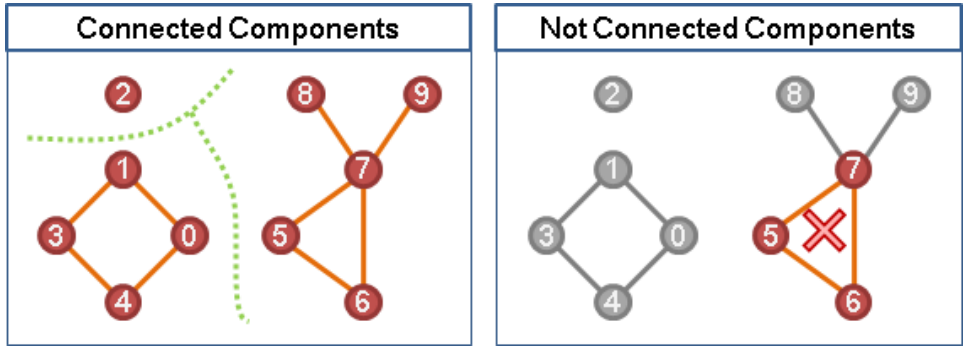

# Graph Algorithms
# Connected Component

for Sprout 2014 by Chin-Huang Lin

Sprout

# 連通元件/分量/單元

- 一張圖任兩點之間皆連通，稱為連通圖 (connected graph)
- 一張圖 $G$ 有很多子圖，如果一個子圖 $G'$ 是連通的，我們稱之為連通元件 (connected component)
- 如果一個連通元件滿足「加上任意一個其他的點就不再連通」，則稱這樣的連通元件是「極大的」 (maximal)
- 如未特別提及，通常默認討論極大連通元件

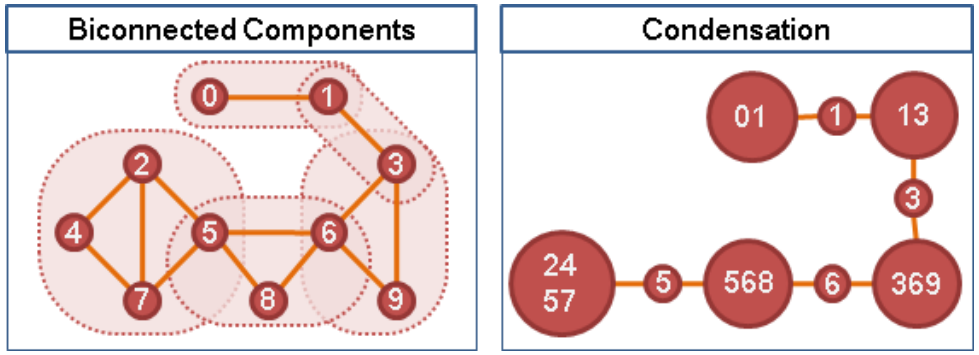


# 連通元件/分量/單元

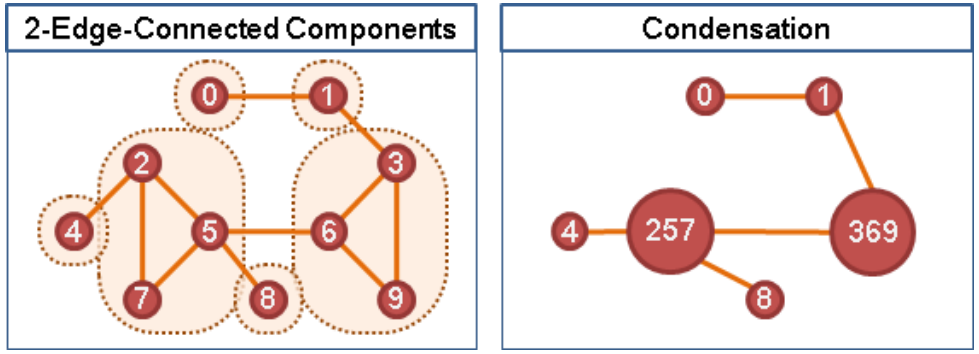
- 根據不同的連通要求，有不同的連通分量
- 如果連通性具有傳遞性，那麼一張圖必定恰可以視為許多個極大連通元件的聯集
  - A, B 連通、B, C 連通，那麼 A, C 連通，因此任兩連通元件交集為空
  - 一個點本身就是一個連通元件，所以每個點都在連通元件裡
  - ex. 無向圖的一般連通中，任意圖都可以看成很多連通塊的聯集
- 我們在意的是，一張圖如何被分成連通元件

# 雙連通元件 (bi-connected components)

- 無向圖中，除了一般連通性，我們還可以要求
  - 點的雙連通：以邊為主體，所以同一點可能在兩個集合內

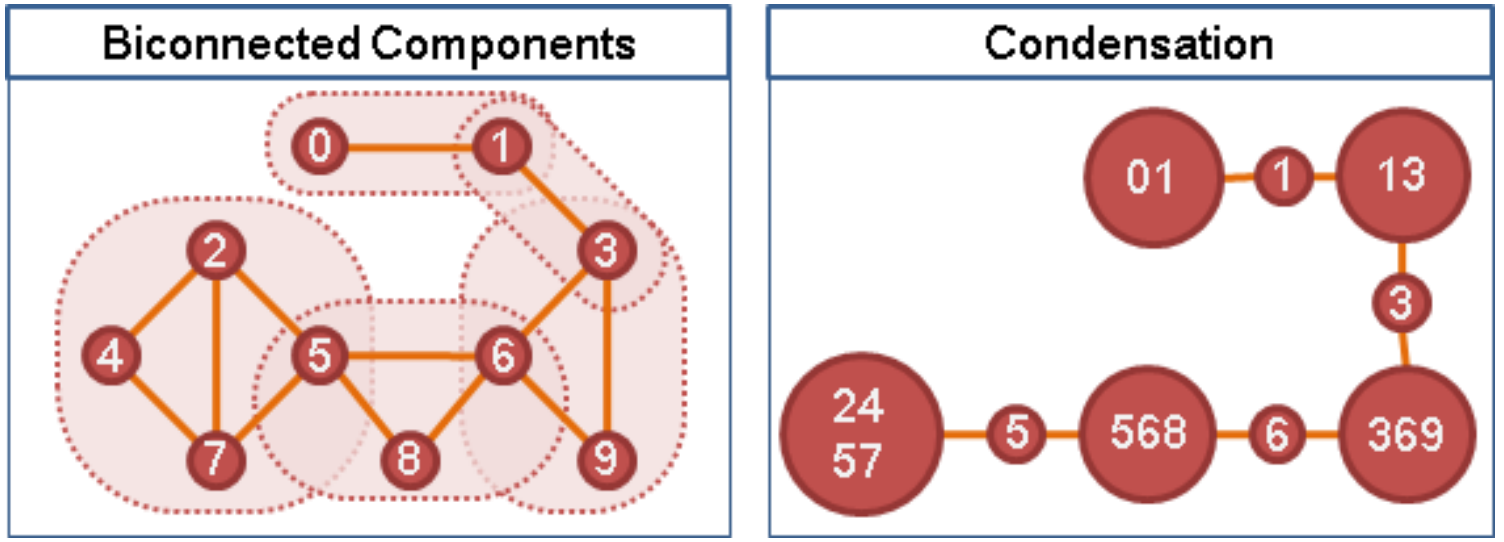



  - 邊的雙連通：以點為主體，所以可能有邊不在任何集合內



# 雙連通元件 (bi-connected components)

- 無向圖中，除了一般連通性，我們還可以要求
  - 點的雙連通：以邊為主體，所以同一點可能在兩個集合內



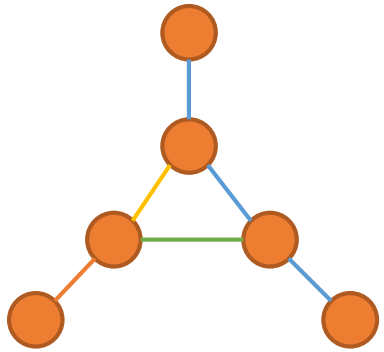

→一個連通元件中不能有任何割點

←沒有割點的連通元件就雙連通了

↔原圖上的割點恰好分割出所有雙連通元件！

# 點雙連通元件

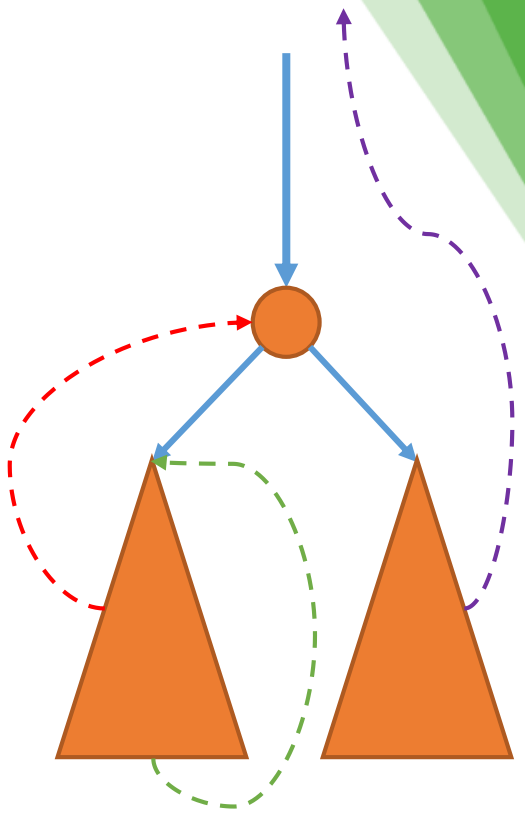
- 想法一：土法煉鋼
  - 先進行一次 `Tarjan's algorithm` 找出所有割點
  - 將所有的割點都標記為不可通行
  - 試圖對「邊」進行 `flood fill`
    - 兩條邊可以互通當且僅當有共同的端點
    - 標記為不可通行的點則失去讓某些相臨邊互通的功能
    - 問題：哪些邊會因為割點而不能互通？



# 點雙連通元件

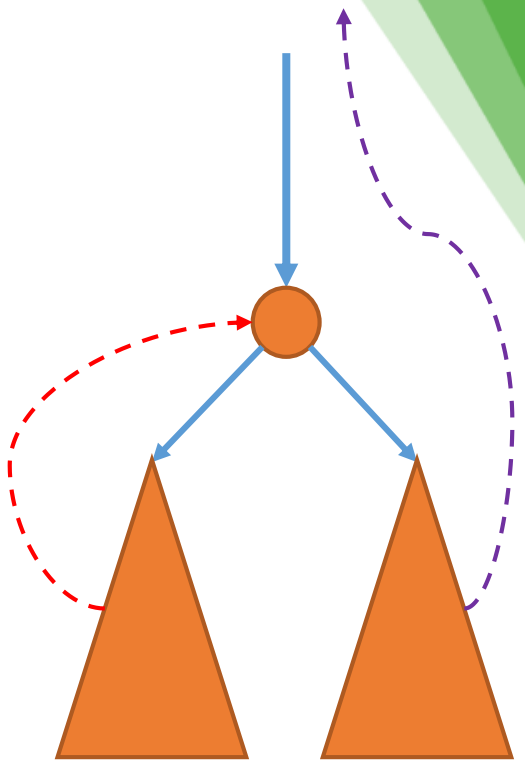
- 想法一：土法煉鋼
  - $p$ 以下的邊有三種（皆為）
    1. 不透過 $p$ 的情況下，可以回到 $p$ 以上祖先的邊
    2. 不透過 $p$ 的情況下，可以回到 $p$ 的邊
    3. 不透過 $p$ 的情況下，回不到 $p$ 的邊

  - 3 代表 $p$ 以下還存在割點，或者單一葉節點
  - 為了簡單起見，先考慮 $p$ 以下沒有割點的情形



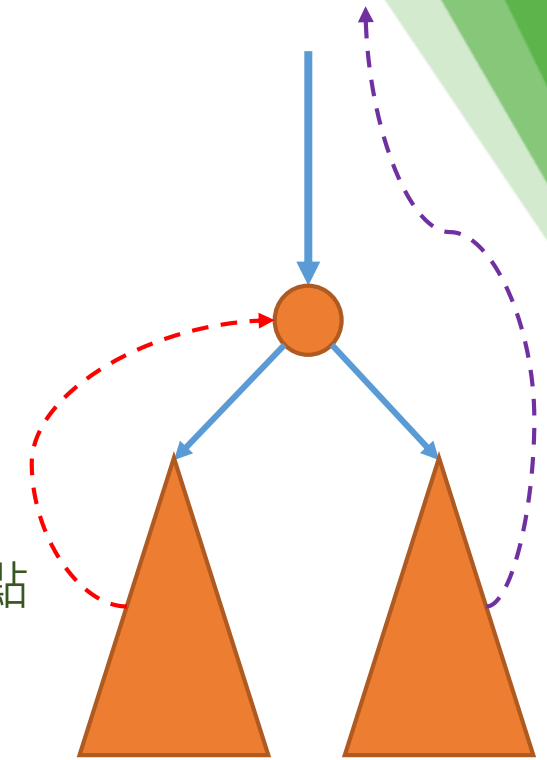
# 點雙連通元件

- 想法一：土法煉鋼
  - 根據我們在 connectivity 討論過的結論……
    - 假如 $p$ 有 $k$ 個兒子，分別形成子樹 $T_1, T_2, \ldots, T_k$
    - 對於任意 $T_i$，子樹中所有的邊，都會是同一種邊
    - 拔除 $p$ 之後……
      - 如果 $T_i$, $T_j$ 都屬於 1，那麼這兩棵子樹的邊都互通，但是也可能和其他不屬於任何 $T_1 \sim T_k$ 的邊互通
      - 如果 $T_i$, $T_j$ 都屬於 2，那麼這兩棵子樹的邊都不會互通
      - 如果 $T_i$, 屬於 1，$T_j$ 屬於 2，那兩棵子樹的邊都不會互通
    - 結論：
      - 屬於 1 的子樹全部都在同一個雙連通元件
      - 屬於 2 的子樹自己會形成一個雙連通元件



# 點雙連通元件

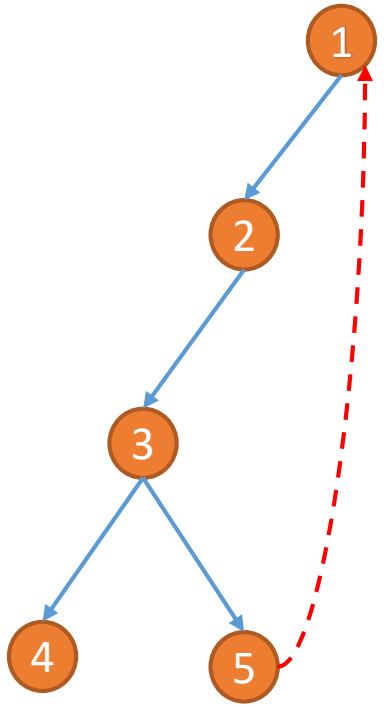
- 想法一：土法煉鋼
  - 結論：
    - 屬於 1 的子樹全部都在同一個雙連通元件
    - 屬於 2 的子樹自己會形成一個雙連通元件
  - 屬於 3 的子樹呢？
    - 因為先不考慮有子代有割點的情形，所以必定是單一節點
    - 不只如此，還必定只有一條邊
    - 這條邊自成一個雙連通元件
  - 屬於 2, 3 的子樹們所在的雙連通元件已經確定
  - 屬於 1 的子樹們所在的雙連通元件尚未確定
  - 所以乾脆把 1 的子樹們留給祖先決定即可！



# 點雙連通元件

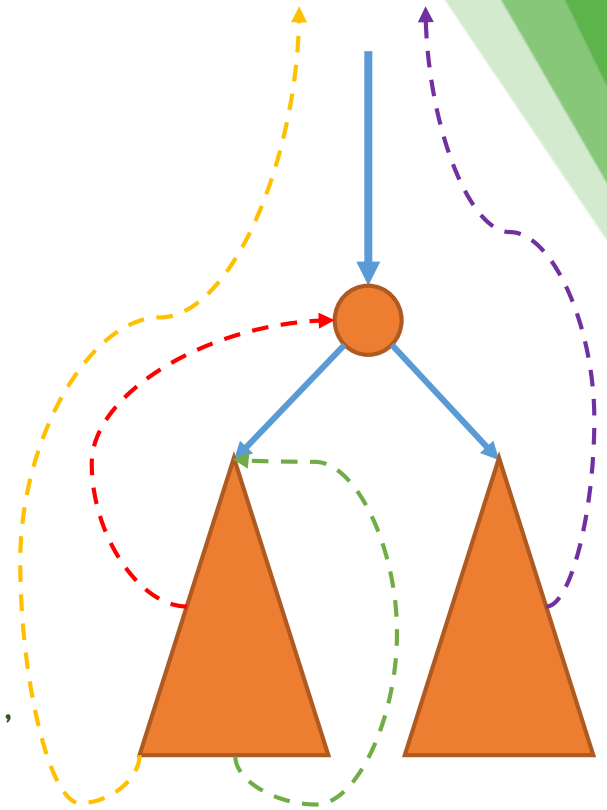
- 對 3 而言，4 是情況 3
  - 先把 4 處理完然後拔掉
- 對 3 而言，5 是情況 1
  - 留著以後處理
- 對 2 而言，3 是情況 1
  - 留著以後處理
- 對 1 而言，2 是情況 2
  - 處理完以後拔掉

→ 狀況 1 遲早有一天會變成狀況 2



# 點雙連通元件

- 想法一：土法煉鋼
  - 加入 $p$ 以下有割點的情形呢？
    - 先考慮 $p$ 底下只有一個割點 $q$ 的情形
    - $q$ 的子代形成的子樹，相對於 $q$ 是
      - 2, 3 的情形自己會獨立成一個連通元件
      - 1 的情形則可以和 $p$ 的某些子代形成的子樹互通
        - 互通的方式則看這些邊與 $p$ 的關係
    - 結論：把 $q$ 的子代中 2, 3 的情形者先遞迴處理完，其餘部份視為 $p$ 的子代處理即可



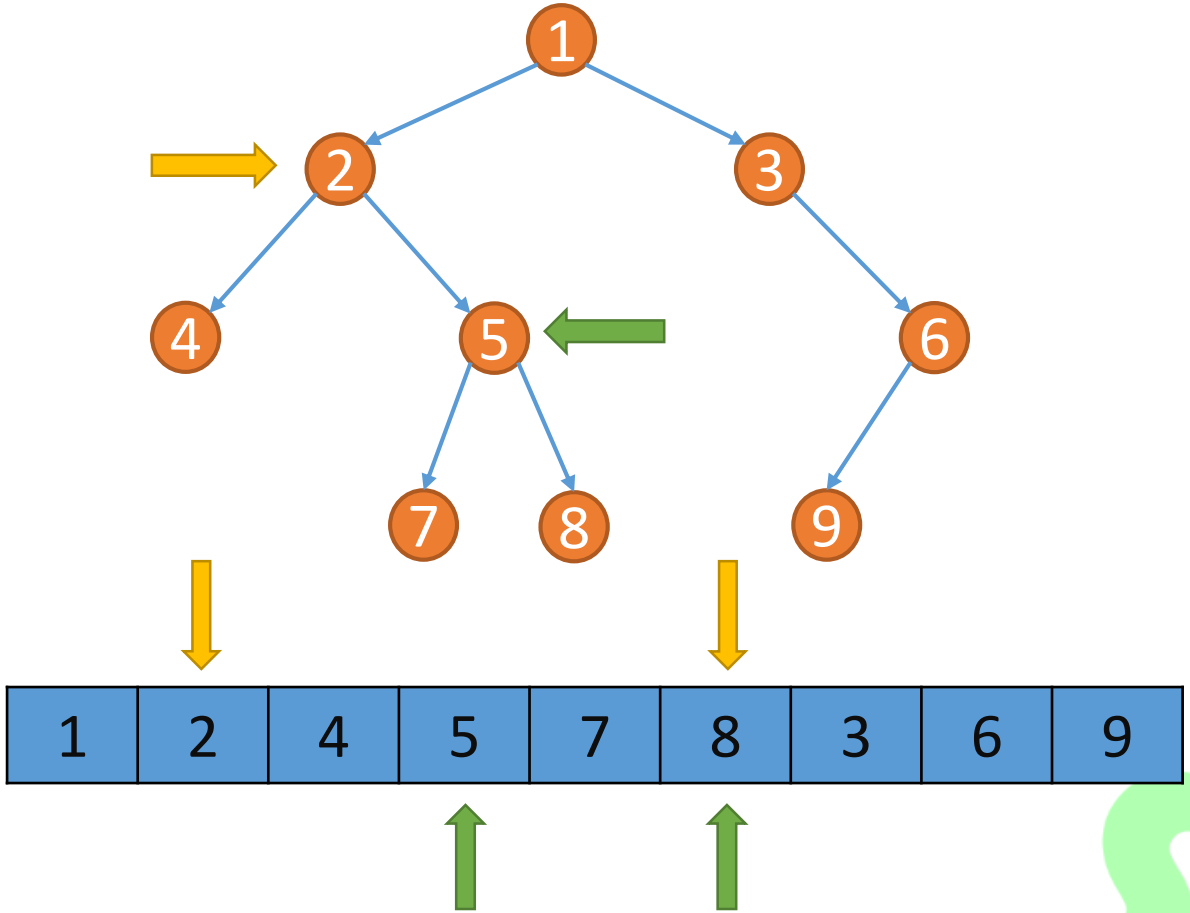
# 點雙連通元件

- 想法一：土法煉鋼
  1. 找出所有割點
  2. 對於所有割點，依據深度由深到淺進行下列步驟：
     a. 假設當前節點是 $p$，有 $k$ 個兒子 $s_1 \sim s_k$
     b. 對於每個兒子 $s_i$：
        1) 如果 $low(s_i) < lv(p)$，則先擱置不理
        2) 如果 $low(s_i) \geq lv(p)$，則對 $s_i$ 進行遍歷，把拜訪到的邊都設為同一個雙連通元件，然後從圖上拔掉除
  3. END
- 注意：拔除邊不要順便把點給拔了！主體是邊不是點。

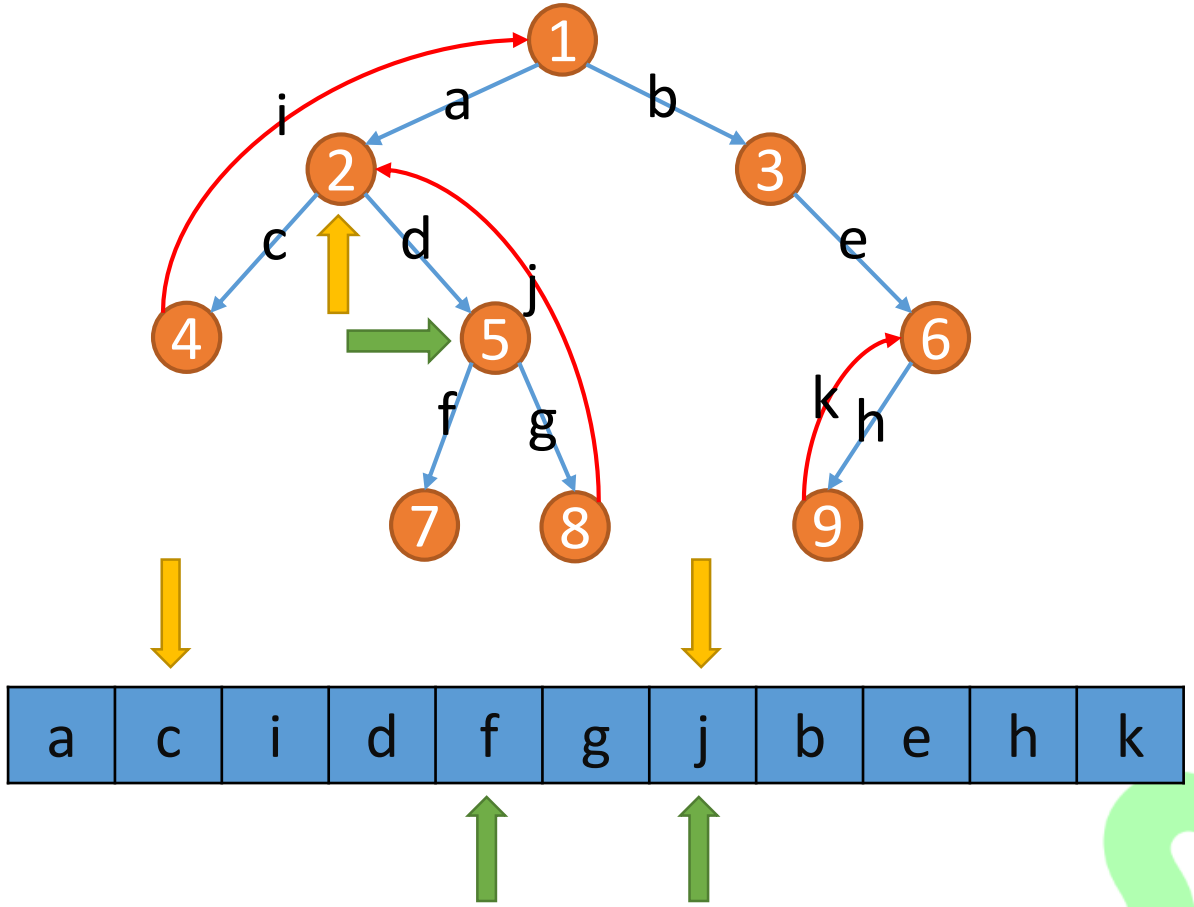
# 點雙連通元件

- 想法二：一鼓作氣
  - 找割點的時候就已經判斷過 $low$ 函數的各種情形了，為什麼要做兩次呢？
  - 分成多次拜訪，還要實做拔邊，增添麻煩
  - DFS 過程中，同一棵子樹內的元素會被連續拜訪到
    - 根據拜訪順序，我們可以獲得一個序列
    - 所有子樹可以對應到序列中連續的一個區間
    - 把樹「壓平」，轉換成一維的結構！

# 樹 → 區間

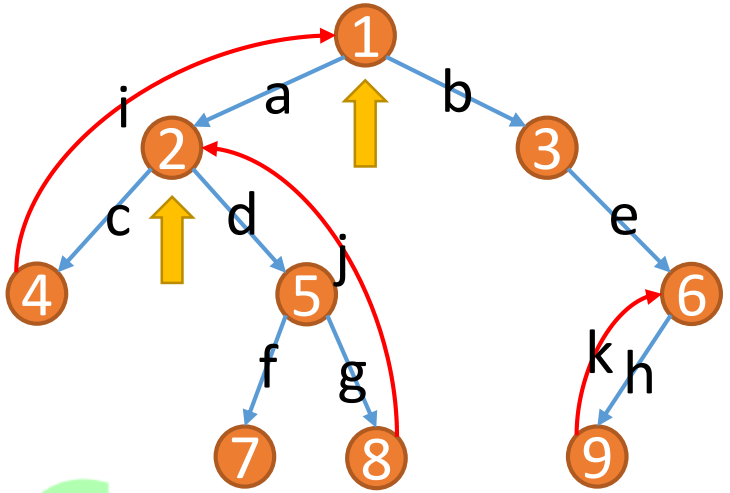


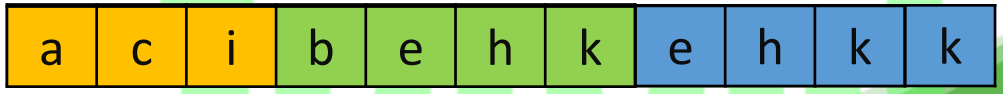
# 樹 → 區間



# 點雙連通元件

- 想法二：一鼓作氣
  - 考慮一個子代同時有狀況 1，2，3 的節點 $p$
  - $p$ 形成的子樹對應到的區間會是狀況 1，2，3 的邊交錯而成的區間
  - 遞迴完子代後，有些元素會消失
  - 接著我們可以把狀況 2，3 的邊處理掉
  - 剩下的部份就以後留給祖先們處理即可
    - 反正一定會有某個祖先覺得他們是狀況 2
  - 既然區間都一定連續，我們可以用一個 stack 來處理！

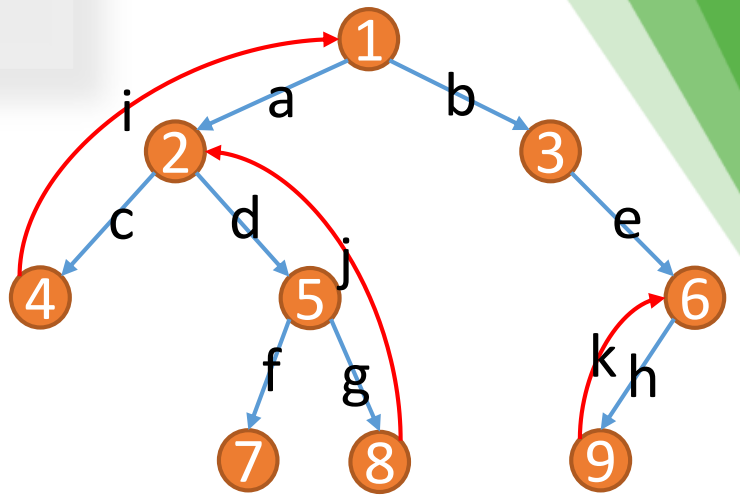


# Tarjan's algorithm

```
1   Stack S;
2   int bcnt := 0;
3
4   dfs(Vertex now, int curlv){
5       low[now] := curlv;
6       lv[now]  := curlv;
7       mark now as visited;
8
9       for all edges e_i as {now, other}
10          if( e_i is visited ) continue;
11          else {
12              set e_i as visited;
13              S.push( {now, other} );
14          }
15          if( other is visited ) low[now] := min (low[now], lv[other]);
16          else {
17              dfs(other, curlv+1);
18              low[now] := min (low[now], low[other]);
19              //case 2 or case 3
20              if( low[other] >= lv[now] ){
21                  do {
22                      Edge top := S.pop();
23                      bcc[top] := bcnt;
24                  }while( top != e_i );
25                  bcnt := bcnt + 1;
26              }
27          }
28  }
```

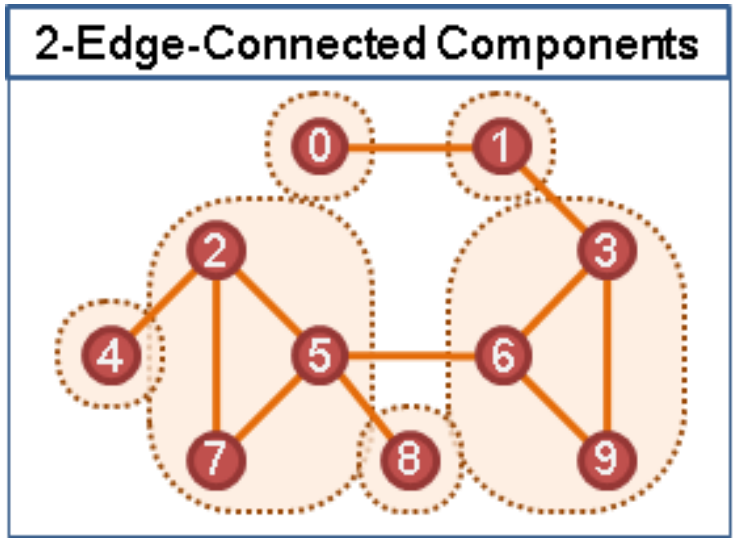
# Demo time

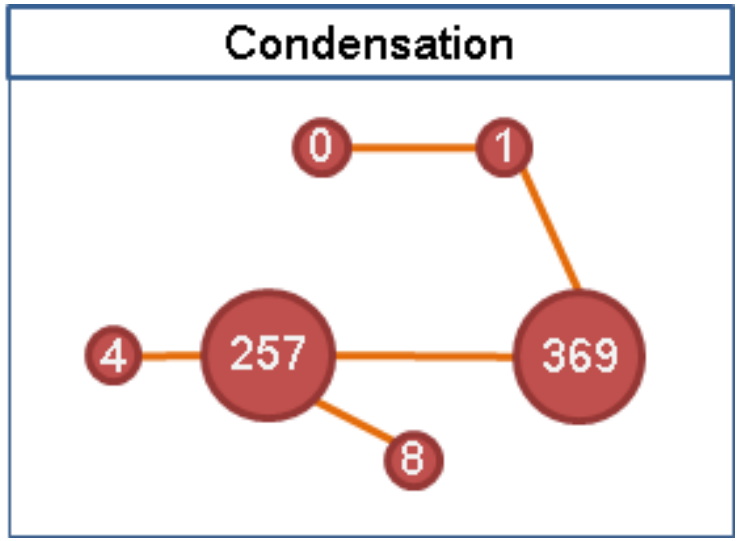



- DFS 1
  - Push(a) → a
  - DFS 2
  - Push(c) → a c
    - DFS 4
    - Push (i) → a c i
  - Push(d) → a c i d
    - DFS 5
    - Push(f) → a c i d f
      - DFS 7
    - Pop() until f → a c i d — f
    - Push(g) → a c i d g
      - DFS(8)
      - Push(j) → a c i d g j — d g j
  - Pop() until d → a c i
- Pop() until a → — a c i
- Push(b) → b
  - Push(e) → b e
    - DFS 6
    - Push(h) → b e h
      - DFS 9
      - Push(k) → b e h k
    - Pop() until h → b e — h k
  - Pop() until e → b — e
- Pop() until b → — b

# 邊雙連通元件

- 無向圖中，除了一般連通性，我們還可以要求
  - 邊的雙連通：以點為主體，所以可能有邊不在兩個集合內




→一個連通元件中不能有任何橋

←沒有橋的連通元件就雙連通了

↔原圖上的橋恰好分割出所有雙連通元件！

# 邊雙連通元件

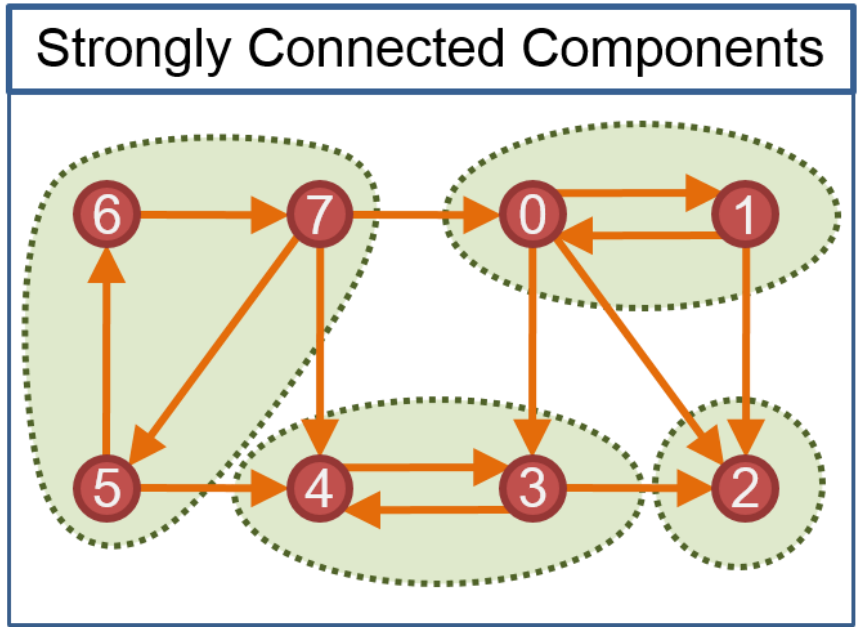
- 想法一：土法煉鋼
  - 先進行一次 Tarjan's algorithm 找出所有橋
  - 將所有的割點都標記為不可通行
  - 試圖對「點」進行 flood fill
    - 對點 flood fill 超簡單，什麼問題都沒有
  - 太棒了，土法煉鋼大成功

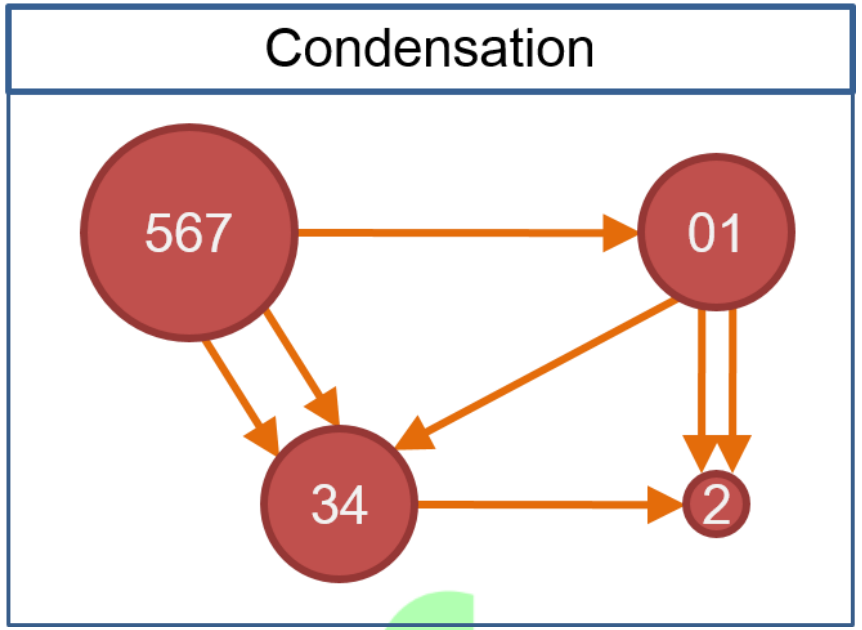
# 邊雙連通元件

- 想法二：一鼓作氣
  - 一樣先遞迴把子代橋割出來的雙連通元件都先拔除
  - 如果 $p-q$ 是一條橋，那麼 $q$ 以下剩下的圖都屬於同一個連通元件
  - 實做起來依舊類似 `Tarjan's algorithm`，只是主體變成點
    - 細節就留給讀者思考了 `:D`

# 強連通元件 (strongly connected component)

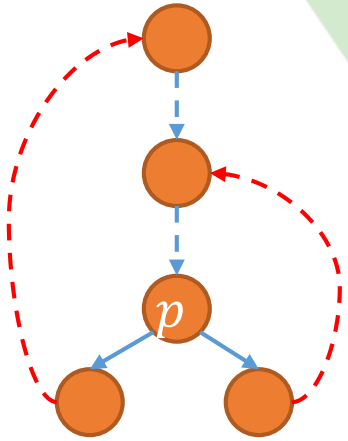
- 一張有向圖任意兩點都互通
  - 以點為主體，所以可能有邊不在任何集合內



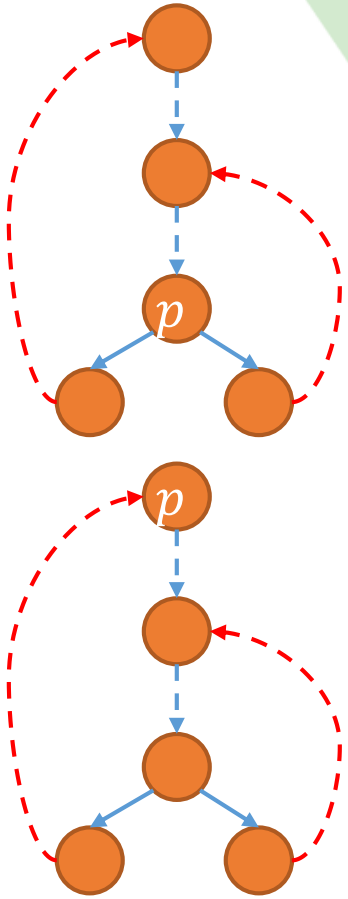
# 強連通元件

- 想法一：再探 Tarjan's algorithm
  - 在 DFS 樹上，一個點 $p$ 和「可以回到 $p$ 的子代」及「$p$ 走得到的祖先」形成一個 SCC
  - 有向圖比無向圖多了兩種邊：forward edge, cross edge
  - Tree edge 保證了祖先可以連到子孫，問題只剩下子孫連到祖先
  - Forward edge 對子孫連到祖先毫無幫助
    - 我們只需要考慮 back edge 和 cross edge！



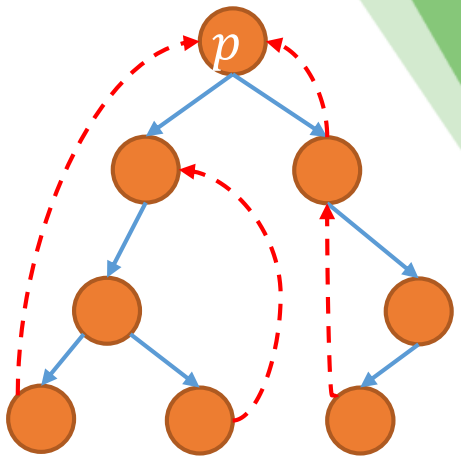
# 強連通元件

- 想法一：再探 Tarjan's algorithm
  - 在 DFS 樹上，一個點 $p$ 和「可以回到 $p$ 的子代」及「$p$ 走得到的祖先」形成一個 SCC
    - 對於 $p$ 而言，如果發現有子代可以回到更老的祖先 $q$，那麼至少 $q$ 也在 $p$ 所在的 SCC 內
    - 整個 SCC 在 $q$ 完全拜訪結束前沒辦法被確定
    - 不如只考慮一個 SCC 內深度最淺的節點！
  - 在 DFS 樹上，一個子代最多只能回到自己的點 $p$ 和「可以回到 $p$ 的子代」形成一個 SCC

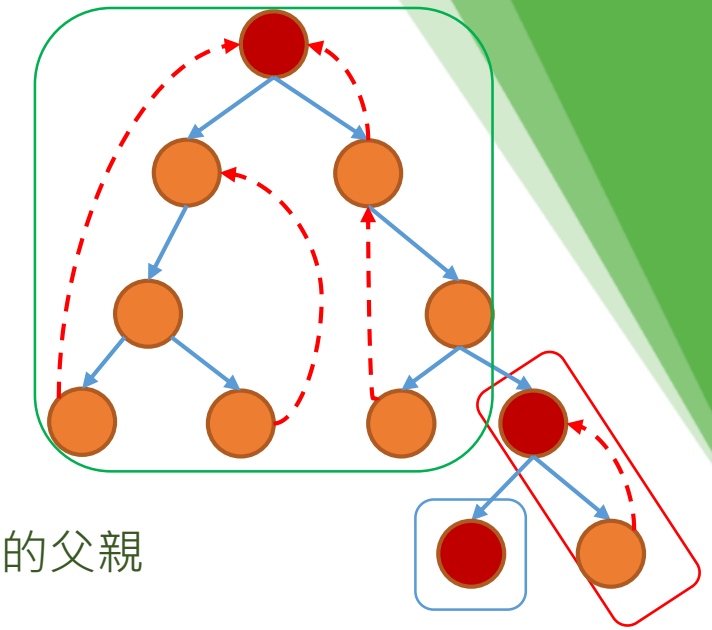


# 強連通元件

- 想法一：再探 Tarjan's algorithm
  - 怎樣才會是一個「子代最多只能回到自己」的點？
  - 我們只需要考慮 back edge 和 cross edge！
  - 如果只有 back edge……
    - 直接看 $low$ 函數即可！
      - $low(p) < lv(p)$ …… $p$ 絕對不是我們要找的點
      - $low(p) = lv(p)$ …… $p$ 正是我們要找的點



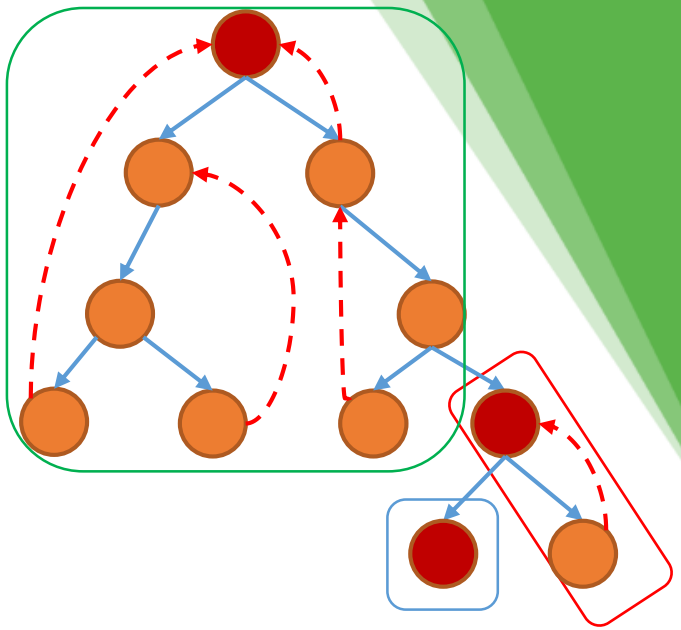
# 強連通元件



- 想法一：再探 Tarjan's algorithm
  - 如果只有 back edge……
  - 找完所有目標點後，樹被目標點「切」成很多塊
  - 結論：形成的每一塊恰形成一個 SCC
    - 每一塊內的元素必定兩兩可以互通 (連通性)
      - 由找目標點的判定原則可知，目標點外任一點都能回到自己的父親
      - 於是任一點都能回到每塊的 root，即目標點
      - 目標點為 root，可以到達樹中任意點
      - 於是任一點皆可透過走回 root 走到任一其他點
    - 任兩不同塊內的元素不可互通 (最大性)
      - 由連通性的傳遞性可知，若兩不同塊中有任何點對可以互通，則兩塊內所有點對都可以互通
      - 此與目標點的判定原則矛盾

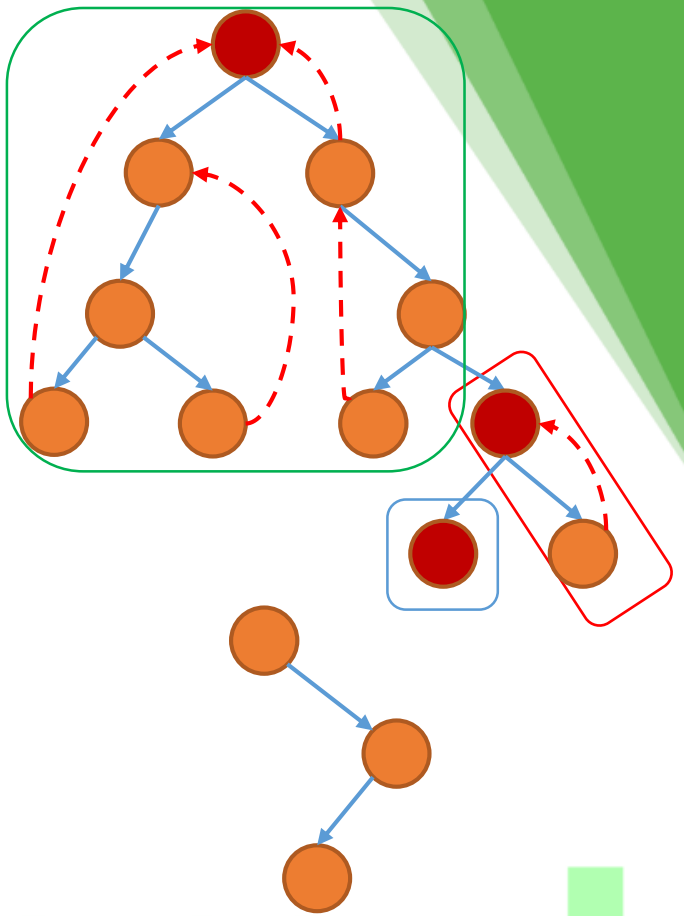
# 強連通元件

- 想法一：再探 Tarjan's algorithm
  - 現在加入 cross edge……
  - 若兩點在只考慮 back edge 時就已經在同一 SCC，加入 cross edge 還是會在同一 SCC
    - 加入 cross edge 只會讓只考慮 back edge 時不同 SCC 的兩點變成在同一個 SCC
    - 問題：cross edge 如何影響兩點的連通性？



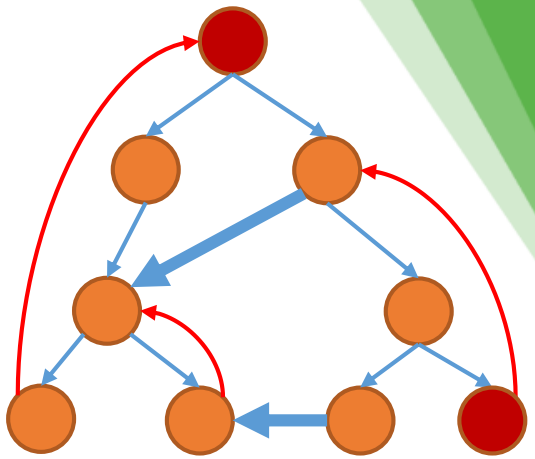
# 強連通元件

- 想法一：再探 Tarjan's algorithm
  - 根據連通性的傳遞性，已經確定在同一個 SCC 內的元素在連通性上是等價的
    - 可以把等價的點都看成同一個點
  - 縮點後圖上不存在 back edge，形成一棵純粹的樹加上若干 cross edge
  - 如果在新樹上 cross edge 通往祖先，則形同新樹上的 back edge
  - 否則形同新樹上的 cross edge



# Cross edge

- 想法一：再探 Tarjan's algorithm
  - 形成 back edge 後又可以將樹上某些節點合併在一起
    - 可以透過再次縮點縮小問題規模！
  - 如果不形成 back edge，則新圖上只有 tree edge 和 cross edge
  - 有沒有可能某個點只靠著 cross edge 就可以走回自己呢？
  - 不可能！



# Cross edge

- 如果圖上存在一條 $p \to q$ 的 cross edge，代表 DFS 的時間戳記滿足

  離開戳記 $ls(q) <$ 進入戳記 $es(p)$

  否則代表要嘛 $q$ 根本還沒被拜訪過，要嘛 $q$ 是 $p$ 的祖先
- 如果某點 $p$ 可以透過 $p_1, p_2, \dots, p_k$ 走到某個 $p$ 的祖先 $p_k$（$p_k$ 可能是 $p$），則代表

  $ls(p_1) < es(p),\ ls(p_2) < es(p_1), \dots, ls(p_k) < es(p_1)$ …(1)
- 根據時間戳記的定義顯然有

  $ls(p) > es(p)\ \forall p$ …(2)
- 結合 (1)(2)，我們有

  $es(p) > ls(p_1) > es(p_1) > ls(p_2) > \cdots > ls(p_k)$ …(3)
- 然而若 $p_k$ 是 $p$ 的祖先，則必定有 $ls(p_k) \geq ls(p) > es(p)$，與(3)矛盾

# 強連通元件

- 想法一：再探 Tarjan's algorithm
  1. 計算 $low$ 函數，並於 DFS 過程忽略 forward edge 與 cross edge
  2. 如果圖上不存在 back edge，則算法結束，當前所有點對應到的點集合即為所求 (算法一開始每個點對應到的點集合為自己)
  3. 否則找出所有滿足 $low(p) = lv(p)$ 的點 $p$，並按照深度由深到淺依序進行 DFS 遍歷 $p$ 為 root 形成的子樹中未拜訪的點，將所有拜訪到的點縮為一點，並標記為已拜訪
  4. 回到 1.

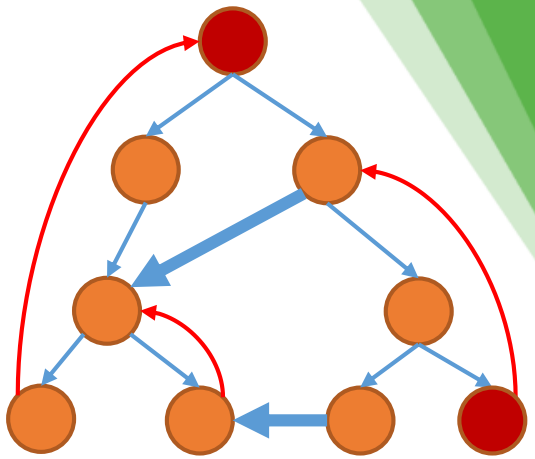
  - 複雜度：最壞情形每次僅拔除一條 cross edge
    - Cross edge 數量不超過 $O(n)$，否則在圖上形成環，由前頁證明可知不可能
    - 最多共 DFS $O(n)$ 次，複雜度為 $O(n(n+m))$

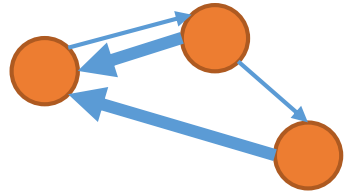

# 強連通元件

- 想法二：一鼓作氣
  - 類似 BCC，我們也可以透過 stack 在 DFS 過程中順便縮點
  - 一進入某點就把該點推入 stack 中，如果拜訪完發現當前點 $p$ 滿足 $low(p) = lv(p)$，則不斷把點從 stack 中 pop 出直到 pop 出 $p$，此時所有 pop 出的點形成一個 SCC
  - 整體算法時空複雜度不變，但編程複雜度低多了！

# 強連通元件

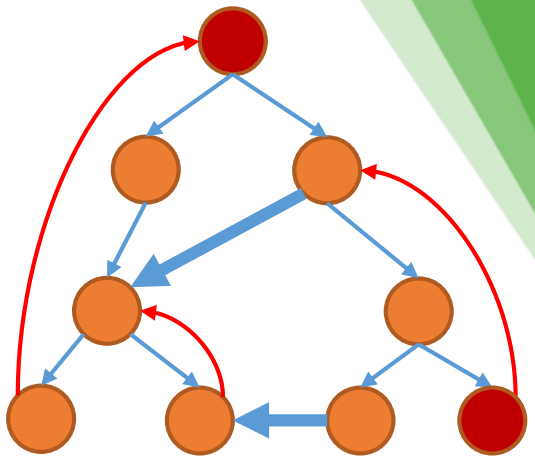
- 想法三：Tarjan's algorithm
  - 想法一二的瓶頸在於，沒辦法「準確」預測一條 cross edge 會不會變成 back edge
  - 根據 cross edge 的特性，圖上的 cross edge 必形成一 DAG
  - 進一步觀察我們會發現，如果 cross edge $(p, q)$ 最後變成 back edge，那麼必定是 $q$ 或者 $q$ 透過 cross edge 走到的某個點 $r$ 早在一開始就是 $p$ 的祖先





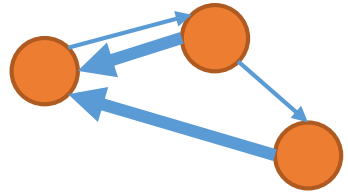
# 強連通元件

- 想法三：Tarjan's algorithm
  - 由於 $p \to q$, $q \to r$ 都是 cross edge：
    - 在 $p$ 透過 cross edge 拜訪 $q$ 的瞬間，$q$ 早就已經被徹底拜訪完了，能不能走到 $r$ 早就確定好了！
    - 能不能透過簡單的方法知道 $q$ 能不能走到這樣的 $r$ 呢？



Robert Tarjan
(photo from http://www.cs.princeton.edu/~ret/)

Of course! I think it's quite easy!



Sprout

# 強連通元件

- 想法三：Tarjan's algorithm
  - 考慮想法二的演算過程……
  - 在拜訪到 $p$ 時，在 stack 中的會是哪些點呢？
    - $p$ 和 $p$ 的祖先們顯然在列
    - 確定與 $p$ 或 $p$ 的某個祖先在同一個 SCC 的點們
  - 已經拜訪過卻不在 stack 中的會是哪些點呢？
    - 確定在不透過 cross edge 的情形下不能回到 $p$ 與 $p$ 的祖先們的點們

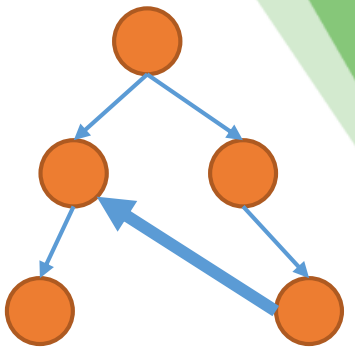
# 強連通元件

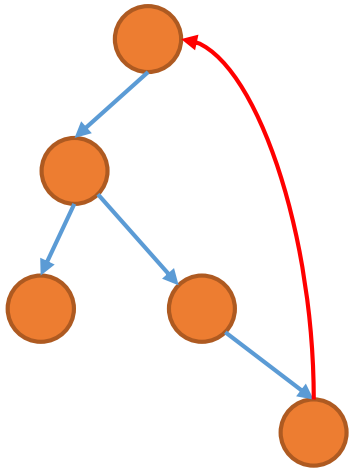
- 想法三：Tarjan's algorithm
  - 假如現在有一條 cross edge 連向一個已經拜訪過的點 $q$：
    - 如果 $q$ 還在 stack 裡，代表透過 $q$ 可以到達 $p$ 或 $p$ 的某個祖先
    - 如果 $q$ 不在 stack 裡，代表 $q$ 的所在 SCC 已經確定了而且不能到達 $p$ 或 $p$ 的任何祖先

    →如果 $q$ 還在 stack，則繼承 $q$ 的連通性，否則可以直接無視此邊！
  - 問題：怎麼好好地描述 $q$ 的連通性呢？

# 強連通元件

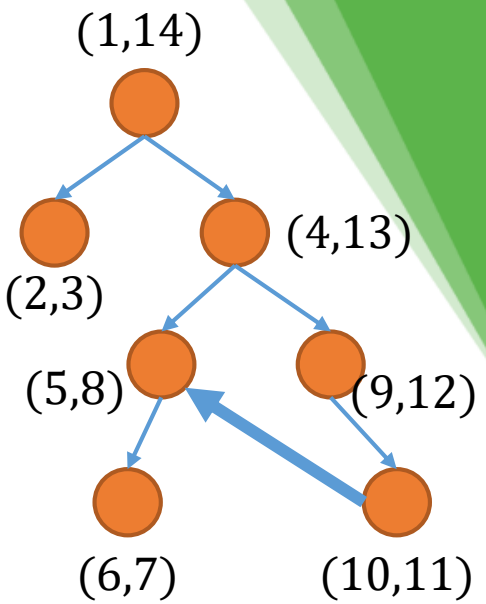
- 想法三：Tarjan's algorithm
  - 以往我們都用 $low$ 函數來表示連通性
  - 在無向圖上不會有 cross edge，以 $lv$ 來決定 $low$ 函數沒有歧義
  - 但在有向圖中，以 $lv$ 來決定 $low$ 函數沒辦法處理 cross edge
    - 我們需要一個能同時反映出祖孫與親戚關係的指標
  - 使用時間戳記取代 $lv$ 函數！





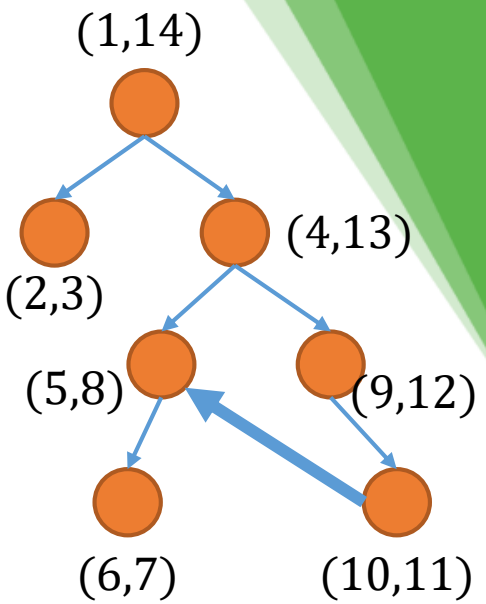
# 強連通元件

- 想法三：Tarjan's algorithm
  - 考慮時間戳記的進入戳記 $es$
    - 如果 $p$ 是 $q$ 的祖先，則必定有 $es(p) < es(q)$
    - 如果 $p$ 是 $q$ 的非直系親戚且先被拜訪，則 $es(p) < es(q)$
    - 若 $p$ 與 $q$ 為非直系親戚，則 $p$ 的祖先中，第一個 $es$ 比 $q$ 還大者即為兩者的最遠共同祖先 (*Longest common ancestor, LCM*)
  - 如果 $p$ 透過 cross edge 連到 $q$，且 $q$ 此時還在 stack 內，那麼至少 $lcm(p,q)$ 到 $p$ 之間的點都和 $q$ 在同一個 SCC 內
    - 因為已知 $q$ 可以回到當前 stack 內的某個祖先，而這祖先至少在 $lcm(p,q)$ 以上



# 達成新目標

- 想法三：Tarjan's algorithm
  - 希望可以讓 $lcm(p,q)$ 到 $p$ 之間的元素都與 $q$ 「共生死」
  - 拿 $es(q)$ 來更新 $low(p)$ 就可以達成這個目標！
  - 證明：若 $low(p) = es(q)$，則 $p$ 和 $q$ 最後會在同一個 SCC 內
    - 由於 $p$ 較晚推入 stack，故若 $p,q$ 不同 SCC，則必定存在將 $p$ 推出的 $r$ 滿足 $r$ 比 $q$ 更晚但比 $p$ 更早推入 stack
    - 由 $es$ 的定義可知 $es(q) < es(r) < es(p)$
    - 對於 $p$ 的任意祖先 $p'$ 都必定有
      $$low(p') \leq low(p) = es(q)$$
    - $r$ 將 $p$ 推出，根據算法有
      $$low(r) = es(r) > es(q)$$
    - 由於 $r$ 亦為滿足條件之 $p'$，矛盾



# 驗證相容性

- 想法三：Tarjan's algorithm
  - 會不會影響由 back edge 取得的 *low* 的作用呢？
    - 會不會原本 *low* 判定連通，新的 *low* 判定不連通？
    - 會不會原本 *low* 判定不連通，加上後來的 cross edge 後也不應該連通，但新的 *low* 判定連通呢？
  - 會不會被 back edge 取得的 low 影響作用呢？
    - 會不會原本 *low* 判定連通，新的 low 判定不連通？
    - 會不會原本 *low* 判定不連通，加上後來的 back edge 後也不應該連通，但新的 *low* 判定連通呢？
  - 答案都是不會，證明並不困難，留給各位完成 :D

# 算法來也

- 想法三：Tarjan's algorithm

```
1   Stack S;
2   int scnt := 0;
3   int index := 0;
4
5   dfs(Vertex now){
6       low[now] := index;
7       dfn[now] := index;
8       index := index + 1;
9       mark now as visited;
10
11      for all edges e_i as {now, other}
12          if( other is not visited )
13              dfs(other);
14              low[now] := min(low[now], low[other]);
15          else if( other is in S )
16              low[now] := min(low[now], dfn[now]);
17      if( low[now] = dfn[now] )
18          do {
19              Vertex top := S.pop();
20              SCC[top] = scnt;
21          }while( top != now );
22          scnt := scnt + 1;
23  }
```

- 實作時一般不會完整實做時間戳記，而只記錄各點是第幾個被拜訪的點(函數 dfn)
- 複雜度 $O(n+m)$


Sprout


Sprout