Basic Graph Algorithms

zhu & sam571128

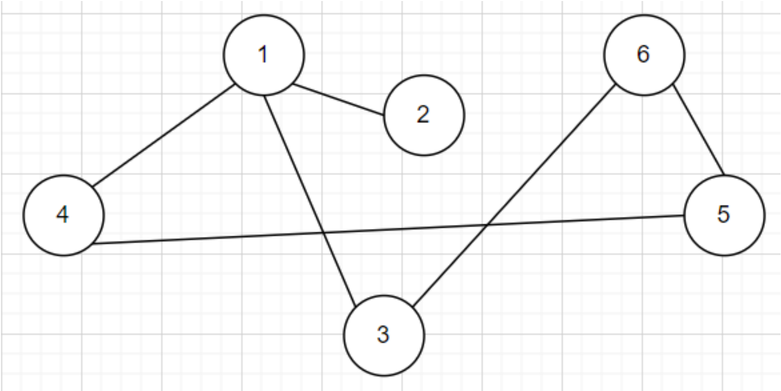
這兩天會教的東西

- 圖的儲存
- 圖的遍歷
- 歐拉迴路、哈密頓迴路
- 拓樸排序
- 最短路徑
- 樹論
  1. 性質
  2. 樹直徑、樹重心
  3. 樹壓平
  4. 樹 dp、換根 dp
- 並查集
- 最低共同祖先
- 最小生成樹

今天簡報上面所有題目的 code 我都放到 github 上了
如果不會寫的話可以去參考 ><

圖是甚麼？

- 圖 G 是一個由點集 V 和邊集 E 所構成的結構 $G = (V, E)$



名詞解釋

- 圖 (Graph)
- 點 (Vertex) / 邊 (Edge)
- 有向圖 (Directed Graph) / 無向圖 (Undirected Graph)
- 權重 (Weight)
- 點度 (Degree)
- 環 (Cycle) / 迴路 (Circuit)

名詞解釋

- 連通的 (Connected)
- 連通分量 / 連通塊 (Connected Components)
- 相鄰 (Adjacent): 鄰邊 (Adjacent Edge) / 鄰點 (Adjacent Vertex)
- 自環 (Self Loop) / 重邊 (Multiple Edge)
- 簡單圖 (Simple Graph) / 簡單路徑 (Simple Path)

特別的圖

- 樹 (Tree): 包含星星 (Star)、鍊 (Chain)
- 有向無環圖 DAG (Directed Acylic Graph)
- Functional Graph: 每個點的出度都是 1 的有向圖
- 二分圖 (Bipartite Graph): 可以被塗成兩種顏色且相同顏色不相鄰的圖

熱身一下

- 笨笨國國王給你 n 個資訊，告訴你有一條一公里的路徑可以讓你從 a 通向 b

熱身一下

- 笨笨國國王給你 n 個資訊，告訴你有一條一公里的路徑可以讓你從 a 通向 b
- 笨竹想要 $O(1)$ 知道他是否可以走一公里就從 u 到達 v

熱身一下

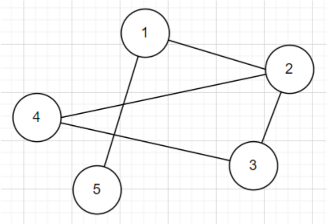
- 笨笨國國王給你 n 個資訊，告訴你有一條一公里的路徑可以讓你從 a 通向 b
- 笨竹想要 $O(1)$ 知道他是否可以走一公里就從 u 到達 v
- 這樣的話，你會想怎麼存資訊？

熱身一下

- 笨笨國國王給你 n 個資訊，告訴你有一條一公里的路徑可以讓你從 a 通向 b
- 笨竹想要 $O(1)$ 知道他是否可以走一公里就從 u 到達 v
- 這樣的話，你會想怎麼存資訊？
- 聰明的你一定想到了 > <

圖的儲存

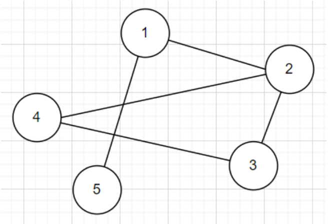
- 對於一張圖，我們用鄰接矩陣 A 儲存
- $A[i][j]$ 會存 i 到 j 的邊的資訊，例如權重或是是否有這條邊
- 在無向圖當中，$A[i][j]$ 會等於 $A[j][i]$
- 可以在 $O(1)$ 查詢 i,j 之間的邊的資訊
- 在遍歷時，要枚舉所有點，複雜度較差



	1	2	3	4	5
1	-	1	0	0	1
2	1	-	1	1	0
3	0	1	-	1	0
4	0	1	1	-	0
5	1	0	0	0	-

鄰接串列 (Adjacency List)

- 我們可以開 $|V|$ 個 vector，用來存節點 i 的鄰邊資訊
- 它的優點是可以節省空間
- 犧牲了 $O(1)$ 查詢邊的優點，換取空間
- 在遍歷時，只需要跑過相鄰的點，複雜度較好



V	$A[V]$
1	2, 5
2	1, 3, 4
3	2, 4
4	2, 3
5	1

圖的遍歷

圖的遍歷 (Traversal)

- 就是把整張圖走一遍
- 在這裡介紹 BFS 跟 DFS 兩種走訪方式

圖的遍歷 (Traversal)

- 就是把整張圖走一遍
- 在這裡介紹 BFS 跟 DFS 兩種走訪方式
- 不同的走訪方式或順序會有不同的效果

廣度優先搜尋 (Breadth First Search)

- BFS 會優先走訪自己的鄰點

廣度優先搜尋 (Breadth First Search)

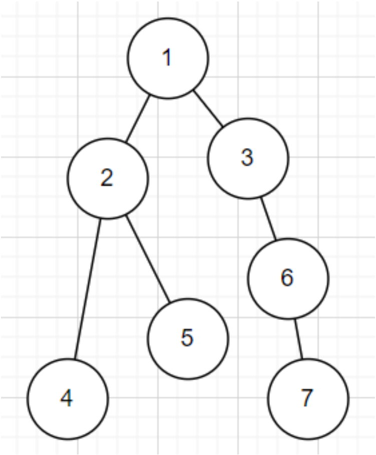
- BFS 會優先走訪自己的鄰點
- 實作上我們會開一個 queue，用來維護接下來要走的點

廣度優先搜尋 (Breadth First Search)

- BFS 會優先走訪自己的鄰點
- 實作上我們會開一個 queue，用來維護接下來要走的點
- 要做的事情是把 queue 最前面的點拿出來，將他的鄰點中沒走過的推進 queue

廣度優先搜尋 (Breadth First Search)

- BFS 會優先走訪自己的鄰點
- 實作上我們會開一個 queue，用來維護接下來要走的點
- 要做的事情是把 queue 最前面的點拿出來，將他的鄰點中沒走過的推進 queue
- 時間複雜度：$O(|V| + |E|)$



1. push 1, queue $= \{1\}$
2. pop 1 , queue $= \{2,3\}$
3. pop 2 , queue $= \{3,4,5\}$
4. pop 3 , queue $= \{4,5,6\}$
5. pop 4 , queue $= \{5,6\}$
6. pop 5 , queue $= \{6\}$
7. pop 6 , queue $= \{7\}$

BFS CODE

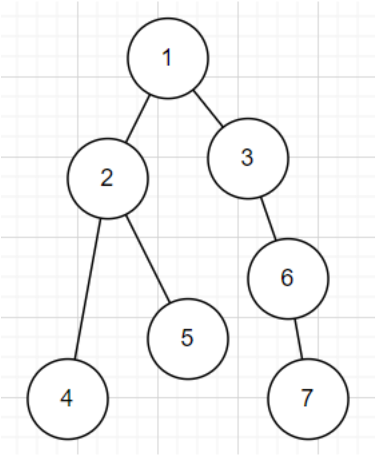
```cpp
queue<int> q;
bool vis[MAXN];
vector<int> g[MAXN];

void bfs(int v){
    q.push(v);
    vis[v]=true;
    while(!q.empty()){
        int now=q.front();
        q.pop();
        for(auto i:g[now]){
            if(!vis[i]){
                vis[i]=true;
                q.push(i);
            }
        }
    }
}
```

深度優先搜尋 (Depth First Search)

- DFS 會優先往深的地方走，概念上與 BFS 相似，但改用 stack
- 實作的時候我們會用遞迴往能走的點走，若沒有可走的點就回到上一個點
- 時間複雜度：$O(|V| + |E|)$

深度優先搜尋 (Depth First Search)



1. dfs 1, stack = {2,3}
2. dfs 2, stack = {4,5,3}
3. dfs 4, stack = {5,3}
4. dfs 5, stack = {3}
5. dfs 3, stack = {6}
6. dfs 6, stack = {7}
7. dfs 7

DFS CODE

```cpp
bool vis[MAXN];
vector<int> g[MAXN];

void dfs(int now){
    vis[now]=true;
    for(auto i:g[now]){
        if(!vis[now]) dfs(i);
    }
}
```

這是一些很裸很裸的題目 ><

CSES Counting Rooms

給一個 $n \times m$ 的地圖，'.' 代表地板，'#' 代表牆壁，問你有幾間房間

CSES Labyrinth

給一個 $n \times m$ 的地圖，'.' 代表地板，'#' 代表牆壁，'A' 代表起點，'B' 代表終點，問是否可以從 A 走到 B，若可以則輸出路徑長並且回溯路徑

喵喵喵喵喵

現在來看一些可以寫的例題 ><

Practice

CSES Round Trip II

給你一張有向圖，判斷這張圖是否有環，如果有環，輸出這個環

Practice

CSES Round Trip II

給你一張有向圖，判斷這張圖是否有環，如果有環，輸出這個環

- 當 DFS 到這個點時，我們會說走進了這個點
- 當 DFS return 的時候，我們會說離開了這個點

Practice

CSES Round Trip II

給你一張有向圖，判斷這張圖是否有環，如果有環，輸出這個環

- 當 DFS 到這個點時，我們會說走進了這個點
- 當 DFS return 的時候，我們會說離開了這個點
- 可以發現到，如果 DFS 到的這個點的鄰點中，有走進但還沒離開的點，就代表有環

Practice

CSES Round Trip II

給你一張有向圖，判斷這張圖是否有環，如果有環，輸出這個環

- 當 DFS 到這個點時，我們會說走進了這個點
- 當 DFS return 的時候，我們會說離開了這個點
- 可以發現到，如果 DFS 到的這個點的鄰點中，有走進但還沒離開的點，就代表有環
- 實作上可以使用 stack 維護走進但還沒離開的點
- 進入時，將點 push 進去。離開時，將點 pop 掉

Practice

CSES Round Trip II

給你一張有向圖，判斷這張圖是否有環，如果有環，輸出這個環

- 當 DFS 到這個點時，我們會說走進了這個點
- 當 DFS return 的時候，我們會說離開了這個點
- 可以發現到，如果 DFS 到的這個點的鄰點中，有走進但還沒離開的點，就代表有環
- 實作上可以使用 stack 維護走進但還沒離開的點
- 進入時，將點 push 進去。離開時，將點 pop 掉
- 當找到環後，從 stack 上還原出環上的點

Practice

TIOJ 1209 圖論之二分圖測試

給你一張圖，問它是否為二分圖

Practice

TIOJ 1209 圖論之二分圖測試

給你一張圖，問它是否為二分圖

- 在 DFS 的時候我們可以對它塗顏色

Practice

TIOJ 1209 圖論之二分圖測試

給你一張圖，問它是否為二分圖

- 在 DFS 的時候我們可以對它塗顏色
- 塗完 u 時，若相鄰的點中有和 u 相同的顏色，表示它不是二分圖

Practice

TIOJ 1209 圖論之二分圖測試

給你一張圖，問它是否為二分圖

- 在 DFS 的時候我們可以對它塗顏色
- 塗完 u 時，若相鄰的點中有和 u 相同的顏色，表示它不是二分圖
- 否則當 DFS 完所有連通塊後，都還沒有矛盾，表示它是一張二分圖

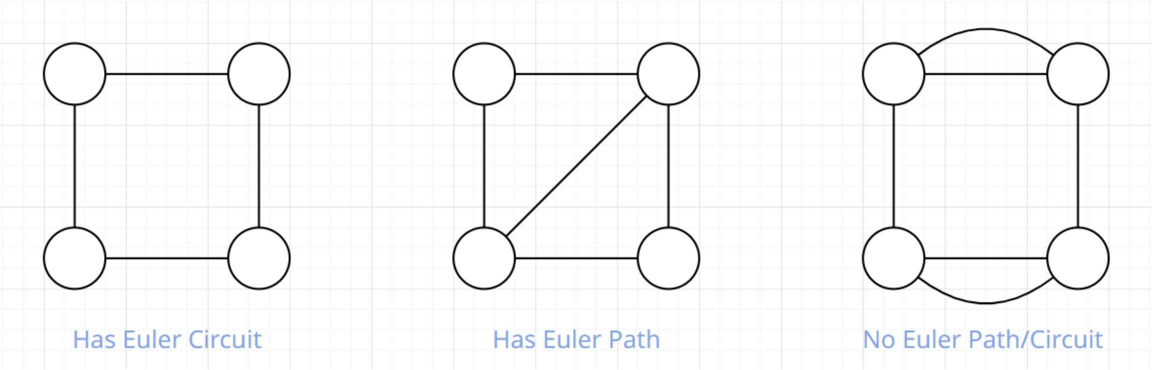
歐拉迴路、哈密頓迴路

歐拉迴路 (Eulerian Circuit)

- 常說的「一筆畫問題」
- 歐拉迴路 (Eulerian Circuit): 不重複的走過所有邊，並回到原點
- 歐拉路徑 (Eulerian Path): 從某個點開始，不重複的走過所有邊
- 中國郵差問題 (Chinese Postman Problem): 找到最短的歐拉迴路

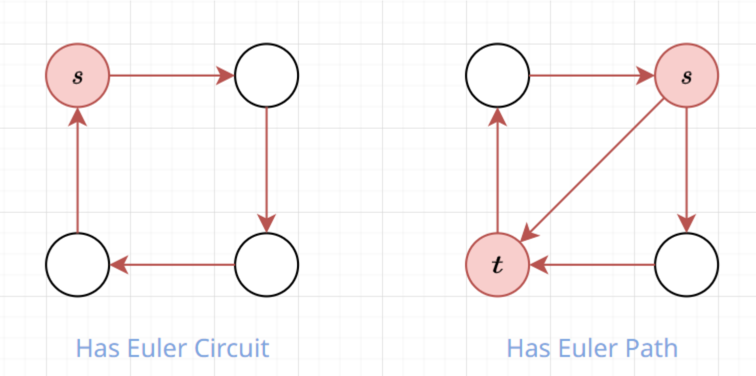
判斷一張圖是不是有歐拉路徑

- 沒有奇數度數的節點： 存在歐拉迴路
- 2 個奇數度數的節點： 存在歐拉路徑
- > 2 個奇數度數的節點： 不可能一筆畫畫完



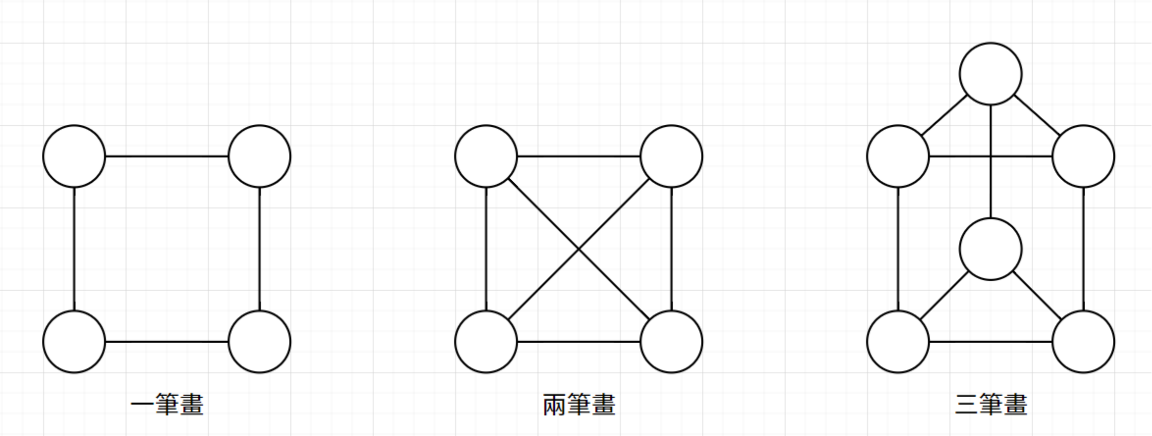
找到一個歐拉路徑

- 沒有奇數度數的節點： 任選一個點開始走不重複的邊
- 2 個奇數度數的節點： 從其中一個奇數度數的點開始走不重複的邊



N 筆畫問題

- 給你一張圖，問最少要幾筆畫才能畫完
- $\max(\frac{O}{2},\ 1)$ （O 是奇數度數節點數量）



這是一題裸題 ><

TIOJ 1084 一筆畫問題

給你一張圖，輸出這張圖的一筆畫路徑。

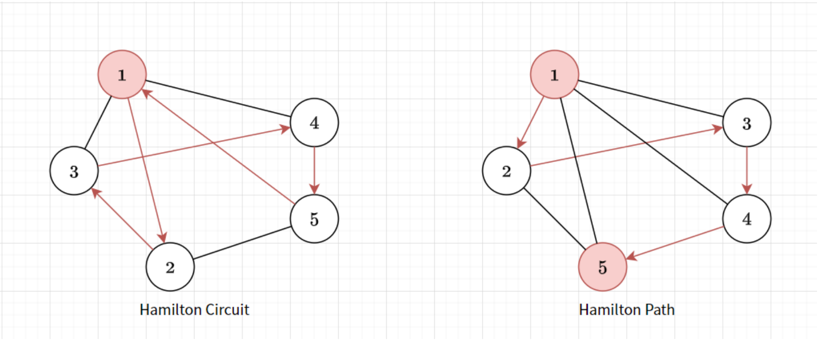
Practice

TIOJ 2171 打卡遊戲

給你一張 $A + B$ 個點，K 條邊的圖，問你一共要幾筆畫可以畫完這張圖

哈密頓迴路 (Hamilton Circuit)

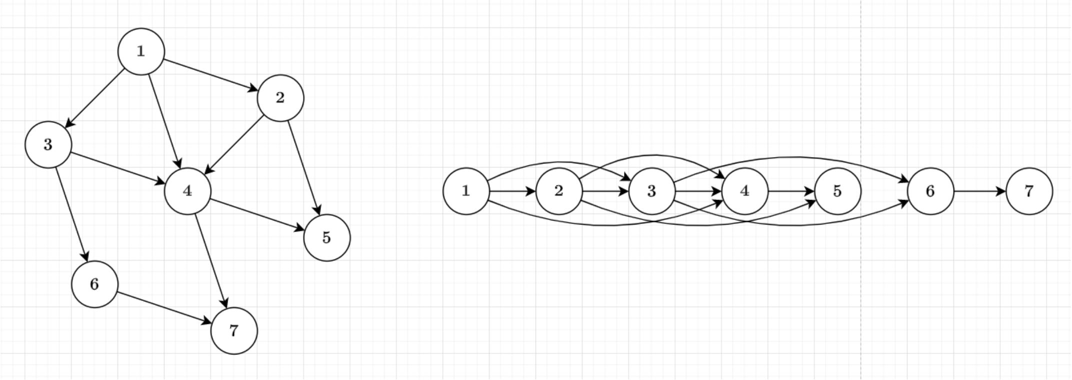
- 哈密頓迴路 (Eulerian Circuit): 不重複的走過所有點，並回到原點
- 哈密頓路徑 (Eulerian Path): 從某個點開始，不重複的走過所有點
- 旅行推銷員問題 (Travelling Salesman Problem): 找到最短的哈密頓迴路
- 可以使用位元 DP 計算答案，但這裡不會講



拓樸排序 (Topological Sort)

拓樸排序 (Topological Sort)

- 對於一張 DAG，我們可以找到一個順序
- 使得邊只會從前面的點連到後面的點



拓樸排序 (Topological Sort)

- DAG 上至少會有一個入度為 0 的點

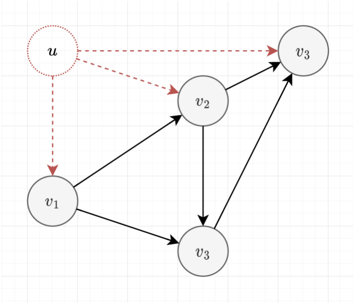
拓樸排序 (Topological Sort)

- DAG 上至少會有一個入度為 0 的點
- 如果有一個點它的入度不為 0，那就代表他要先走過前面的點

拓樸排序 (Topological Sort)

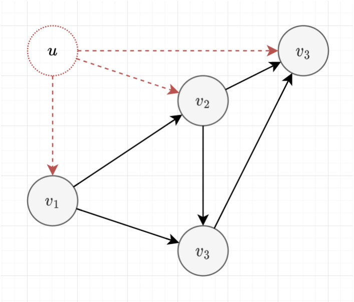
- DAG 上至少會有一個入度為 0 的點
- 如果有一個點它的入度不為 0，那就代表他要先走過前面的點
- 所以我們的起點一定是入度為 0 的點

- 我們可以開一個陣列 `indeg[v]` 表示節點 v 的入度



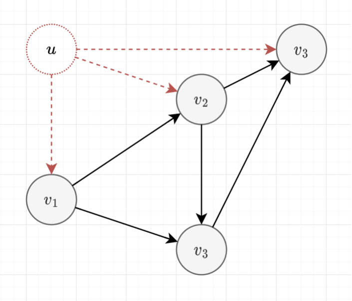
拓樸排序 (Topological Sort) 實作

- 我們可以開一個陣列 indeg[v] 表示節點 v 的入度
- 再開一個 queue 把 indeg[v]=0 的節點推進去



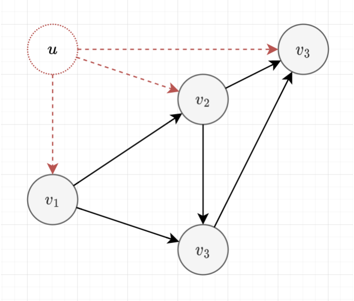
拓樸排序 (Topological Sort) 實作

- 我們可以開一個陣列 `indeg[v]` 表示節點 v 的入度
- 再開一個 queue 把 `indeg[v]=0` 的節點推進去
- 取出 queue 最前面的點 u，把它加進拓樸排序裡面
- 把 u 和它的出邊拔掉



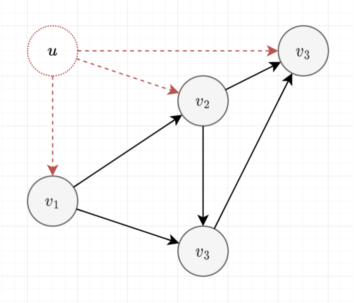
拓樸排序 (Topological Sort) 實作

- 我們可以開一個陣列 indeg[v] 表示節點 v 的入度
- 再開一個 queue 把 indeg[v]=0 的節點推進去
- 取出 queue 最前面的點 u，把它加進拓樸排序裡面
- 把 u 和它的出邊拔掉
- 再去檢查有沒有入度為 0 的點，把它推進 queue



拓樸排序（Topological Sort）實作

- 我們可以開一個陣列 indeg[v] 表示節點 v 的入度
- 再開一個 queue 把 indeg[v]=0 的節點推進去
- 取出 queue 最前面的點 u，把它加進拓樸排序裡面
- 把 u 和它的出邊拔掉
- 再去檢查有沒有入度為 0 的點，把它推進 queue
- 重複做最後就會找到拓樸排序



拓樸排序 (Topological Sort) Code

```
queue<int> q;
int indeg[MAXN]{};

for(int i=1;i<=n;++i){
    if(indeg[i]==0) q.push(i);
}
vector<int> topo;
while(!q.empty()){
    int now=q.front();
    q.pop();
    topo.emplace_back(now);
    for(auto i:g[now]){
        indeg[i]--;
        if(indeg[i]==0) q.push(i);
    }
}
```

Practice

Codeforces 510C Fox and Names

給你一些照某個奇怪字典序排好的字串，找到字母在字典序的順序。

DAG 上 DP (DP on DAG)

- 在做拓樸排序時，順便轉移 DP 式
- 我們來看個例子吧 ><

Practice

CSES Game Routes

給你一張 DAG，問你從 1 走到 n 有幾種不同路徑。

最短路徑 (Shortest Path)

最短路徑的東東

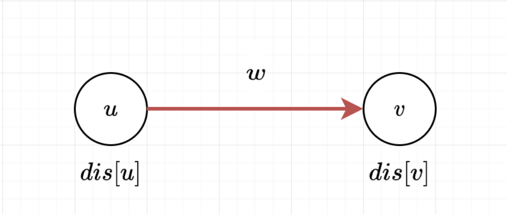
- 單點源最短路徑 (Single Source Shortest Path)
  - BFS (在圖的邊權都是 1 時)
  - Dijkstra (在圖的邊權是非負時)
  - 0-1 BFS (在圖的邊權只有 $0/1$ 時)
  - Bellman-Ford / SPFA (都可以，也可以判斷負環)
- 全點對最短路徑 (All Pairs Shortest Path)
  - Floyd-Warshall

SSSP - BFS

- 在邊權只有 1 的圖中，BFS 會依序走過離起點最近的點
- 會發現到，在這樣的順序中，已經走過的點不會再有更短的走法
- 因此，使用 BFS 即可找到這種圖中的最短路徑

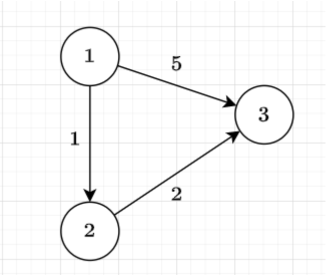
鬆弛 (Relaxation)

- 考慮在找最短距離時，判斷從 u 走到 v 可以讓走到 v 的距離變短
- 如果 $dis[u] + w < dis[v]$，就把 $dis[v]$ 更新為 $dis[u] + w$
- 這個動作就被稱為「鬆弛」



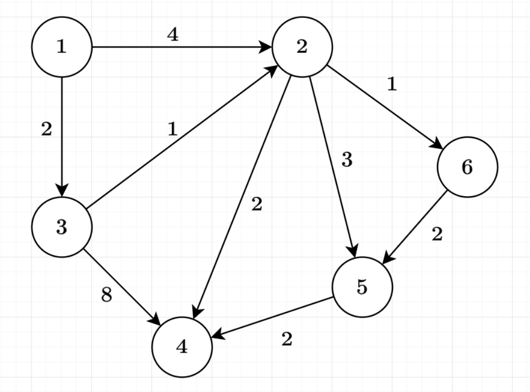
SSSP - Dijkstra

- 在剛剛講 BFS 時，因為圖的邊權都是 1，因此不會重複鬆弛
- 不過在邊權不一定是 1 的圖上，如下圖
- 從 1 開始，會先走到 $2,\ 3$，接著 3 又被 2 鬆弛
- 發現到 3 會重複走到兩次



SSSP - Dijkstra

- 重複鬆弛很顯然地會 TLE
- 那避免這個的方法就是從近的點開始先走
- 可以使用 priority_queue 來解決
- priority_queue 會幫我們把推進去的東西由小排到大！



(pq 按照起點走到 u 的距離由小到大排序)

1. push 1, dis[1] = 0, $\text{pq} = \{3, 2\}$
2. pop 3 , dis[3] = 2, $\text{pq} = \{2, 4\}$
3. pop 2 , dis[2] = 4, $\text{pq} = \{6, 4, 5\}$
4. pop 6 , dis[6] = 5, $\text{pq} = \{4, 5\}$
5. pop 4 , dis[4] = 5, $\text{pq} = \{5\}$
6. pop 5 , dis[5] = 6

SSSP - Dijkstra

- 一些實作細節 ><
- priority_queue 裡面塞的是 pair，要按照距離排序，所以會是 $\{$ 距離, 點 $\}$
- 要避免從重複的點進行鬆弛，所以當點的最短距離 $<$ pq 中的距離，就 continue
- 直接看 code 吧 ><

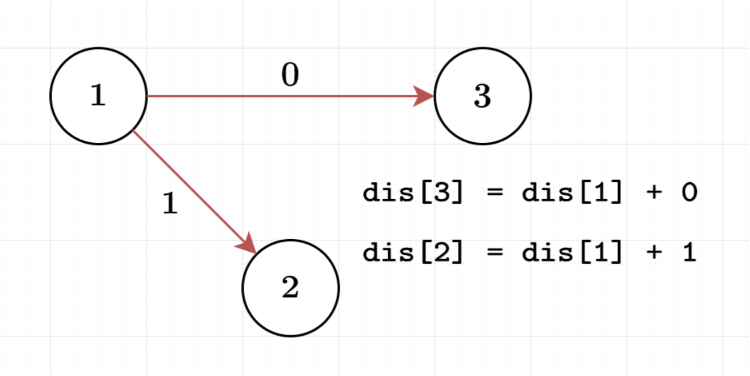
SSSP - Dijkstra CODE

```cpp
int n,m;cin>>n>>m;
vector<pair<int,int>> g[n+1];
vector<int> dis(n+1,1LL<<60);
for(int i=0;i<m;++i){
    int u,v,w;cin>>u>>v>>w;
    g[u].emplace_back(make_pair(v,w));
}
dis[1]=0;
priority_queue<pair<int,int>,vector<pair<int,int>>,greater<>> pq;
pq.push(make_pair(0,1));
while(!pq.empty()){
    int now=pq.top().second;
    int d=pq.top().first;
    pq.pop();
    if(d!=dis[now]) continue;
    for(auto i:g[now]){
        if(d+i.second<dis[i.first]){
            dis[i.first]=d+i.second;
            pq.push(make_pair(dis[i.first],i.first));
        }
    }
}
for(int i=1;i<=n;++i) cout<<dis[i]<<" ";
```

SSSP - Dijkstra

- 每個邊最多會被 `relax` 到一次，所以更新 `dis` 的次數是 $|E|$
- 放進 `pq` 複雜度會是 $O(|E|\log|V|)$
- 將點的距離初始化的複雜度會是 $O(|V|)$
- 所以總時間複雜度會是 $O(|V| + |E|\log|V|)$

- 觀察一下這張圖 owo
- 思考看看只有邊權 $0, 1$ 的時候，Dijkstra 的 priority queue 會怎麼樣



SSSP - 0-1 BFS

- 會發現當我們鬆弛了邊權是 0 的邊 $u \to v$ 之後
- 走到 v 的距離會跟走到 u 的距離相同
- 所以在 `priority queue` 裡面會被排序到最前面！

SSSP - 0-1 BFS

- 會發現當我們鬆弛了邊權是 0 的邊 $u \to v$ 之後
- 走到 v 的距離會跟走到 u 的距離相同
- 所以在 priority queue 裡面會被排序到最前面！
- 只要用一個 deque，在邊權是 0 的時候把點推到最前面
- 否則推到最後面就好了！

SSSP - 0-1 BFS

- 如果使用 Dijkstra，複雜度會是 $O(|E|\log|V|)$
- 但只使用 deque 的話，複雜度就只要 $O(|V|+|E|)$

SSSP - 0-1 BFS Practice

CodeChef Chef and Reversing

給一張有向圖，求最少需要翻轉幾條邊使得最少有一條路徑可以從 1 走到 N

CodeChef Chef and Reversing

給一張有向圖，求最少需要翻轉幾條邊使得最少有一條路徑可以從 1 走到 N

- ~~既然都放在 0-1 BFS 的例題了，那就想想它跟 0/1 有甚麼關係吧 XD~~

CodeChef Chef and Reversing

給一張有向圖，求最少需要翻轉幾條邊使得最少有一條路徑可以從 1 走到 N

- ~~既然都放在 0-1 BFS 的例題了，那就想想它跟 0/1 有甚麼關係吧 XD~~
- 要讓翻轉的邊數最少，那我們可以把它想成要經過最少條翻轉過的邊

CodeChef Chef and Reversing

給一張有向圖，求最少需要翻轉幾條邊使得最少有一條路徑可以從 1 走到 N

- ~~既然都放在 0-1 BFS 的例題了，那就想想它跟 0/1 有甚麼關係吧 XD~~
- 要讓翻轉的邊數最少，那我們可以把它想成要經過最少條翻轉過的邊
- 再轉換一下，如果我們將原本的有向邊權重設為 0，翻轉的邊權重設為 1

CodeChef Chef and Reversing

給一張有向圖，求最少需要翻轉幾條邊使得最少有一條路徑可以從 1 走到 N

- ~~既然都放在 0-1 BFS 的例題了，那就想想它跟 0/1 有甚麼關係吧 XD~~
- 要讓翻轉的邊數最少，那我們可以把它想成要經過最少條翻轉過的邊
- 再轉換一下，如果我們將原本的有向邊權重設為 0，翻轉的邊權重設為 1
- 那這不就是最短路徑問題了嗎！

CodeChef Chef and Reversing

給一張有向圖，求最少需要翻轉幾條邊使得最少有一條路徑可以從 1 走到 N

- ~~既然都放在 0-1 BFS 的例題了，那就想想它跟 0/1 有甚麼關係吧 XD~~
- 要讓翻轉的邊數最少，那我們可以把它想成要經過最少條翻轉過的邊
- 再轉換一下，如果我們將原本的有向邊權重設為 0，翻轉的邊權重設為 1
- 那這不就是最短路徑問題了嗎！
- 當然，這題也是可以用 dijkstra 做的

如果今天的權重有負的怎麼辦？

- Dijkstra 不能做負權嗎？

如果今天的權重有負的怎麼辦？

- Dijkstra 不能做負權嗎？
- 不能！因為如果有負權，路徑不一定會越走越長，鬆弛一次是不夠的！

如果今天的權重有負的怎麼辦？

- Dijkstra 不能做負權嗎？
- 不能！因為如果有負權，路徑不一定會越走越長，鬆弛一次是不夠的！
- 介紹兩個可以處理負權的方法：Bellman-Ford, SPFA

SSSP - Bellman-Ford

- 鬆弛一次不夠，那就鬆弛很多次

SSSP - Bellman-Ford

- 鬆弛一次不夠，那就鬆弛很多次
- 其實就是暴力做
- 我們可以很輕易的證明：在沒有負環的圖上，起點 s 到所有點的最短路徑最多只會經過 $|V|-1$ 條邊
- 所以跑 $|V|-1$ 次，每次都對每個點鬆馳

- 鬆弛一次不夠，那就鬆弛很多次
- 其實就是暴力做
- 我們可以很輕易的證明：在沒有負環的圖上，起點 s 到所有點的最短路徑最多只會經過 $|V|-1$ 條邊
- 所以跑 $|V|-1$ 次，每次都對每個點鬆馳
- 時間複雜度：$O(|V| \times |E|)$

SSSP - Bellman-Ford Code

```
vector<pair<pair<int,int>,int>> edges;
//input
for(int i = 1;i < n;i++){
    for(auto [p,w] : edges){
        auto [u,v] = p;
        if(dis[v] > dis[u] + w){
            dis[v] = dis[u] + w;
        }
    }
}
```

CSES High Score

給一張有向圖，走過一條邊會得到 w 分，邊權可正可負。問從 1 走到 n 最高可以拿到幾分，如果可以拿到無限大的分數，輸出 -1。

CSES High Score

給一張有向圖，走過一條邊會得到 w 分，邊權可正可負。問從 1 走到 n 最高可以拿到幾分，如果可以拿到無限大的分數，輸出 -1。

- 把邊權乘以 -1 之後，就變成最短路徑了

SSSP - Bellman-Ford Practice

CSES High Score

給一張有向圖，走過一條邊會得到 w 分，邊權可正可負。問從 1 走到 n 最高可以拿到幾分，如果可以拿到無限大的分數，輸出 -1。

- 把邊權乘以 -1 之後，就變成最短路徑了
- 不過，這題可能會出現負環

CSES High Score

給一張有向圖，走過一條邊會得到 w 分，邊權可正可負。問從 1 走到 n 最高可以拿到幾分，如果可以拿到無限大的分數，輸出 -1。

- 把邊權乘以 -1 之後，就變成最短路徑了
- 不過，這題可能會出現負環
- 要怎麼判斷 1 走到 n 會不會經過負環?

CSES High Score

給一張有向圖，走過一條邊會得到 w 分，邊權可正可負。問從 1 走到 n 最高可以拿到幾分，如果可以拿到無限大的分數，輸出 -1。

- 把邊權乘以 -1 之後，就變成最短路徑了
- 不過，這題可能會出現負環
- 要怎麼判斷 1 走到 n 會不會經過負環?
- 如果第 $|V|$ 次鬆弛到的點 v，可以從 $1 \to v \to n$ 的話
- 表示 1 到 n 會經過負環，可以使用兩次 DFS 來判斷

- 已知在沒有負環的圖上，起點 s 到所有點的最短路徑最多只會經過 $|V|-1$ 條邊

- 已知在沒有負環的圖上，起點 s 到所有點的最短路徑最多只會經過 $|V|-1$ 條邊
- 也就是說，在跑完 $|V|-1$ 次後，我們已經找到起點到所有點的最短路

SSSP - Bellman-Ford

- 已知在沒有負環的圖上，起點 s 到所有點的最短路徑最多只會經過 $|V|-1$ 條邊
- 也就是說，在跑完 $|V|-1$ 次後，我們已經找到起點到所有點的最短路
- 所以如果跑第 $|V|$ 次時，還有節點被鬆弛，那就代表有負環

SSSP - Bellman-Ford

```cpp
fill(dis,dis+N,1e18);

dis[1] = 0;

for(int i = 0;i < n;i++){
    for(auto [u,v,w] : edges){
        if(dis[v] > dis[u] + w){
            dis[v] = dis[u] + w;
        }
    }
}

bool ok = false;
for(int i = 0;i < n;i++){
    for(auto [u,v,w] : edges){
        if(dis[v] > dis[u] + w){
            dis[v] = dis[u] + w;
            if(v==n) ok = true;
        }
    }
}
```

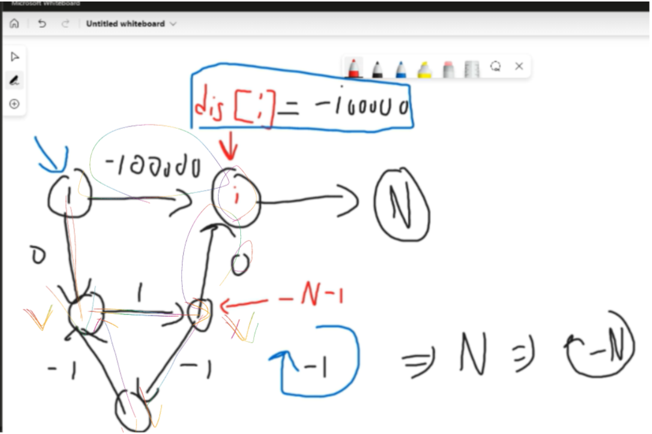
Frame Title

```
signed main(){
    fastio
    fill(dis,dis+N,1e18);

    dis[1] = 0;

    for(int i = 0;i < n;i++){
        for(auto [u,v,w] : edges){
            if(dis[v] > dis[u] + w){
                dis[v] = dis[u] + w;
            }
        }
    }

    bool ok = false;
    for(int i = 0;i < n;i++){
        for(auto [u,v,w] : edges){
            if(dis[v] > dis[u] + w){
                dis[v] = dis[u] + w;
                if(v==n) ok = true;
            }
        }
    }
    cout << (ok ? "YES\n" : "NO\n");
}
```



SSSP - SPFA

- 全名叫 Shortest Path Faster Algorithm，它是 Bellman-Ford 的優化

- 全名叫 Shortest Path Faster Algorithm，它是 Bellman-Ford 的優化
- 實作上會把更新過的節點放進一個 queue，而且只檢查 queue 中節點連出去的邊

- 全名叫 Shortest Path Faster Algorithm，它是 Bellman-Ford 的優化
- 實作上會把更新過的節點放進一個 queue，而且只檢查 queue 中節點連出去的邊
- 期望複雜度為 $O(2|E|)$

- 全名叫 Shortest Path Faster Algorithm，它是 Bellman-Ford 的優化
- 實作上會把更新過的節點放進一個 queue，而且只檢查 queue 中節點連出去的邊
- 期望複雜度為 $O(2|E|)$
- 但在最差的情況還是可能會跑到 $O(|V| \times |E|)$

- 全名叫 Shortest Path Faster Algorithm，它是 Bellman-Ford 的優化
- 實作上會把更新過的節點放進一個 queue，而且只檢查 queue 中節點連出去的邊
- 期望複雜度為 $O(2|E|)$
- 但在最差的情況還是可能會跑到 $O(|V| \times |E|)$
- 我們一樣可以用 SPFA 來檢查負環
- 如果有一個點被鬆弛了 $|V|$ 次，那就代表有負環

```cpp
vector<vector<pair<int,int>>> adj(n+1);
vector<int> inque(n+1,0), dis(n+1,INF);
//input

queue<int> q;
q.push(s);
inque[s]=1;

while(!q.empty()){
    int u=q.front();q.pop();
    inque[u]=0;
    for(auto [v,w]:adj[u]){
        if(dis[v]>dis[u]+w){
            dis[v]=dis[u]+w;
            if(!inque[v]){
                inque[v]=1;
                q.push(v);
            }
        }
    }
}
```

APSP - 最直覺的作法

- 既然已經學會了單點源的演算法
- 那要找全點對時，其實我們只要對所有的點都做一次 SSSP 就好了
- $|V|$ 次 Dijkstra: $O(|V||E|\log V)$，最糟會變成 $O(|V|^3 \log |V|)$
- $|V|$ 次 Bellman-Ford: $O(|V|(|V||E|))$，最糟會變成 $O(|V|^4)$
- 但其實有很好寫也更快速的方式，可以在 $O(|V|^3)$ 找完全點對！

APSP - Floyd-Warshall

- 它其實就是 DP
- 我們可以令 $dp[k][i][j]$ 表示從 i 走到 j 只經過 $[1, k]$ 的節點的最短距離
- 那這個的轉移式可以寫成

$$dp[k+1][i][j] = \min\{dp[k][i][j], dp[k][i][k+1] + dp[k][k+1][j]\}$$

APSP - Floyd-Warshall

- 它其實就是 DP
- 我們可以令 $dp[k][i][j]$ 表示從 i 走到 j 只經過 $[1, k]$ 的節點的最短距離
- 那這個的轉移式可以寫成

$$dp[k+1][i][j] = \min\{dp[k][i][j], dp[k][i][k+1] + dp[k][k+1][j]\}$$

- 我們可以發現第一維可以滾掉

APSP - Floyd-Warshall

- 它其實就是 DP
- 我們可以令 $dp[k][i][j]$ 表示從 i 走到 j 只經過 $[1,k]$ 的節點的最短距離
- 那這個的轉移式可以寫成

$$dp[k+1][i][j] = \min\{dp[k][i][j], dp[k][i][k+1] + dp[k][k+1][j]\}$$

- 我們可以發現第一維可以滾掉
- 時間複雜度：$O(|V|^3)$

要注意的點

- 初始化時，$dp[i][i]$ 要設成 0，其他初始化成 ∞
- 轉移的順序為 $k \to i \to j$
- 但在 2019 年的一篇論文中，提到不管哪個順序，只要跑三次都會是對的！
- 所以在使用時，如果真的忘記順序，跑三次就好了 XD

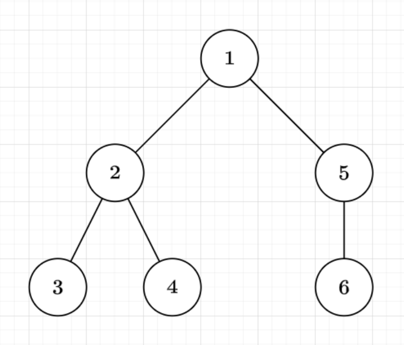
APSP - Floyd-Warshall Code

```
for(int k=0;k<n;++k){
    for(int i=0;i<n;++i){
        for(int j=0;j<n;++j){
            dp[i][j]=min(dp[i][j],dp[i][k]+dp[k][j]);
        }
    }
}
```

樹論

樹的性質

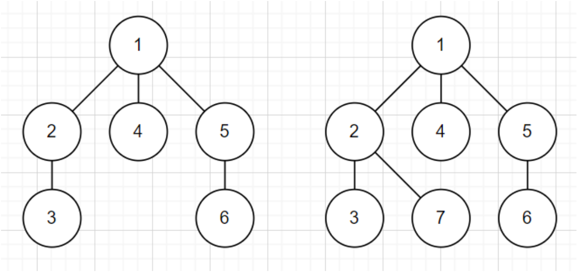
- 有 $|V|-1$ 條邊，兩兩節點之間有唯一的一條路徑
- 增加一條邊後，必形成環
- 去掉一條邊後，圖不再連通
- 很多棵樹 $\Rightarrow$ 森林



樹直徑

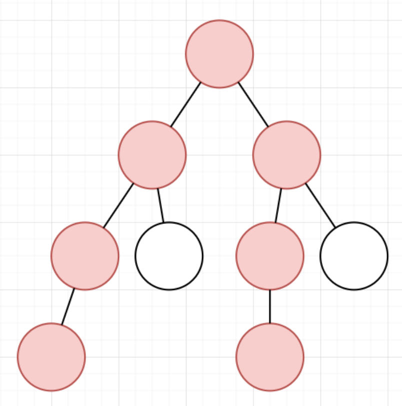
熱身一下 ><

- 找這棵樹上最長的路徑



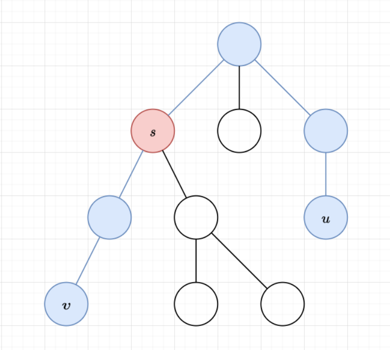
甚麼是樹直徑？

- 樹上最長的那條路徑
- 可能會不只一條樹直徑



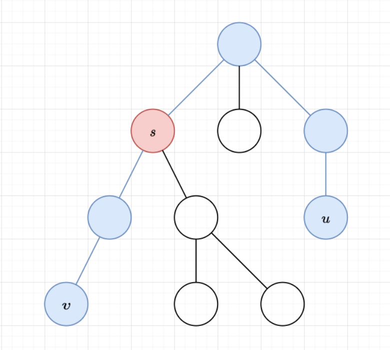
樹直徑實作

- 如果權重沒有負的
- 那我們可以找隨便一個點當起點，假設該點是 s



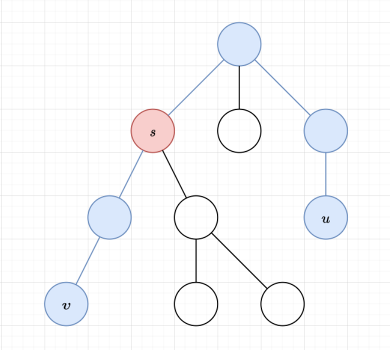
樹直徑實作

- 如果權重沒有負的
- 那我們可以找隨便一個點當起點，假設該點是 s
- 先 `dfs` 找到距離點 s 最遠的點 u

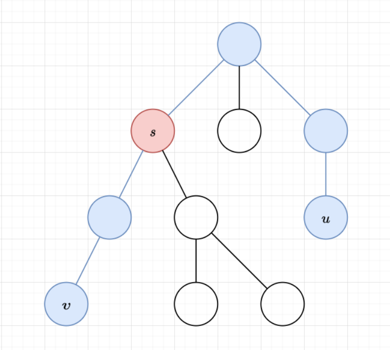


樹直徑實作

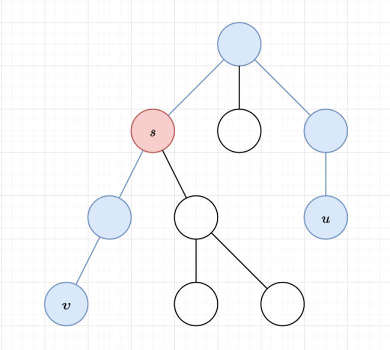
- 如果權重沒有負的
- 那我們可以找隨便一個點當起點，假設該點是 s
- 先 dfs 找到距離點 s 最遠的點 u
- 再以 u 為起點，dfs 找到距離 u 最遠的點 v
- u 和 v 之間的路徑，就是一條樹直徑



- 如果權重沒有負的
- 那我們可以找隨便一個點當起點，假設該點是 s
- 先 dfs 找到距離點 s 最遠的點 u
- 再以 u 為起點，dfs 找到距離 u 最遠的點 v
- u 和 v 之間的路徑，就是一條樹直徑
- 為甚麼我們可以確定距離樹上任一點最遠的點，必定是樹直徑的兩端？



- 如果權重沒有負的
- 那我們可以找隨便一個點當起點，假設該點是 s
- 先 dfs 找到距離點 s 最遠的點 u
- 再以 u 為起點，dfs 找到距離 u 最遠的點 v
- u 和 v 之間的路徑，就是一條樹直徑
- 為甚麼我們可以確定距離樹上任一點最遠的點，必定是樹直徑的兩端？
- 等一下會證 owo



樹直徑實作

- 那如果權重有負的怎麼辦？

樹直徑實作

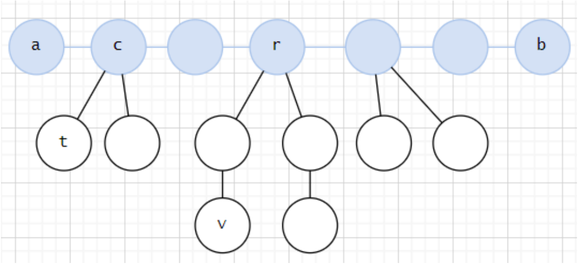
- 那如果權重有負的怎麼辦？
- dp（後面樹 dp 會講）

兩次 dfs 作法的證明

- 好欸那我們回來證明「為甚麼做兩次 dfs 會是對的？」

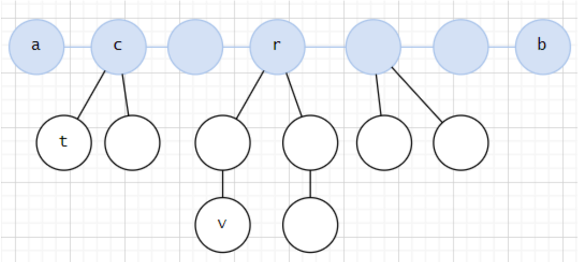
兩次 dfs 作法的證明

- 現在假設 a 到 b 之間的路徑是樹直徑
- $dis(v,a) \geq dis(v,t)$
  $\implies (dis(v,a) - dis(v,c)) \geq (dis(v,t) - dis(v,c))$
  $\implies dis(a,c) \geq dis(c,t)$



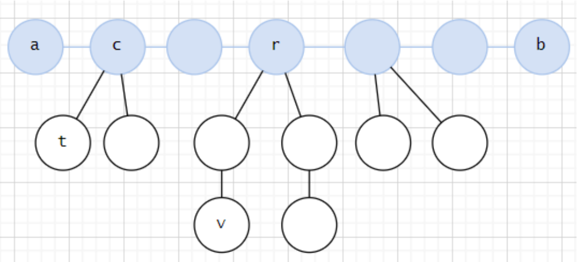
兩次 dfs 作法的證明

- 現在假設 a 到 b 之間的路徑是樹直徑
- $dis(v,a) \geq dis(v,t)$
  $\Longrightarrow (dis(v,a) - dis(v,c)) \geq (dis(v,t) - dis(v,c))$
  $\Longrightarrow dis(a,c) \geq dis(c,t)$
- 將位在樹直徑上左半邊的節點當作根
- 那它的子樹深度不會超過該節點到樹直徑最左端的距離。



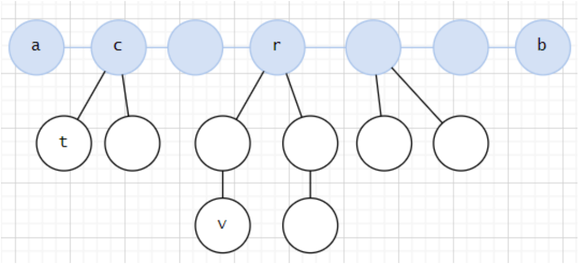
兩次 dfs 作法的證明

- 現在假設 a 到 b 之間的路徑是樹直徑
- $dis(v,a) \geq dis(v,t)$
  $\Longrightarrow (dis(v,a) - dis(v,c)) \geq (dis(v,t) - dis(v,c))$
  $\Longrightarrow dis(a,c) \geq dis(c,t)$
- 將位在樹直徑上左半邊的節點當作根
- 那它的子樹深度不會超過該節點到樹直徑最左端的距離。
- 同理，我們也可以推出右半邊也是如此



兩次 dfs 作法的證明

- 現在假設 a 到 b 之間的路徑是樹直徑
- $dis(v,a) \geq dis(v,t)$
  $\implies (dis(v,a) - dis(v,c)) \geq (dis(v,t) - dis(v,c))$
  $\implies dis(a,c) \geq dis(c,t)$
- 將位在樹直徑上左半邊的節點當作根
- 那它的子樹深度不會超過該節點到樹直徑最左端的距離。
- 同理，我們也可以推出右半邊也是如此
- 所以這邊的結論是：樹直徑上任一節點 x，其子樹深度都不會超過 $\min(dis(x,a), dis(x,b))$

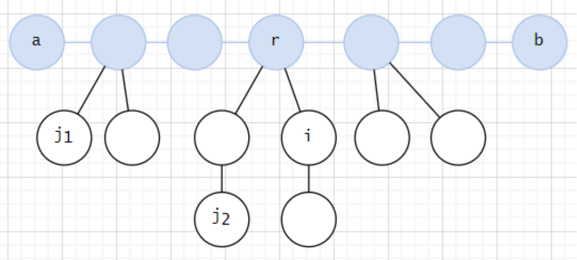


兩次 dfs 作法的證明

- 我們現在需要證明的是：為甚麼第一次 dfs 找到的點會是樹直徑的其中一端？
- Claim: 對於所有點 i，找到距離它最遠的點 j，則 j 必定滿足 $j = a$ or $j = b$

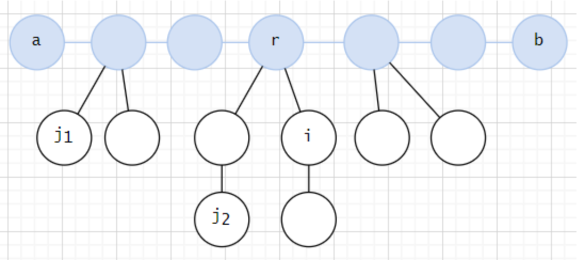
兩次 dfs 作法的證明

- 我們可以用剛剛證明出來的結論得出



兩次 dfs 作法的證明

- 我們可以用剛剛證明出來的結論得出
- if $j = j_1$,
  $dis(i, j_1) = dis(i, r) + dis(r, j_1) \leq dis(i, r) + dis(r, a) = dis(i, a)$
- if $j = j_2$,
  $dis(i, j_2) = dis(i, r) + dis(r, j_2) \leq dis(i, r) + dis(r, a) = dis(i, a)$



兩次 dfs 作法的證明

- 參考資料 Diameter of a tree and its applications

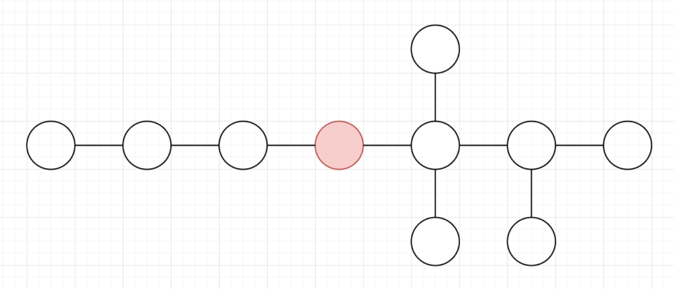
這是一題很裸很裸的題目 ><

CSES Tree Diameter

找樹直徑長度

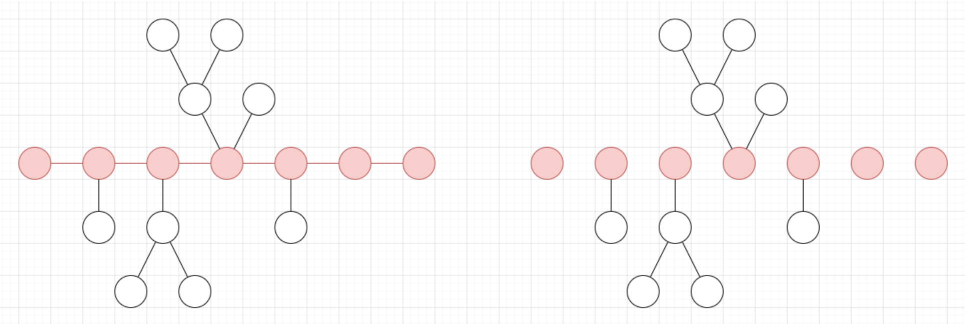
樹圓心

- 樹圓心：樹直徑中間的那個點（不一定會跟重心一樣）
- 樹圓心可能會有兩個
- 以樹圓心為根時，樹的深度會最小



直徑的性質

- 把樹上樹直徑的邊拔掉，會得到森林
- 每棵樹的節點會以在直徑上的點為根
- 若以點 u 為根的樹，它的深度不會超過到直徑端點 a 的距離 $dis(u,a)$

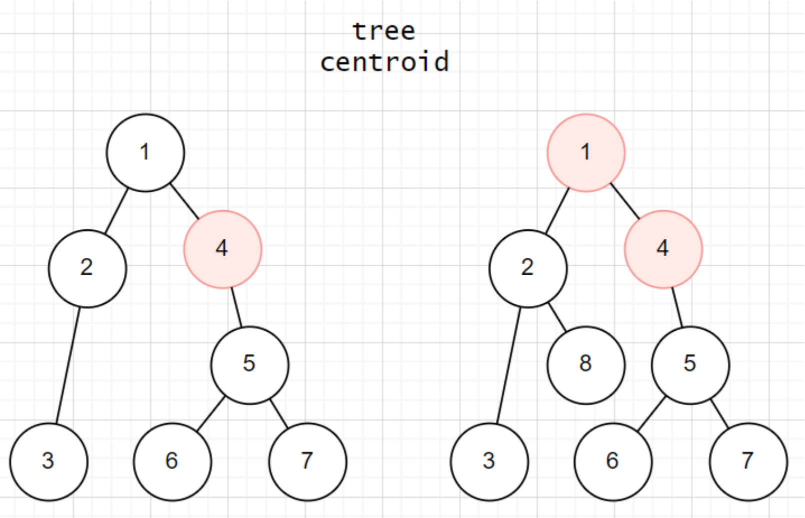


樹重心

甚麼是樹重心？

- 拔除後可以使所有連通塊的最大值最小的點
- 把它拔掉以後，每個連通塊大小都不超過 $N/2$
- 樹重心可能不只一個，但最多就兩個
- 最常見的應用是重心剖分 (Centroid Decomposition)，但我們不會講

甚麼是樹重心？



樹重心實作

- 隨便找一個節點，把它當根
- DFS 去找每個點的子樹大小

樹重心實作

- 隨便找一個節點，把它當根
- DFS 去找每個點的子樹大小
- 把該點拔去以後，會產生兩種連通塊
- 分別是「孩子的子樹大小」和「節點數 - 自己的子樹大小」

樹重心實作

- 隨便找一個節點，把它當根
- DFS 去找每個點的子樹大小
- 把該點拔去以後，會產生兩種連通塊
- 分別是「孩子的子樹大小」和「節點數 - 自己的子樹大小」
- 根據前面的定義，如果大小都不超過 $N/2$，那它就是樹重心（之一）

這是一題很裸很裸的題目 ><

NEOJ 293 樹重心

給一棵樹，找樹重心，若有多個，那就輸出編號最小的

樹壓平 (Euler Tour)

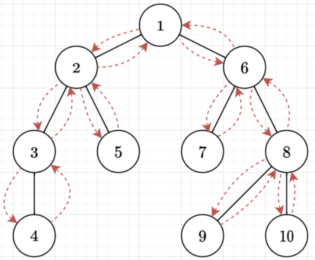
樹壓平

- 現在給你一棵樹，能不能用一個序列表示這棵樹?

樹壓平

- 現在給你一棵樹，能不能用一個序列表示這棵樹?
- 把 DFS 的順序紀錄下來 owo

- 對每個點，紀錄進入這個點和離開這個點的時間
- 我們會得到一個長度是 $2n$ 的序列



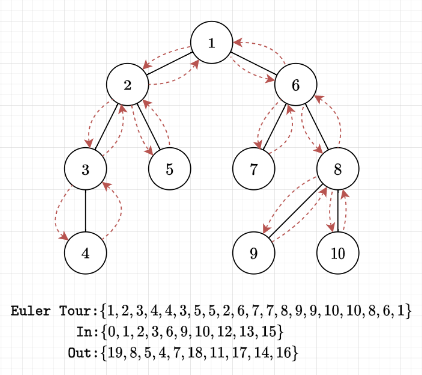
Euler Tour: $\{1,2,3,4,4,3,5,5,2,6,7,7,8,9,9,10,10,8,6,1\}$

In: $\{0,1,2,3,6,9,10,12,13,15\}$

Out: $\{19,8,5,4,7,18,11,17,14,16\}$

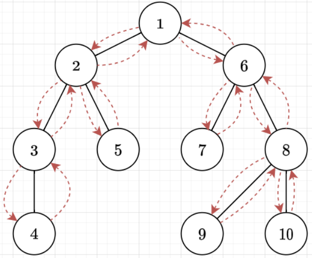
樹壓平

- 對每個點，紀錄進入這個點和離開這個點的時間
- 我們會得到一個長度是 $2n$ 的序列
- 在 $in[u]$ 和 $out[u]$ 之間的點都是 u 的子孫



樹壓平

- 對每個點，紀錄進入這個點和離開這個點的時間
- 我們會得到一個長度是 $2n$ 的序列
- 在 $in[u]$ 和 $out[u]$ 之間的點都是 u 的子孫
- 可以搭配資料結構維護子樹



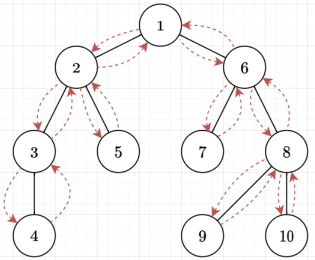
Euler Tour: $\{1,2,3,4,4,3,5,5,2,6,7,7,8,9,9,10,10,8,6,1\}$

In: $\{0,1,2,3,6,9,10,12,13,15\}$

Out: $\{19,8,5,4,7,18,11,17,14,16\}$

樹壓平

- 對每個點，紀錄進入這個點和離開這個點的時間
- 我們會得到一個長度是 $2n$ 的序列
- 在 $in[u]$ 和 $out[u]$ 之間的點都是 u 的子孫
- 可以搭配資料結構維護子樹
- 因為是把樹壓成序列，被稱作「樹壓平」



Euler Tour: $\{1, 2, 3, 4, 4, 3, 5, 5, 2, 6, 7, 7, 8, 9, 9, 10, 10, 8, 6, 1\}$
In: $\{0, 1, 2, 3, 6, 9, 10, 12, 13, 15\}$
Out: $\{19, 8, 5, 4, 7, 18, 11, 17, 14, 16\}$

樹 dp

樹 dp

- 就是在樹上做 dp
- 通常節點的狀態與子樹有關，從孩子轉移過來
- 我們可以先來看一個簡單的例子

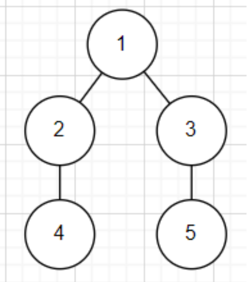
這是一個很簡單的例子 ><

CSES Subordinates

給一棵樹，問每個節點的子樹大小（不包含自己）

$1 \leq n \leq 2 \cdot 10^5$

- 以範測為例，我們可以畫出右圖



CSES Subordinates example

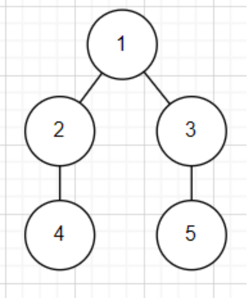
這是一個很簡單的例子 ><

CSES Subordinates

給一棵樹，問每個節點的子樹大小（不包含自己）

$1 \le n \le 2 \cdot 10^5$

- 以範測為例，我們可以畫出右圖
- 若這題用暴力解決，顯然會 TLE，我們可以考慮一下 dp 的作法



CSES Subordinates example

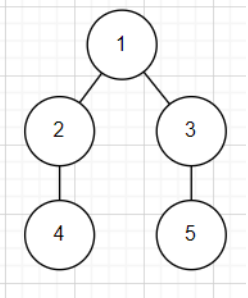
這是一個很簡單的例子 ><

CSES Subordinates

給一棵樹，問每個節點的子樹大小（不包含自己）

$1 \leq n \leq 2 \cdot 10^5$

- 以範測為例，我們可以畫出右圖
- 若這題用暴力解決，顯然會 TLE，我們可以考慮一下 dp 的作法
- 令 $dp[i] =$ 節點 i 的子樹大小



CSES Subordinates example

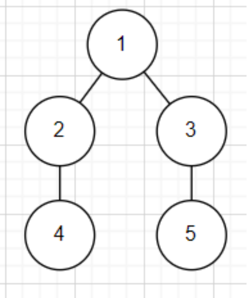
這是一個很簡單的例子 ><

CSES Subordinates

給一棵樹，問每個節點的子樹大小（不包含自己）

$1 \leq n \leq 2 \cdot 10^5$

- 以範測為例，我們可以畫出右圖
- 若這題用暴力解決，顯然會 TLE，我們可以考慮一下 dp 的作法
- 令 $dp[i] =$ 節點 i 的子樹大小
- 令 j 為節點 i 的子節點，令 cnt 為 i 的子節點個數

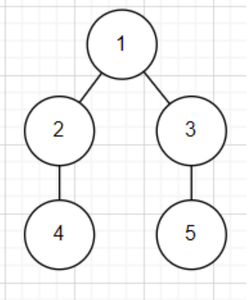


CSES Subordinates example

CSES Subordinates

給一棵樹，問每個節點的子樹大小（不包含自己）
$1 \le n \le 2 \cdot 10^5$

- 以範測為例，我們可以畫出右圖
- 若這題用暴力解決，顯然會 TLE，我們可以考慮一下 dp 的作法
- 令 $dp[i] =$ 節點 i 的子樹大小
- 令 j 為節點 i 的子節點，令 cnt 為 i 的子節點個數
- 那你會發現 $dp[i] = \sum dp[j] + cnt$
- 遞迴的時候處理



CSES Subordinates example

樹直徑

CSES Tree Diameter

找樹直徑長度

$1 \le n \le 2 \cdot 10^5$

樹直徑

CSES Tree Diameter

找樹直徑長度

$1 \leq n \leq 2 \cdot 10^5$

- 剛剛講過兩次 DFS 的作法了 ><
- 可是在有負權的時候，不能直接 DFS QwQ
- 所以我們要用樹 DP

樹直徑

CSES Tree Diameter

找樹直徑長度

$1 \leq n \leq 2 \cdot 10^5$

- 設 $dp_1[u]$ 表示 u 往下走的**最長**路徑
- 設 $dp_2[u]$ 表示 u 往下走的**次長**路徑
- 那樹直徑就會是所有的 $dp_1[u] + dp_2[u]$ 中的最大值 owo

Practice

Codeforces 1528A Parsa's Humongous Tree

給你一棵樹，每個節點 u 上有兩個權值 l_u, r_u，你可以對每個節點寫上一個數字 a_u。你希望可以知道最大的 $\sum_{(u,v)\in E} |a_u - a_v|$ 是多少?

Codeforces 1528A Parsa's Humongous Tree

給你一棵樹，每個節點 u 上有兩個權值 l_u, r_u，你可以對每個節點寫上一個數字 a_u。你希望可以知道最大的 $\sum_{(u,v)\in E} |a_u - a_v|$ 是多少?

- 這題應該可以很顯然地發現只需要注意 l 和 r

Practice

Codeforces 1528A Parsa's Humongous Tree

給你一棵樹，每個節點 u 上有兩個權值 l_u, r_u，你可以對每個節點寫上一個數字 a_u。你希望可以知道最大的 $\sum_{(u,v)\in E} |a_u - a_v|$ 是多少?

- 這題應該可以很顯然地發現只需要注意 l 和 r
- 可以開 $dp[u][0]$ 表示 $a_u = l_u$ 時子樹的最大價值
- 同理，開 $dp[u][1]$ 表示 $a_u = r_u$ 時子樹的最大價值

Practice

Codeforces 1528A Parsa's Humongous Tree

給你一棵樹，每個節點 u 上有兩個權值 l_u, r_u，你可以對每個節點寫上一個數字 a_u。你希望可以知道最大的 $\sum_{(u,v)\in E} |a_u - a_v|$ 是多少?

- 這題應該可以很顯然地發現只需要注意 l 和 r
- 可以開 $dp[u][0]$ 表示 $a_u = l_u$ 時子樹的最大價值
- 同理，開 $dp[u][1]$ 表示 $a_u = r_u$ 時子樹的最大價值
- 然後這樣就結束了

Practice

Codeforces 1646D Weight the Tree

給你一棵樹，你要將所有的點標上一個權值。定義好的節點表示它的權值等於與其相鄰的節點的權值總和。請找到可以讓好的節點最多，且總權值和最小的一種方法。

Codeforces 1646D Weight the Tree

給你一棵樹，你要將所有的點標上一個權值。定義好的節點表示它的權值等於與其相鄰的節點的權值總和。請找到可以讓好的節點最多，且總權值和最小的一種方法。

- 觀察到當 $n \geq 3$ 的時候，好的節點一定不會與好的節點相鄰

Practice

Codeforces 1646D Weight the Tree

給你一棵樹，你要將所有的點標上一個權值。定義好的節點表示它的權值等於與其相鄰的節點的權值總和。請找到可以讓好的節點最多，且總權值和最小的一種方法。

- 觀察到當 $n \geq 3$ 的時候，好的節點一定不會與好的節點相鄰
- 定義 DP 式 $dp[u][0]$ 表示整個子樹的最優解，且 u 是壞的節點
- 同理，定義 $dp[u][1]$ 表示整個子樹的最優解，且 u 是好的節點

Practice

Codeforces 1646D Weight the Tree

給你一棵樹，你要將所有的點標上一個權值。定義好的節點表示它的權值等於與其相鄰的節點的權值總和。請找到可以讓好的節點最多，且總權值和最小的一種方法。

- 觀察到當 $n \geq 3$ 的時候，好的節點一定不會與好的節點相鄰
- 定義 DP 式 $dp[u][0]$ 表示整個子樹的最優解，且 u 是壞的節點
- 同理，定義 $dp[u][1]$ 表示整個子樹的最優解，且 u 是好的節點
- 在轉移時，好的節點只能從壞的節點轉移

Practice

Codeforces 1646D Weight the Tree

給你一棵樹，你要將所有的點標上一個權值。定義好的節點表示它的權值等於與其相鄰的節點的權值總和。請找到可以讓好的節點最多，且總權值和最小的一種方法。

- 觀察到當 $n \geq 3$ 的時候，好的節點一定不會與好的節點相鄰
- 定義 DP 式 $dp[u][0]$ 表示整個子樹的最優解，且 u 是壞的節點
- 同理，定義 $dp[u][1]$ 表示整個子樹的最優解，且 u 是好的節點
- 在轉移時，好的節點只能從壞的節點轉移
- 由於要輸出一種方案，在轉移時，順便維護轉移來的人即可

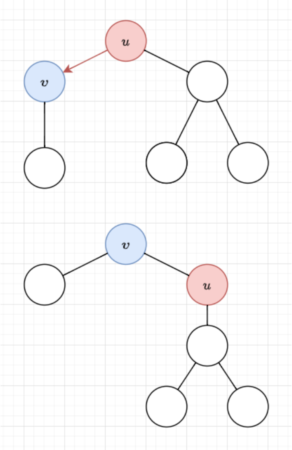
換根 DP (Rerooting DP)

換根 dp (Rerooting DP)

- 有時候題目會希望你找到對於所有點當根的 DP 答案
- 假設一次樹 DP 要花 $O(n)$，每一次都從某個點開始 DFS 就要 $O(n^2)$ 了耶 owo
- 所以我們不能夠直接做 n 次

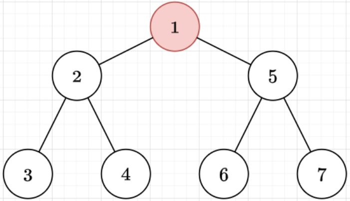
換根 dp (Rerooting DP)

- 在這裡我們要用到換根的想法
- 假設我們知道節點 u 的 DP 值
- 想要知道與 u 相鄰的點 v 當根的 DP 值
- 我們可以在 $O(1)$ 或 $O(\log n)$ 的時間做完
- 這個技巧在日本叫做「全方位木 DP」



換根 dp (Rerooting DP)

- 會換根之後，要怎麼換才能得到所有點的值?

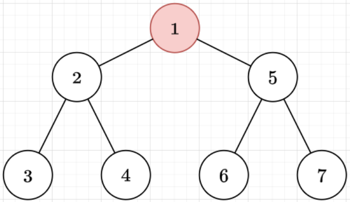


換根順序: (左到右，上到下)

$1 \to 2$	$2 \to 3$	$3 \to 2$
$2 \to 4$	$4 \to 2$	$2 \to 1$
$1 \to 5$	$5 \to 6$	$6 \to 5$
$5 \to 7$	$7 \to 5$	$5 \to 1$

換根 dp (Rerooting DP)

- 會換根之後，要怎麼換才能得到所有點的值?
- 什麼順序可以讓我們剛好走過樹上每個點?

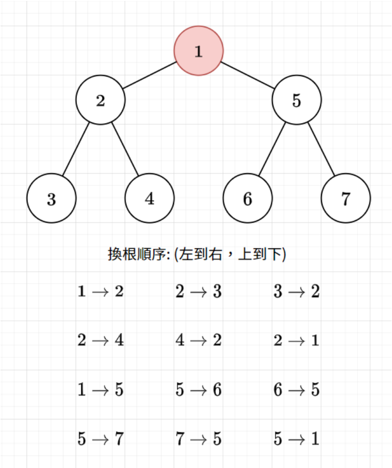


換根順序: (左到右，上到下)

$1 \to 2$	$2 \to 3$	$3 \to 2$
$2 \to 4$	$4 \to 2$	$2 \to 1$
$1 \to 5$	$5 \to 6$	$6 \to 5$
$5 \to 7$	$7 \to 5$	$5 \to 1$

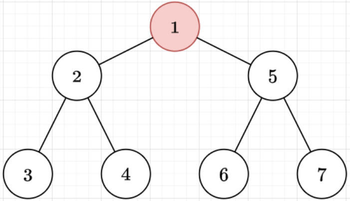
換根 dp (Rerooting DP)

- 會換根之後，要怎麼換才能得到所有點的值?
- 什麼順序可以讓我們剛好走過樹上每個點?
- Euler Tour!



換根 dp (Rerooting DP)

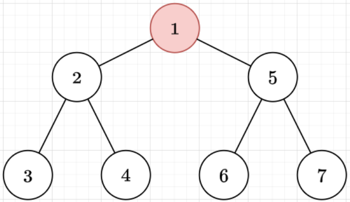
- 會換根之後，要怎麼換才能得到所有點的值?
- 什麼順序可以讓我們剛好走過樹上每個點?
- Euler Tour!
- 一共只會做 $O(n)$ 次換根
- 複雜度比起每次重新做好很多！



換根順序: (左到右，上到下)

$1 \to 2$	$2 \to 3$	$3 \to 2$
$2 \to 4$	$4 \to 2$	$2 \to 1$
$1 \to 5$	$5 \to 6$	$6 \to 5$
$5 \to 7$	$7 \to 5$	$5 \to 1$

- 會換根之後，要怎麼換才能得到所有點的值?
- 什麼順序可以讓我們剛好走過樹上每個點?
- Euler Tour!
- 一共只會做 $O(n)$ 次換根
- 複雜度比起每次重新做好很多！
- 直接看例題應該比較好懂 owo



換根順序: (左到右，上到下)

$1 \to 2$	$2 \to 3$	$3 \to 2$
$2 \to 4$	$4 \to 2$	$2 \to 1$
$1 \to 5$	$5 \to 6$	$6 \to 5$
$5 \to 7$	$7 \to 5$	$5 \to 1$

換根 DP Practice

Codeforces 1092F Tree with Maximum Cost

給你一棵樹，每個點 i 都有一個點權 a_i，定義以某個點 v 為根時，樹的價值為 $\sum_{i=1}^{n} dist(v,i) \cdot a_i$。問這棵樹的最大價值可以是多少?

$1 \leq n \leq 2 \cdot 10^5$

Codeforces 1092F Tree with Maximum Cost

給你一棵樹，每個點 i 都有一個點權 a_i，定義以某個點 v 為根時，樹的價值為 $\sum_{i=1}^{n} dist(v, i) \cdot a_i$。問這棵樹的最大價值可以是多少?

$1 \leq n \leq 2 \cdot 10^5$

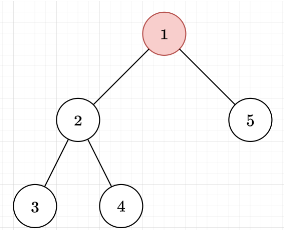
- 首先，如果我們固定了一個點（例如：1）當根
- 要怎麼計算這棵樹的價值呢?

Codeforces 1092F Tree with Maximum Cost

給你一棵樹，每個點 i 都有一個點權 a_i，定義以某個點 v 為根時，樹的價值為 $\sum_{i=1}^{n} dist(v,i) \cdot a_i$。問這棵樹的最大價值可以是多少?

$1 \leq n \leq 2 \cdot 10^5$

- 觀察到 $dist(v,i)$ 其實就是 i 的深度
- 因此我們設 $dp[v]$ 表示 v 的子樹的總價值
- 轉移式： $dp[v] = \sum_{u \in v_c} dp[u] + \mathtt{dep}[v] \cdot a_i$



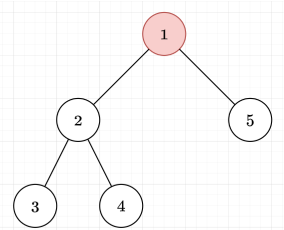
以某個點 u 為根之後，dis(u,i) 就是 i 的深度

Codeforces 1092F Tree with Maximum Cost

給你一棵樹，每個點 i 都有一個點權 a_i，定義以某個點 v 為根時，樹的價值為 $\sum_{i=1}^{n} dist(v,i) \cdot a_i$。問這棵樹的最大價值可以是多少?

$1 \le n \le 2 \cdot 10^5$

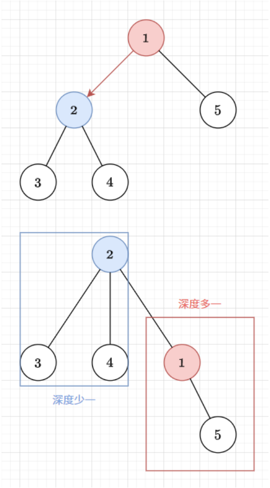
- 觀察到 $dist(v,i)$ 其實就是 i 的深度
- 因此我們設 $dp[v]$ 表示 v 的子樹的總價值
- 轉移式: $dp[v] = \sum_{u \in v_c} dp[u] + \mathtt{dep}[v] \cdot a_i$
- 找到以 1 為根的答案了，然後呢 ><



以某個點 u 為根之後，dis(u,i) 就是 i 的深度

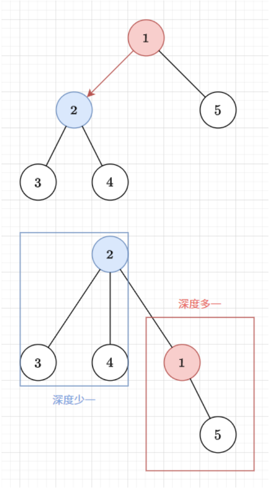
換根 DP Practice

- 再來我們要想如何換根 owo



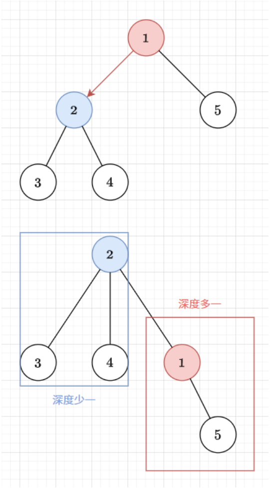
換根 DP Practice

- 再來我們要想如何換根 owo
- 根從 $u \to v$ 最直接的改變就是深度
- v 的子樹深度都會少 1，u 的子樹深度都會多 1

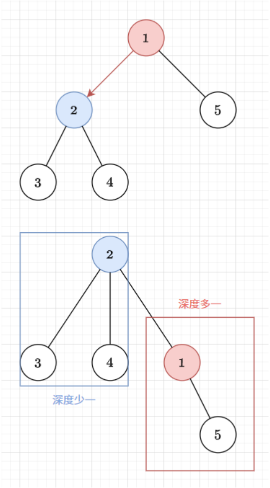


換根 DP Practice

- 再來我們要想如何換根 owo
- 根從 $u \to v$ 最直接的改變就是深度
- v 的子樹深度都會少 1，u 的子樹深度都會多 1
- $dp[u]$ 不會再包含 v 的子樹，$dp[u]$ 要減少 $dp[v]$

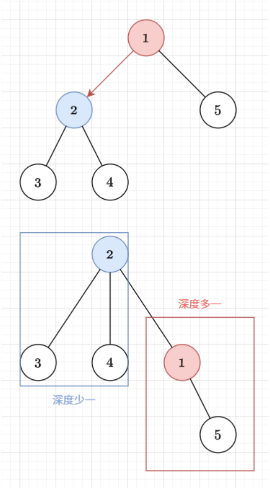


- 再來我們要想如何換根 owo
- 根從 $u \to v$ 最直接的改變就是深度
- v 的子樹深度都會少 1，u 的子樹深度都會多 1
- $dp[u]$ 不會再包含 v 的子樹，$dp[u]$ 要減少 $dp[v]$
- u 子孫深度都多 1，$dp[u]$ 要增加 $\sum_{i \in U} a_i$
- v 子孫深度都少 1，$dp[v]$ 要減少 $\sum_{j \in V} a_j$



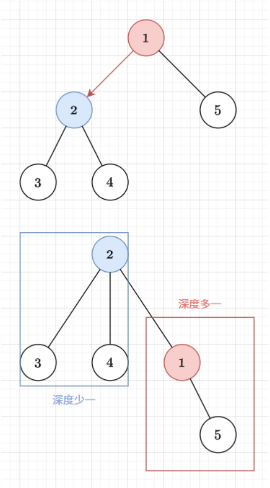
換根 DP Practice

- 再來我們要想如何換根 owo
- 根從 $u \to v$ 最直接的改變就是深度
- v 的子樹深度都會少 1，u 的子樹深度都會多 1
- $dp[u]$ 不會再包含 v 的子樹，$dp[u]$ 要減少 $dp[v]$
- u 子孫深度都多 1，$dp[u]$ 要增加 $\sum_{i \in U} a_i$
- v 子孫深度都少 1，$dp[v]$ 要減少 $\sum_{j \in V} a_j$
- $dp[v]$ 要加上 $dp[u]$，因為 u 變成 v 的孩子了



換根 DP Practice

- 再來我們要想如何換根 owo
- 根從 $u \to v$ 最直接的改變就是深度
- v 的子樹深度都會少 1，u 的子樹深度都會多 1
- $dp[u]$ 不會再包含 v 的子樹，$dp[u]$ 要減少 $dp[v]$
- u 子孫深度都多 1，$dp[u]$ 要增加 $\sum_{i \in U} a_i$
- v 子孫深度都少 1，$dp[v]$ 要減少 $\sum_{j \in V} a_j$
- $dp[v]$ 要加上 $dp[u]$，因為 u 變成 v 的孩子了
- 實作上會再開一個 $sum[u]$ 表示 $\sum_{i \in U} a_i$



換根 DP Practice

Codeforces 1092F Tree with Maximum Cost

給你一棵樹，每個點 i 都有一個點權 a_i，定義以某個點 v 為根時，樹的價值為 $\sum_{i=1}^{n} dist(v,i) \cdot a_i$。問這棵樹的價值可以是多少?

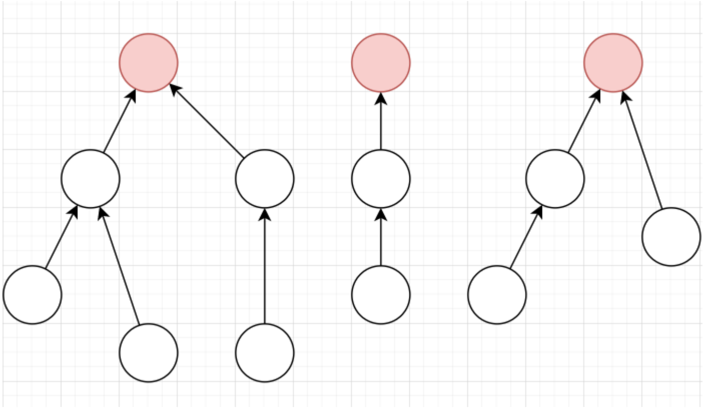
$1 \leq n \leq 2 \cdot 10^5$

- 所以我們花了 $O(n)$ 的時間找到某個點為根的答案
- 接著一次換根需要 $O(1)$，我們換了 $O(n)$ 次
- 複雜度是 $O(n)$，比起做 n 次 DFS 快了超級多 owo

並查集

甚麼是並查集？

- 並查集，又叫 DSU (Disjoint Set Union)
- 一種資料結構，可以維護一些集合，支援**合併**和**查詢**兩種操作



操作

- 每個集合都是一棵樹

操作

- 每個集合都是一棵樹
- 最初每個點都是一個集合，且自己就是自己的根節點的父節點

操作

- 每個集合都是一棵樹
- 最初每個點都是一個集合，且自己就是自己的根節點的父節點
- 要查詢兩個點是否位於同一個集合，就看他們的祖先是不是同一個

操作

- 每個集合都是一棵樹
- 最初每個點都是一個集合，且自己就是自己的根節點的父節點
- 要查詢兩個點是否位於同一個集合，就看他們的祖先是不是同一個
- 要把兩個集合合併，就讓一個集合的根節點的父節點變成另一個集合的根節點

code

```
vector<int> dsu,sz;

void init(){
    dsu.resize(n+1);
    for(int i=1;i<=n;++i) dsu[i]=i;
}

int findDSU(int a){
    if(dsu[a]==a) return dsu[a];
    return findDSU(dsu[a]);
}

void unionDSU(int a,int b){
    a=findDSU(a);
    b=findDSU(b);
    if(a==b) return;
    dsu[b]=a;
}
```

優化

- 你會發現合併以後可能會有很長的鍊

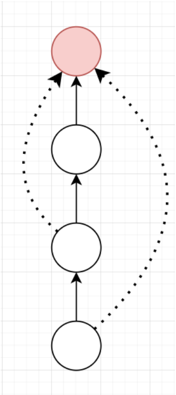
優化

- 你會發現合併以後可能會有很長的鍊
- 如果是一條鍊，那麼每次向上找祖先時，複雜度最差會是 $O(n)$

優化

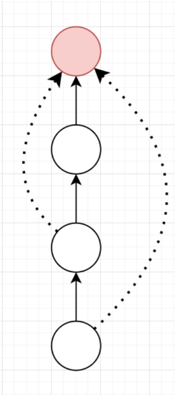
- 你會發現合併以後可能會有很長的鍊
- 如果是一條鍊，那麼每次向上找祖先時，複雜度最差會是 $O(n)$
- 所以我們的目標是：
  不讓他找祖先時需要走那麼多步，又或者是說，乾脆不要讓他有機會形成一條鏈

- 就是把路變短



優化 - 路徑壓縮

- 就是把路變短
- 讓每個點往上走一步就會到根節點
- 在遞迴（`findDSU`）的時候做掉一整條
- 這樣複雜度會變 $O(\log n)$



code

```cpp
vector<int> dsu;

void init(){
    dsu.resize(n+1);
    for(int i=1;i<=n;++i) dsu[i]=i;
}

int findDSU(int a){
    if(dsu[a]!=a) dsu[a]=findDSU(dsu[a]);
    return dsu[a];
}

void unionDSU(int a,int b){
    a=findDSU(a);
    b=findDSU(b);
    if(a==b) return;
    dsu[b]=a;
}
```

- 除了把路變短，我們可以一開始就讓路不要那麼長

優化 - 啟發式合併

- 除了把路變短，我們可以一開始就讓路不要那麼長
- 在合併的時候將較大集合的根節點當作小集合的根節點的父節點

優化 - 啟發式合併

- 除了把路變短，我們可以一開始就讓路不要那麼長
- 在合併的時候將較大集合的根節點當作小集合的根節點的父節點
- 這樣每往上走一個點，子樹大小至少會變兩倍

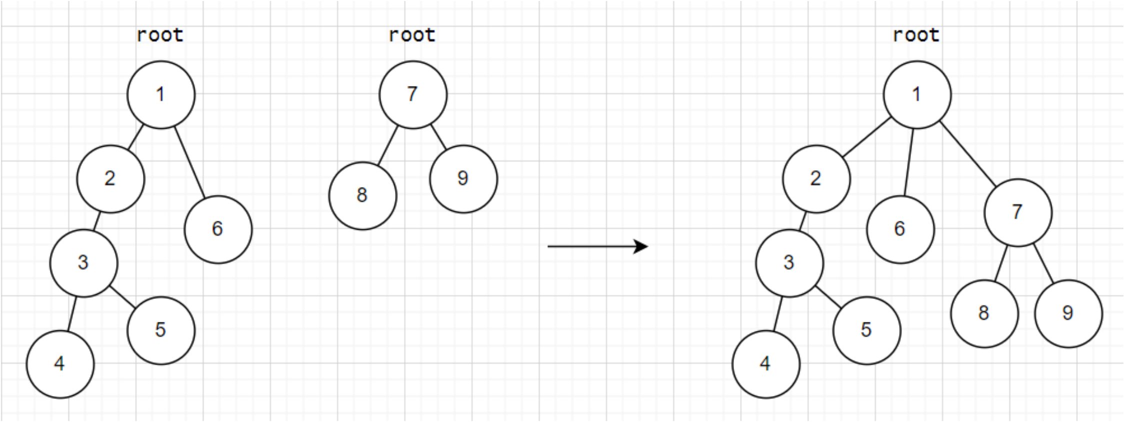
優化 - 啟發式合併

- 除了把路變短，我們可以一開始就讓路不要那麼長
- 在合併的時候將較大集合的根節點當作小集合的根節點的父節點
- 這樣每往上走一個點，子樹大小至少會變兩倍
- 因此樹的深度最多會是 $O(\log n)$
- 換句話說，最多走 $O(\log n)$ 步就會抵達根節點

優化 - 啟發式合併

- 除了把路變短，我們可以一開始就讓路不要那麼長
- 在合併的時候將較大集合的根節點當作小集合的根節點的父節點
- 這樣每往上走一個點，子樹大小至少會變兩倍
- 因此樹的深度最多會是 $O(\log n)$
- 換句話說，最多走 $O(\log n)$ 步就會抵達根節點
- 實作的話多維護一個集合大小就好

優化 - 啟發式合併



code

```cpp
vector<int> dsu,sz;

void init(){
    dsu.resize(n+1);
    for(int i=1;i<=n;++i) dsu[i]=i;
}

int findDSU(int a){
    if(dsu[a]==a) return dsu[a];
    return findDSU(dsu[a]);
}

void unionDSU(int a,int b){
    a=findDSU(a);
    b=findDSU(b);
    if(sz[a]<sz[b]) swap(a,b);
    dsu[b]=a;
    sz[a]+=sz[b];
}
```

優化

- 好耶好快
- 還可以更快！！！

優化

- 好耶好快
- 還可以更快！!!
- 兩個優化一起用！

優化

- 好耶好快
- 還可以更快！！！
- 兩個優化一起用！
- 兩個一起用的時候複雜度是 $O(\alpha(n))$
- α 是一個成長率超級無敵霹靂小的函數
- 非常接近常數

code

```cpp
vector<int> dsu,sz;

void init(){
    dsu.resize(n+1);
    for(int i=1;i<=n;++i) dsu[i]=i;
}

int findDSU(int a){
    if(dsu[a]!=a) dsu[a]=findDSU(dsu[a]);
    return dsu[a];
}

void unionDSU(int a,int b){
    a=findDSU(a);
    b=findDSU(b);
    if(sz[a]<sz[b]) swap(a,b);
    dsu[b]=a;
    sz[a]+=sz[b];
}
```

優化

- 但並不是所有題目都可以用以上兩種優化
- 因為這兩種優化會改到樹本身的結構

喵喵喵喵喵

現在來看一些可以寫的例題 ><

Practice

CodeForces D. The Number of Imposters

imposter 只會說謊，crewmate 只會說真話
給 m 個線索，例如：a 說 b 是 crewmate
在符合這些關係的前提下，輸出最多會有幾個 imposter，若他給的線索已經造成矛盾，那就輸出 -1。

Practice

CodeForces D. The Number of Imposters

imposter 只會說謊，crewmate 只會說真話
給 m 個線索，例如：a 說 b 是 crewmate
在符合這些關係的前提下，輸出最多會有幾個 imposter，若他給的線索已經造成矛盾，那就輸出 -1。

- 轉換一下題目
- 當 a 說 b 是 crewmate，a 和 b 的身分相同
- 當 a 說 b 是 imposter，a 和 b 的身分相反

Practice

CodeForces D. The Number of Imposters

imposter 只會說謊，crewmate 只會說真話
給 m 個線索，例如：a 說 b 是 crewmate
在符合這些關係的前提下，輸出最多會有幾個 imposter，若他給的線索已經造成矛盾，那就輸出 -1。

- 轉換一下題目
- 當 a 說 b 是 crewmate，a 和 b 的身分相同
- 當 a 說 b 是 imposter，a 和 b 的身分相反
- 只考慮第二種邊，題目就變成
- 給你一張圖，然後判斷它是不是二分圖

Practice

CodeForces D. The Number of Imposters

imposter 只會說謊，crewmate 只會說真話
給 m 個線索，例如：a 說 b 是 crewmate
在符合這些關係的前提下，輸出最多會有幾個 imposter，若他給的線索已經造成矛盾，那就輸出 -1。

- 轉換一下題目
- 當 a 說 b 是 crewmate，a 和 b 的身分相同
- 當 a 說 b 是 imposter，a 和 b 的身分相反
- 只考慮第二種邊，題目就變成
- 給你一張圖，然後判斷它是不是二分圖
- 就 DFS 塗顏色就好了 R

Practice

CodeForces D. The Number of Imposters

imposter 只會說謊，crewmate 只會說真話
給 m 個線索，例如：a 說 b 是 crewmate
在符合這些關係的前提下，輸出最多會有幾個 imposter，若他給的線索已經造成矛盾，那就輸出 -1。

- 轉換一下題目
- 當 a 說 b 是 crewmate，a 和 b 的身分相同
- 當 a 說 b 是 imposter，a 和 b 的身分相反
- 只考慮第二種邊，題目就變成
- 給你一張圖，然後判斷它是不是二分圖
- 就 DFS 塗顏色就好了 R
- 對！這題就這樣

Practice

CodeForces D. The Number of Imposters

imposter 只會說謊，crewmate 只會說真話
給 m 個線索，例如：a 說 b 是 crewmate
在符合這些關係的前提下，輸出最多會有幾個 imposter，若他給的線索已經造成矛盾，那就輸出 -1。

- 轉換一下題目
- 當 a 說 b 是 crewmate，a 和 b 的身分相同
- 當 a 說 b 是 imposter，a 和 b 的身分相反
- 只考慮第二種邊，題目就變成
- 給你一張圖，然後判斷它是不是二分圖
- 就 DFS 塗顏色就好了 R
- 對！這題就這樣
- 那我們來看一下另一題

Practice

CodeForces DSU edu step2 J First Non-Bipartite Edge

一開始圖上只有 n 個點，接著每次加入一條邊
請找出一條邊，使得二分圖加上該邊後不是二分圖

Practice

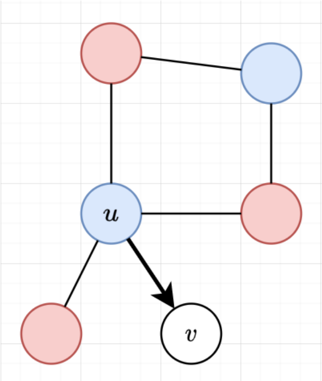
CodeForces DSU edu step2 J First Non-Bipartite Edge

一開始圖上只有 n 個點，接著每次加入一條邊
請找出一條邊，使得二分圖加上該邊後不是二分圖

- 這題其實有二分搜 + DFS 塗色的做法
- 可是我們要用 DSU owo

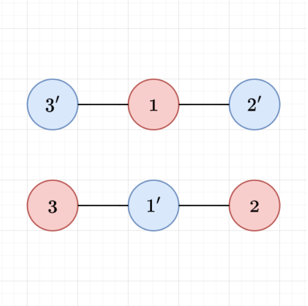
DSU 判二分圖

- 思考一下 DFS 判斷二分圖的方式
- 我們會先決定好起點的顏色
- 接著從 u 走到 v 時，將 v 和 u 塗上不同的顏色



DSU 判二分圖

- 既然我們會去決定 u 的顏色再進行塗色
- 那如果我們把兩種塗色方法一起做呢?
- 對每個點都開兩個不同的點，並分別標上 1 和 0
- 在塗色的時候，把 u 連 v'，v 連 u'
- 最後會建出兩個塗上不同顏色的二分圖

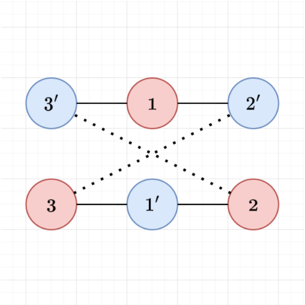


DSU 判二分圖

- 可是如果這張圖不是二分圖怎麼辦 QwQ

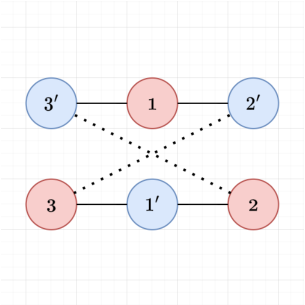
- 可是如果這張圖不是二分圖怎麼辦 QwQ
- 照上面的方式處理，會發生什麼事情

- 如果不是二分圖，兩張圖會連在一起！



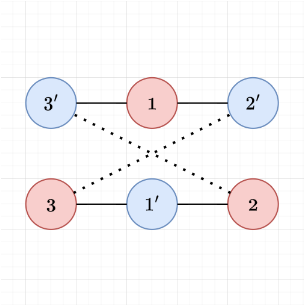
DSU 判二分圖

- 如果不是二分圖，兩張圖會連在一起！
- 所以只要在加邊的時候，判斷要連的 u, v 是不是在同個連通塊就好了！



DSU 判二分圖

- 如果不是二分圖，兩張圖會連在一起！
- 所以只要在加邊的時候，判斷要連的 u, v 是不是在同個連通塊就好了！
- 如果 u, v 在同個連通塊，加完之後，會產生矛盾
- 否則加上去之後，還是一張二分圖

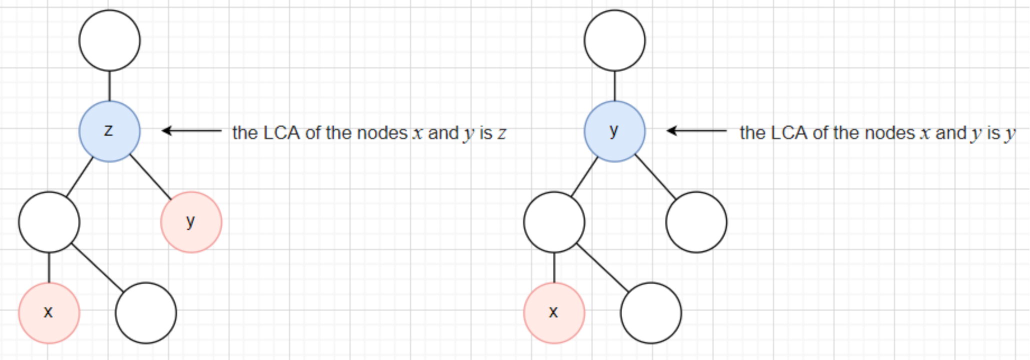


最低共同祖先

甚麼是最低共同祖先？

- 最低共同祖先，又叫 LCA (Lowest Common Ancestor)
- 找 a 和 b 的 LCA，如果 a 是 b 的祖先，那 a 就是他們的 LCA
- 否則就是找他們兩個的祖先中，深度最深的那個點

甚麼是最低共同祖先？


z
the LCA of the nodes x and y is z
y
x
y
the LCA of the nodes x and y is y
x

最低共同祖先

- 很簡單 R
- 就一步一步走嘛
- 樹上走路誰不會 R

最低共同祖先

- 很簡單 R
- 就一步一步走嘛
- 樹上走路誰不會 R
- 對 R... 好水喔　然後你就 TLE 了

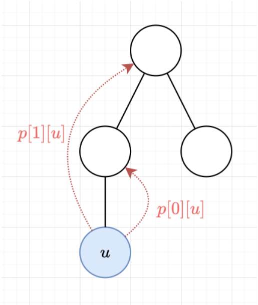
最低共同祖先

- 剛剛一步一步往上走的方式太慢了
- 最糟的狀況下，我們從根開始，會往上走整棵樹的高度欸 QwQ

最低共同祖先

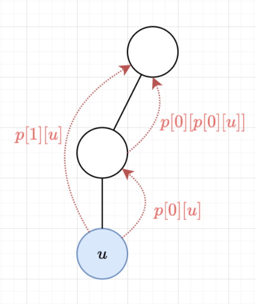
- 剛剛一步一步往上走的方式太慢了
- 最糟的狀況下，我們從根開始，會往上走整棵樹的高度欸 QwQ
- 所以你不能用走的，要用跳的
- 有一個東西叫**倍增法**

- 我們用二進位的概念來想
- 對每個點 u 開一個陣列 $p[i][u]$，代表從點 u 往上跳 2^i 條邊會到的節點



倍增法往上跳

- 要計算這個，我們可以枚舉 $i = 0 \sim \lfloor \log_2 n \rfloor$
- 依序去轉移每個點往上跳的答案
- 轉移式：$p[i][u] = p[i-1][p[i-1][u]]$
- u 往上跳 2^i 會是 u 往上跳 2^{i-1} 再往上跳 2^{i-1}

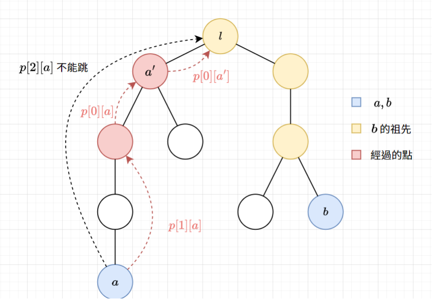


倍增法找 LCA

- 有了剛剛的表，現在來找 a 和 b 的 LCA owo
- 想法是我們可以把 a 往上跳
- 既然要知道 a, b 兩個人的 LCA
- 那其實我們只要一路去判斷 a 的祖先 v 是不是 b 的祖先就好了
- 判斷祖先可以用剛剛講到的樹壓平來處理 ><

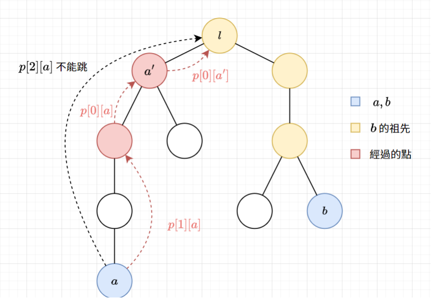
倍增法找 LCA

- 只要 a 往上跳 2^i 格不是 b 的祖先，就一直往上跳
- 之後 LCA 就會是 $p[0][a]$ 了

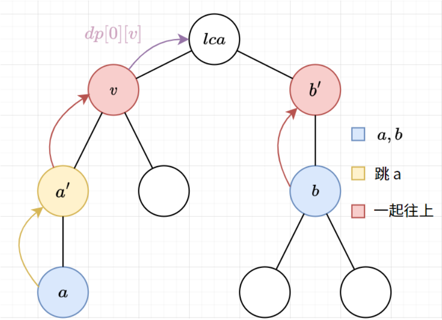


倍增法找 LCA

- 只要 a 往上跳 2^i 格不是 b 的祖先，就一直往上跳
- 之後 LCA 就會是 $p[0][a]$ 了
- 預處理 $O(n\log n)$、單筆詢問 $O(\log n)$

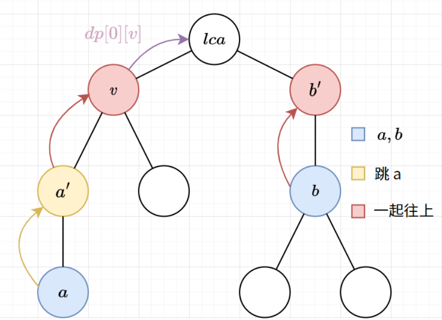


- 有另一種寫法



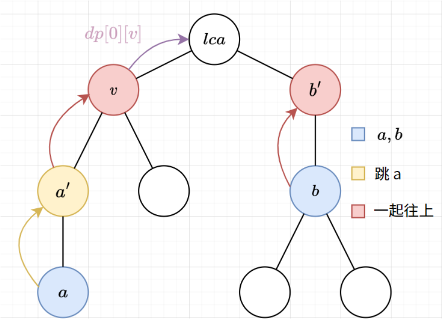
倍增法找 LCA

- 有另一種寫法
- 我們先把 a 和 b 拉到同一個高度
- 讓 a 和 b 同時一起往上跳
- 只要不會是同個點就往上跳



倍增法找 LCA

- 有另一種寫法
- 我們先把 a 和 b 拉到同一個高度
- 讓 a 和 b 同時一起往上跳
- 只要不會是同個點就往上跳
- 最後我們會找到的點 v 是他們 LCA 的子節點
- 所以 LCA 是 v 的父節點 $dp[0][v]$



倍增法找 LCA

- 有第三種寫法和第四種寫法

倍增法找 LCA

- 有第三種寫法和第四種寫法
- 樹壓平 + RMQ 或輕重鍊剖分 (HLD)
- 這些東西今天不會講，因為會用到資料結構的東西
- 有興趣的自己上網查 XD

喵喵喵喵喵

現在來看一些可以寫的例題 ><

這是一題很裸很裸的題目 ><

CSES Company Queries II

找 LCA

樹上兩點距離

TIOJ 2172 物種演化 (Evolution)

給你一棵樹，接下來詢問 a, b 的距離

樹上兩點距離

TIOJ 2172 物種演化 (Evolution)

給你一棵樹，接下來詢問 a, b 的距離

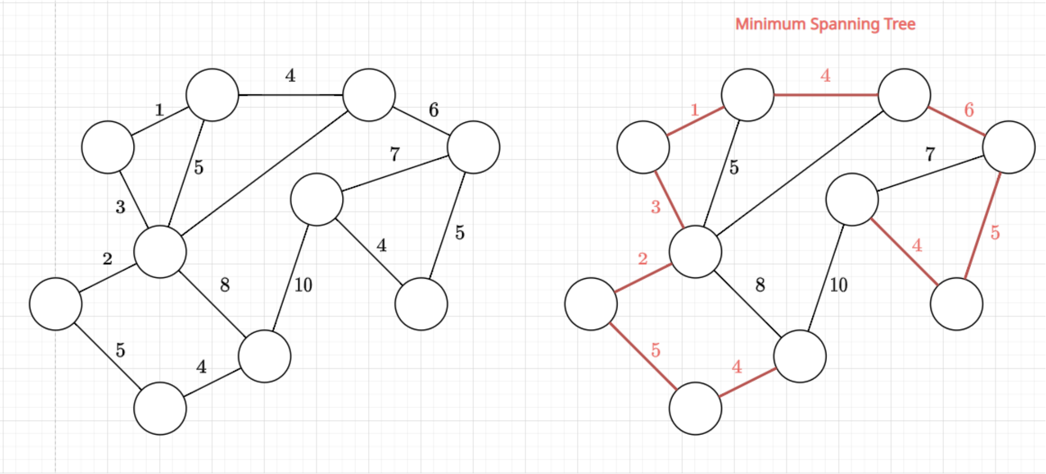
- 兩點的距離其實就是 $dep[a] + dep[b] - 2 \cdot dep[lca(a, b)]$

最小生成樹

甚麼是最小生成樹？

- 最小生成樹，又叫 MST（Minimum Spanning Tree）
- 給你一張圖，從圖上找一些邊，讓它是一棵樹並且使邊權總合最小
- 可能不唯一
- 兩種演算法：Kruskal’s Algorithm、Prim’s Algorithm

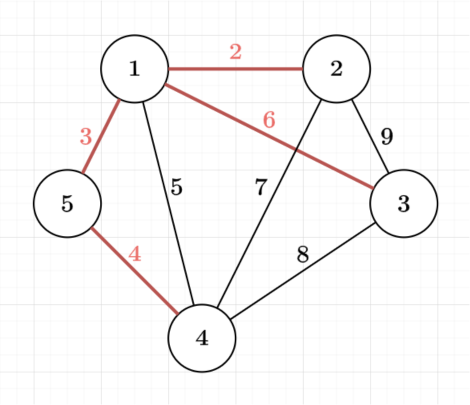
最小生成樹



Kruskal's Algorithm

- 把邊權由小排到大
- 從最小的開始，把該條邊加進樹
- 並查集維護：如果邊所連接的兩點祖先相同，就會產生環
- 如果會形成環，就略過它
- 它算是一種 greedy
- 時間複雜度 $O(|E| \log |E|)$

Kruskal's Algorithm

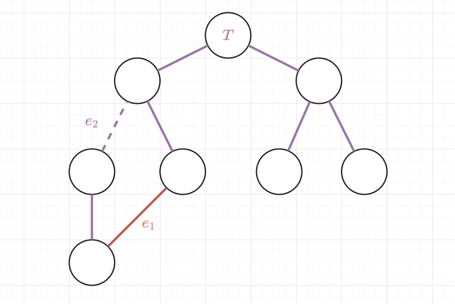


Sorted Edges:
{1,2}, {1,5}, {4,5}, {1,4}
{1,3}, {2,4}, {3,4}, {2,3}

1. add {1,2}
2. add {1,5}
3. add {4,5}
4. ~~add {1,4}~~ (cycle!)
5. add {1,3}
6. ~~add {2,4}~~ (cycle!)
7. ~~add {3,4}~~ (cycle!)
8. ~~add {2,3}~~ (cycle!)

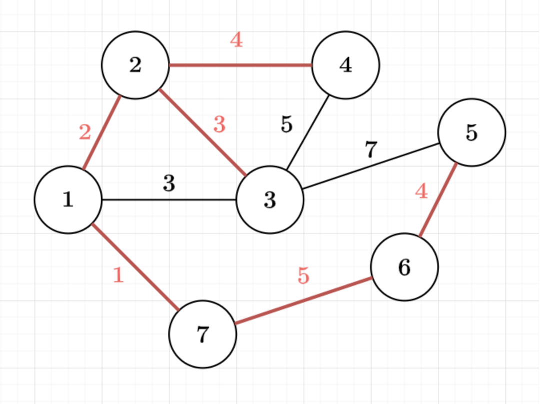
Kruskal's Algorithm 證明

令 Kruskal 算法得出的生成樹為 K，另一棵生成樹為 T。
按照 Kruskal 算法加邊的順序，找到第一條不在 T 裡的邊 e_1，把 e_1 加到 T 裡必定會形成環，環上必定會有一條不在 K 中的邊 e_2（一定只有一條），因為 Kruskal 算法先選了 e_1，代表 $w(e_1) \leq w(e_2)$。
然後再把 e_2 拔掉，可以得到權重比原本的 T 還要小的生成樹，所以一直重複上述動作，就可以把 T 中權重較大的邊替換掉，最後 T 就會 $= K$，K 就會是最小生成樹



Prim's Algorithm

- 起點：隨便一個點，然後把它加到樹上
- 每次選樹上的所有點連出去的邊中權重最小的
- 然後把點跟邊加到樹裡面，最後就會得到 MST
- 時間複雜度 $O(|V| + |E| \log |V|)$
- 稠密圖可以在 $O(|V|^2)$ 的時間完成，會比 Kruskal 快



(pq 按照走到 u 的邊權由小到大排)

1. push 1, $\sum w = 0$, pq = {7,2,3}
2. pop 7 , $\sum w = 1$, pq = {2,3,6}
3. pop 2 , $\sum w = 3$, pq = {3,4,6}
4. pop 3 , $\sum w = 6$, pq = {4,6,5}
5. pop 4 , $\sum w = 10$, pq = {6,5}
6. pop 6 , $\sum w = 15$, pq = {5}
7. pop 5 , $\sum w = 19$

Prim's Algorithm

實作細節:

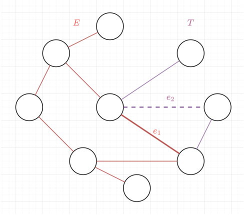
- 與 Dijkstra 的寫法很像
- priority queue 存 {邊權, 點}
- 稠密圖可以直接做 $|V|$ 次，每次枚舉與當前點集相鄰且邊權最小的點

Prim's Algorithm 證明

證明：走的每一步都在 MST 中
假設第 k 步成立，這個時候的邊集為 E，且包含在 MST 中
令 MST 為 T，接下來要加入的邊為 e_1。

1. 如果 e_1 在 T 中，那就會成立。
2. 否則將 e_1 加入 T 中，加入後必定會形成環，環上必定有一條不在 E 中的邊 e_2。

e_1 不可能大於 e_2，因為算法先選了 e_1。
e_1 不可能小於 e_2，因為如果 e_1 小於 e_2，$T + e_1 - e_2$ 會比 MST 更小。
因此 $e_1 = e_2$，$T + e_1 - e_2$ 是 MST，E 包含在其中。



練習

CSES Road Reparation

找最小生成樹

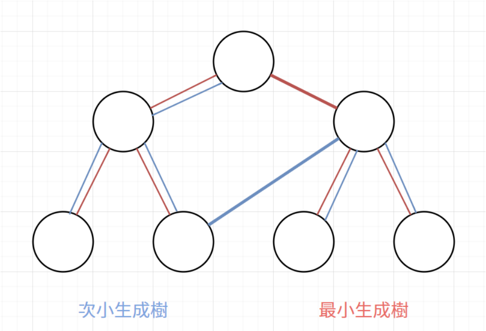
Cycle Property

- 定義：
  假設 T 是一個帶權的最小生成樹，加入一條邊不在 T 上的邊 e 以後，則會形成一個環，並且 e 會比環上任一條邊的邊權都大。(T 一定不會選環上權重最大的邊)
- 證明：
  我們可以利用反證法，如果環上存在一條邊 d 比 e 大，那我們可以把 d 拔掉，換成 e，那麼這樣得到的生成樹會比 T 還要小，矛盾。

Cut Property

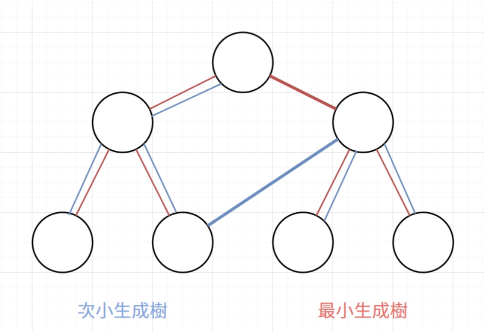
- 定義：
  若將圖 G 分成兩個連通塊，那我們找到的最小生成樹中，必存在一條連接兩個連通塊的邊，而且這條邊是所有可以連接兩個連通塊的邊中最小的。(T 一定會選割上權重最小的邊)
- 證明：
  一樣可以用反證法，我覺得你們可以自己證。

- 找非嚴格次小生成樹。



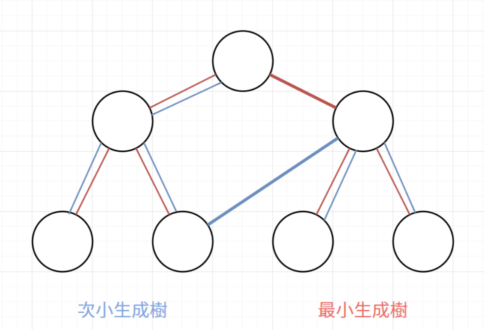
應用

- 找非嚴格次小生成樹。
- 根據 `cycle property` 我們可以發現，次小生成樹跟最小生成樹相差一條邊



應用

- 找非嚴格次小生成樹。
- 根據 cycle property 我們可以發現，次小生成樹跟最小生成樹相差一條邊
- 既然我們知道只差一條邊了，假設那條邊連接 u 和 v，那就枚舉不在 MST 上的邊
- 讓這些邊去跟在 MST 中 u 到 v 路徑上的邊做替換，替換掉最大的邊最優



非嚴格次小生成樹

- 可以用前面講過的倍增法去維護兩點路徑間的最大邊
- 維護一個 $mx[i][u]$ 表示 u 往上跳 2^i 條邊的最大邊
- 在轉移時順便維護這個值
- 轉移式：

$$mx[i][u] = \max(mx[i-1][u], mx[i-1][p[i-1][u]])$$