其中之一向上走到兩者的 LCA，再向下走到另一個點。因為如此，LCA 在路徑問題中佔了蠻重要的地位。那這題可以用倍增法來解決嗎？利用倍增法的確可以在往上跳的過程中順便進行區間查詢，不過要怎麼把節點變成序列，使得在往上跳的過程中不經過太多區塊？因為每個節點都是往根的方向跳，所以很難構造出這個序列。因此，我們在這裡提供另外一種 LCA 的作法 (其實就是樹鍊剖分！)。

用樹鍊剖分解 LCA

首先，DFS 第一次找出每個子樹的大小，並對每一個節點找出他子節點中最大子樹是哪一棵，也順便紀錄每個節點的父節點和每個點的深度。

接著進行第二次 DFS。注意，我這裡直接把細節都說出來喔！從根節點開始，走向他最大的子樹，接下來繼續，直到走到葉為止，這就是第一條重鍊！走的時候，要順便紀錄每個點所屬的重鍊的最高的點，也可以同時為這些點重新編號（也就是第二次 DFS 的進入順序，紀錄了新的序列中每個節點在哪裏）。而找出第一條重鍊後，其他不在這條鍊中的點，也要從自己開始找重鍊。這樣說明可能不是很清楚，不過待會看 code 的時候或許可以體會到喔！為甚麼我們要將點重新編號呢？還記得我們的目標是將這顆樹轉換成一個序列嗎？這個新的編號，就是新序列的順序喔！我們終於拿到一個序列了，接著就對這個序列建構一棵線段樹 (或是其他資料結構) 吧！好，預處理到此可說是告一段落。我們做了那麼多事，到底是為了甚麼？先賣個關子，讓我們繼續看下去。我們終於要來處理 LCA 了！在介紹方法之前，我們先來看看會用到剛剛預處理的哪些東西：

1. 每個節點深度 dep[x]

2. 每個節點自己所在的重鍊中，深度最小的節點 link_top[x]
3. 每個原編號轉換成的新編號 link[x]
4. 每個節點的父節點 pa[x]

這部分看 code 蠻好了解的，我就借卦長的 code 來讓大家看看 XD！(Source：http://sunmoon-template.blogspot.com/2015/07/heavy-light-decomposition.html)

```
#include<vector>
#define MAXN 100005
int siz[MAXN],max_son[MAXN],pa[MAXN],dep[MAXN];
/*節點大小、節點大小最大的孩子、父母節點、深度*/
int link_top[MAXN],link[MAXN],cnt;
/*每個點所在鏈的鏈頭、樹鏈剖分的DFS序、時間戳*/
std::vector<int >G[MAXN];/*用vector存樹*/
void find_max_son(int x){
    siz[x]=1;
    max_son[x]=-1;
    for(auto i : G[x]){
        if( i== pa[x] )continue;
        pa[i] = x;
        dep[i] = dep[x]+1;
        find_max_son(*i);
        if(max_son[x]==-1 || siz[i]>siz[max_son[x]] ) max_son[x] =
i;
        siz[x]+= siz[i];
    }
}

void build_link(int x,int top){
    link[x] = ++cnt;/*記錄x點的時間戳*/
    link_top[x]=top;
    if(max_son[x] == -1)return;
    build_link(max_son[x], top);/*優先走訪最大孩子*/
    for(auto i : G[x]){
        if( i==max_son[x] || i == pa[x])continue;
        build_link(i,i);
    }
}
```

接下來呢？其實很簡單，要找兩點之間的 LCA 時，首先他判斷這兩點是不是在在同一條重鍊上，如果是的話，那頭的深度小的那點就是 LCA 囉 XD。不是的話怎麼辦？我們可以發現，**如果把深度比較低的那個點換成它那條重鍊的深度最小的節點的父節點之後，兩者的 LCA 並不會改變** (因為他跳過的部分絕不可能是 LCA)。因此就這樣一直換，直到兩者在同一條重鍊中囉！

```
int find_lca(int a,int b){
    /*求LCA，可以在過程中對區間進行處理*/
    int ta = link_top[a], tb = link_top[b];
    while(ta!=tb){
```

```
        if(dep[ta] < dep[tb]){
            swap(ta, tb);
            swap(a, b);
        }
        //這裡可以對a所在的鏈做區間處理
        //區間為(link[ta], link[a])
        ta = link_top[ a=pa[ta] ];
    }
    /*最後a,b會在同一條鏈，若a!=b還要在進行一次區間處理*/
    return dep[a] < dep[b] ? a : b;
}
```

Code 中的註解應該也寫得頗為清楚，在將低的節點往上跳的同時，可以順便進行區間操作，因此，我們的 UVa 12424 這題可以說是得到了大致的解法。剩下的問題就是，這麼做的複雜度有比較好嗎？

樹鍊剖分的複雜度分析

已知每次進行區間操作的複雜度是 $O(\log N)$，那對於每一組要求 LCA 的點對，總共要進行幾次「跳躍」呢？這裡我們考慮從一條重鍊上的某一個點跳到他所處的重鍊的深度最小的節點 (稱作 h) 的父節點 (稱作 p) 的時候其子樹的大小，因為 h 的子樹並非 p 的子節點中的最大子樹 (否則他們會在同一條重鍊中)，因此 p 必有一個子節點的子樹的大小比 h 的子樹要來的大，因此，每跳躍一次，跳躍後的節點的子樹都會是原本的兩倍以上。就這個角度來看，一個節點總共最多也只能跳 $\log N$ 次，總共有兩個節點，因此跳躍的次數為 $O(\log N)$。最終，我們可以了解到每次詢問的總複雜度是 $O(\log N)$(跳躍次數)$\times$ $O(\log N)$(區間操作複雜度)！因此，以上的題目可以在總複雜度 $O(N + Q\log^2 N)$ 做完。以下附上程式碼。

跟著蕭電這樣做

要注意的是，樹鍊剖分可以套上其他各式各樣資料結構，包括 Treap、BIT、線段樹，還有其他持久化資料結構。

2.7 習題

仔細思考這些題目有那裏要用到 LCA，都是比較難的題目！

習題 2.7.1: UVa 11354 Bond （TIOJ 1163）

非常厲害的 Ovuvuevuevue Enyetuenwuevue Ugbemugbem Osas（簡稱 Osas）要環遊非洲！他選了 N 個城市，編號為 $1, 2, 3, \ldots, N$，並且他的地圖上寫了其中一些城市間有 M 條雙向的路，長度為 d_i。他要走 Q 次，每一次想要從 s_i 走到 t_i。對於一個從 s_i 走到 t_i 的走法，Osas 的疲累度就是路徑上經過路中最長的那一個。請問，對於每一個 s_i 和 t_i，他可以走的最小疲累度為何（也就是對於每一個 s_i 到 t_i 的路徑中，最大路長最小為何）？($1 \le N \le 50000$，$1 \le M \le 10^5$)

習題 2.7.2: TIOJ 1798 Can You Arrive？

Osas 現在有了困境——到了先進的新加坡（他非洲逛完了），想要搭地鐵前往美食中心，Newton Hawker！但是，他有一個困擾：新加坡有 N 個站，其中被 $N-1$ 條邊連著（恰好是一棵樹！），但是只有 K 種車子行駛，第 i 種車子會在 x_i 到 y_i 之間的唯一路徑往返（因為是樹！），而他現在有 Q 個詢問，請問從編號為 a 的站可不可以經過若干次轉車到達編號為 b 的站？($K \leq N \leq 10^6$，$Q \leq 10^6$)

習題 2.7.3: TIOJ 1445 機器人組裝大賽

現在的 Osas 已經今非昔比，是個科技的人才！他現在參加了一個機器人組裝比賽，其中有 N 個零件，有 M 種方法可以連結它們，第 i 種方法可以連結第 a_i 和 b_i 個零件，代價有 w_i。但是：他發現比賽場地也有一個勁敵，名稱為 Kkwazzawazzakkwaquikkwalaquaza '* Zzabolazza（簡稱 Zzabolazza），他已經找到一個最佳的方法連結所有的零件了（也就是代價最小），請幫助 Osas，找到一個代價盡量小的零件連接方法（一定要全部連！不行的話輸出-1），並且與 Zzabolazza 的連接方法不完全相同）。

離線處理淺談 - 強大的工具 3

如果一個演算法在計算答案之前不需要知道所有輸入，我們稱這個演算法為**在線**演算法；反之若在計算答案的時候要先知道所有輸入，我們稱此為**離線**演算法。例如插入排序在排序所有數字時可以一個一個讀進來再各自插入，選擇排序則是必須先讀進所有數字後，才能按照順序把最小的數字放到前面。

犧牲了即時更新性所換來的通常是計算量或 coding 複雜度的減少，我們先以 RMQ 當作一個簡單的例子。

習題 3.0.1: 經典題 - RMQ (ZJ d539)

給定一個序列 $a_1, a_2, \ldots, a_n$，之後有 q 次詢問 $[l, r]$ 中的最大值。$n, q \leq 10^5$。

我們可以讀進所有詢問 $[l, r]$，並依照左界由大到小排序。如此一來，對於固定的左界只需要維護一個前綴 max 就足以回答所有詢問，我們可以想到用 BIT 來維護這個東西，對於左界的遞減剛好對應到單點更新，就這樣輕鬆完成不須線段樹的 RMQ 啦！雖然這個例子可能不一定能感受到離線演算法的威力，但大家應該了解到了，如果離線計算的話可以用很多奇怪的方法去減輕時間、空間或 coding 複雜度。

3.1 莫隊 (Mo's Algorithm)

從前面的題目可以發現，在面對一些區間詢問的題目時，有時我們可以藉由將詢問以某種方式排序來降低操作整體的複雜度，其中有一種演算法稱為**莫隊**。

莫隊演算法通常適用於那些當區間左右界稍微改變時，能以很小的代價維護新區間的題目，簡單來說，我們要想辦法排序詢問使得兩兩之間的「距離」都很相近，讓我們可以從前一個詢問的答案用不多的時間推出後一個詢問的答案以節省時間複雜度。如果能花 $F(n)$ 的時間從 $[l, r)$ 的答案推到 $[l \pm 1, r)$ 或 $[l, r \pm 1)$，由 $[l_1, r_1)$ 的答案推出 $[l_2, r_2)$ 的答案所需要的時間就是 $F(n) \cdot (|l_2 - l_1| + |r_2 - r_1|)$，這也可以看成座標平面上兩點的曼哈頓距離，我們可以用曼哈頓最小生成樹的演算法達到最差 $O(F(n) \cdot n\sqrt{q})$ 的複雜度，但是否有更簡單的方法能達到相同的複雜度呢？

我們可以想到先將詢問依照左界排序，這樣當我們要查詢時左界只需要修改最多 $O(n)$，不過聰明的大家應該會發現右界的順序是亂的！這就會導致我們的修改次數最多會退化至 $O(nq)$，為了解決這個問題，我們決定也以某種方式對右界排序。

我們把原序列分成 m 塊 (每塊有 n/m 個元素)，並且依照左界所在的塊排序，若在同一塊則以右界排序。左界在同一塊間的移動次數總和最多就是 $q \cdot n/m$，而不同塊之間的移動次數最多是 $m \cdot 2(n/m) = 2n$；右界在同一塊間的移動次數總和最多是 nm ，而不同塊之間右界的移動次數最多也同樣是 nm；因此總複雜度為 $O(F(n) \cdot (nq/m + nm + 2n))$，可以發現取 $m = \sqrt{q}$ 會得到最佳複雜度 $O(F(n) \cdot n\sqrt{q})$ ，這也和曼哈頓最小生成樹可得到的結果相同，我們通常採用這種寫法。莫濤提出的莫隊算法，除了將詢問排序之外，剩下都只是暴力計算，但是就是因為把詢問好好排序，得以把複雜度降低了一個根號。

實作上有一些細節，就是在更新左、右界時，盡量不要讓右界比左界小（也就是不要讓區間不合理），這部分的程式碼通常會用數個while來實現，其架構大概長的像下面這樣。

```
struct Query{
    int l, r, id, block;
} Q[MAXQ];

bool cmp(Query &a, Query &b){
    return (a.block!=b.block) ? a.block<b.block : a.r<b.r;
}

int n, m, res, ans[MAXQ];
void add(int pos){ ... } // 維護增加一個數的改變
void sub(int pos){ ... } // 維護減少一個數的改變

void MO(){
    int K = 400; // 有時會依範圍直接寫一個固定的大小
    for(int i = 0; i < q; i++) Q[i].block = Q[i].l/K;
    sort(Q, Q+q, cmp); // 有時沒排序也沒過(X
    // 初始化也很重要，要注意一開始的區間
    int l = 1, r = 0;
    for(int i = 0; i < q; i++){
        // 先擴展再縮減
        // 這邊是左閉右閉的區間寫法，邊界問題自己要注意
        while(l > Q[i].l) add(--l);
        while(r < Q[i].r) add(++r);
        while(l < Q[i].l) sub(l++);
        while(r > Q[i].r) sub(r--);
        ans[Q[i].id] = res;
    }
    for(int i = 0; i < q; i++) cout << ans[i] << '\n';
}
```

雖然莫隊通常是用來解決不帶修改的題目，但大陸人也有研發出可以解決待修改問題的莫隊，想法大概是把時間視為第三個維度，變成三維曼哈頓距離的感覺 (前提是要能在時間軸上前後移動)，用類似的方法可以有 $O(n^{\frac{5}{3}})$ 的複雜度。

3.2 操作分治

操作分治，又名 CDQ 分治。既然名字裡面有個分治，顧名思義就是利用了 Divide & Conquer 的算法。通常我們都希望問題的維度越低越好，但操作分治的做法卻是**加上了一個時間維度，對時間分治，並保持其中一個維度有序**，使問題的維度由 n 到 $n+1$ 再降到 $n-1$。

儘管這個想法聽起來莫名其妙，但在處理帶修改的可離線問題時非常有效。尤其是原本是帶修改的二維問題（加上時間變成三維），可能可以透過操作分治降低複雜度。

實際上操作分治可以分為三個步驟:

- 計算左邊區間的答案
- 計算右邊區間不受左邊區間影響時的答案
- 計算左邊區間對右邊區間的影響

注意這邊是對時間分治，所以左邊右邊代表了時間上的先後；而以上三個步驟的順序也可能會依照題目改變。直接上例題吧！

習題 3.2.1: 經典題 (No judge)

一個二維平面上，有 n 次操作，每次操作可以在一個點上加上權重，或者詢問某個座標的左下角所有權重的和。$n \leq 10^5, |x|, |y| \leq 10^9$。

離散化是不可避免的，但就算離散化後用二維 BIT 或線段樹依然會 MLE，我們嘗試用操作分治降低問題的維度。

首先依照對每個操作加上一個時間維度，並直接對其分治。對每個區間來說，左右兩個區間的答案會變為兩個子問題，因此我們需要處理的只有「左邊區間的修改」對「右邊區間的查詢」的影響。可以發現，如果按照 x 座標排序好，我們就可以直接動態維護 y 座標前綴和 (BIT) 以得到答案；實作上有點類似 merge sort 中 merge 的部分，下面的 code 大概顯示了如何運作。

```
Query Q[MAXQ];
void merges(int L,int M,int R){
    int i = L, j = M;
    vector<query> tmp;
    init(); // BIT
    while(i<M || j<R){
        if((j==R) || ((i<M&&Q[i].x<Q[j].x){
            if(Q[i].type == ADD)
                add(Q[i].y,1); // BIT
            tmp.push_back(Q[i++]);
        }else{
            if(Q[j]].type == QRY)
                ans[Q[j].id] += query(Q[j].y); // BIT
            tmp.push_back(Q[j++]);
        }
    }
    for(int i=0;i<R-L;i++)
```

```
18          Q[i+L] = tmp[i];
19  }
20  void CDQ(int l=0, int r=n){
21      if(r-l == 1) return;
22      int mid = l+(r-l)/2;
23      CDQ(l,mid),CDQ(mid,r);
24      merges(l,mid,r);
25  }
```

每次把一個區間 $[l,r)$ 的兩部份以 BIT 合併求解的複雜度是 $O((r-l)\log n)$ (在離散化的前提下)，總時間複雜度 $T(n)=2T(n/2)+O(n\log n)$ ，由主定理得知 $T(n)=O(n\log^2 n)$。

雖然這題可能也可以用動態開點四叉樹或其他奇怪的資料結構解決，但操作分治寫法在空間、coding 複雜度都表現得很好，並且時間複雜度的常數也很小。

3.3 整體二分搜

讀到這裡的同學應該都了解什麼是二分搜了吧！二分搜的檢查有時會有些代價，而如果直接各自二分搜這些代價加起來會使複雜度爛掉，我們想辦法讓這些代價能夠共用而減輕複雜度，因此就誕生了**整體二分搜**的演算法，顧名思義就是整體一起進行二分搜。

習題 3.3.1: 經典題 - 靜態區間第 k 小

給定一個序列 $a_1,a_2,\ldots,a_n$，之後有 q 個詢問 (l,r,k) ，請對每個詢問輸出 $[l,r]$ 第 k 小的數字。

區間第 k 大可能在嵌套或持久化資料結構也會提到，但利用持久化線段樹的寫法空間複雜度很高（筆者寫了兩天還一直 MLE，QAQ），也不是我們今天的主題。

類似於lower_bound，考慮對答案二分搜：每次詢問一個 x 並檢查每個詢問區間有幾個數不大於 x ，如此可以分成答案要比 x 大及答案要比 x 小的兩種詢問；注意原序列的數也可以分成比 x 大及比 x 小的兩種，因此能把原序列也分成兩部分，變為兩個子問題遞迴解決。

至於我們要怎麼知道每個區間有多少數不大於 x 呢？我們可以用 BIT 維護「$\le x$ 的數的個數」的前綴和，如此對於右半部遞迴的前綴和只需要以原序列中大於 x 的數修改前半部遞迴時的 BIT 即可，省下每次都重算原序列或重置 BIT 的時間複雜度。整體二分的核心程式碼大概長的像這樣 (參考自日月卦長的部落格)，當然不同題目一定會需要修改細節部分。

```
1  // 原序列的數可視為操作一起放進陣列
2  // V,L,R 是操作們的索引，傳遞 int 比傳遞物件效率好
3  void totalBinarySearch(int l,int r,vector<int> &V){
4      int mid = l+(r-l>>1);
5      vector<int> L,R;
6      // 將V中對應的原序列數及詢問以mid為界分到L,R
7      split(V,L,R,mid);
8      // 二分搜答案不大於 mid 的
```

```
9       totBS(l,mid,L);
10      // 在BIT中記錄不大於 mid 的數造成的影響
11      update(L);
12      // 二分搜答案大於 mid 的
13      totBS(mid+1,R);
14      // 復原操作，以免影響到之後的搜索
15      undo_update(L);
16  }
17  // split的實作，利用BIT
18  void split(vector<int> &V,vector<int> &L,vector<int> &R,    int x){
19      for(int id:V){
20          if(...){ //是原序列數a_i
21              if(a_i < x) {
22                  add(i,1); // BIT
23                  L.push_back(id);
24              }else R.push_back(id);
25          }
26      for(int id:V){
27          if(...){ // 是詢問(l,r,k)
28              int cnt = query(r)-query(l-1); // BIT
29              if(cnt < x) L.push_back(id);
30              else R.push_back(id);
31          }
32      }
33      // 離開時記得 undo 讓 BIT 和進來前相同
34      undo_update(L);
35      V.clear(); // 節省空間
36  }
```

我們來簡單分析一下整體二分搜的複雜度：原序列和詢問都在遞迴樹中出現，每個原序列的數以及每一個詢問出現的次數都是樹高 $O(\log(n+q))$ ，也是 BIT 修改及查詢的時間複雜度，而每次 BIT 修改最多 $O(\log n)$（在離散化的前提下），因此我們的均攤時間複雜度是 $O((n+q)\log n\log(n+q))$。整體二分搜的一個明顯優點是空間複雜度低，只有 $O(n+q)$ ，畢竟除了遞迴呼叫使用的空間外也只開了一條 BIT，想當然爾出題者可能藉由記憶體限制的方式強制離線，又因為 code 短且不需要實作持久化樹狀資料結構，因此整體二分在競賽中被廣泛使用。

另外，在這題中由於我們是利用 BIT 來維護，undo 的成本很低，因此可以輕鬆的以 DFS 的方式呼叫遞迴，但有時候對於一些資料結構要 undo 其實是很困難的 (例如 DSU)，這時候我們就需要用到 BFS 的技巧。每次操作只在資料結構中增加內容，當二分搜進入新的一層後再將整個資料結構重置，便能避免掉昂貴的 undo 操作。

3.4 習題

習題的唷

習題 3.4.1: XOR and Favorite Number(CF 617E)

給定 k 及一個長度為 n 的正整數序列 s ，對於 Q 次詢問 l, r ，每次輸出 $[s_l, \dots s_r]$ 中有幾對 (i, j) 使得 $s_i \oplus s_{i+1} \cdots \oplus s_j = k$，其中 $\oplus$ 代表 XOR 運算。$n \leq 10^5; s_i, k \leq 10^6$。

習題 3.4.2: 區間眾數 (ZJ b417)

給定一個長度為 n 的正整數序列 s ，對於 m 次詢問 l, r，每次輸出 $[s_l, \dots s_r]$ 眾數的個數以及有幾種數字是眾數。$n, m \leq 10^5, 1 \leq s_i \leq n$。

習題 3.4.3: 區間逆序數對 (TIOJ 1694)

給定一個長度為 n 的正整數序列 s ，對於 q 次詢問 l, r ，每次輸出 $[s_l, \dots s_r]$ 逆序數對的個數。$n, q \leq 10^6$，$s_i \leq 10^9$。

習題 3.4.4: Coding Days (TIOJ 1840)

給定一長度為 N 的序列以及 Q 筆操作, 每筆操作可能為下列兩種:

1. `1 l r k`: 請輸出區間 $[l, r]$ 中第 k 小的值。
2. `2 p v`: 將序列中的第 p 個值改成 v。
3. `3 x v`: 關於這項操作，詳細的內容請觀察題目敘述。

$N \leq 50000, Q \leq 10000$，序列中的數皆可以用`int`儲存。

習題 3.4.5: Stamp Rally (Atcoder)

給定一張 N 個點、M 條邊的無向圖，每條邊編號為 $1 \tilde{} M$。有 Q 次詢問 (x, y, s) ，輸出最小的 i ，使得存在分別以 x 和 y 為起點的兩條路徑，路徑上的邊編號都不大於 i，且兩條路徑上共有 s 個不同的點（包含 x, y ，可以經過重複的邊或點但不會重複算到）。$N, M, Q \leq 10^5$。