Advanced Tree Problem

樹相關問題

複習：無根樹的常見輸入(與圖的輸入相同)

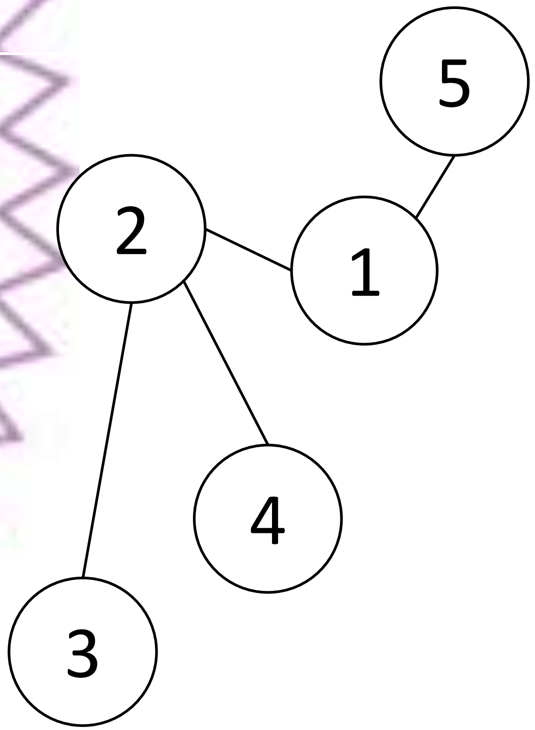
1	2	5	
2	1	3	4
3	2		
4	2		
5	1		

n 個點

$n-1$ 條邊

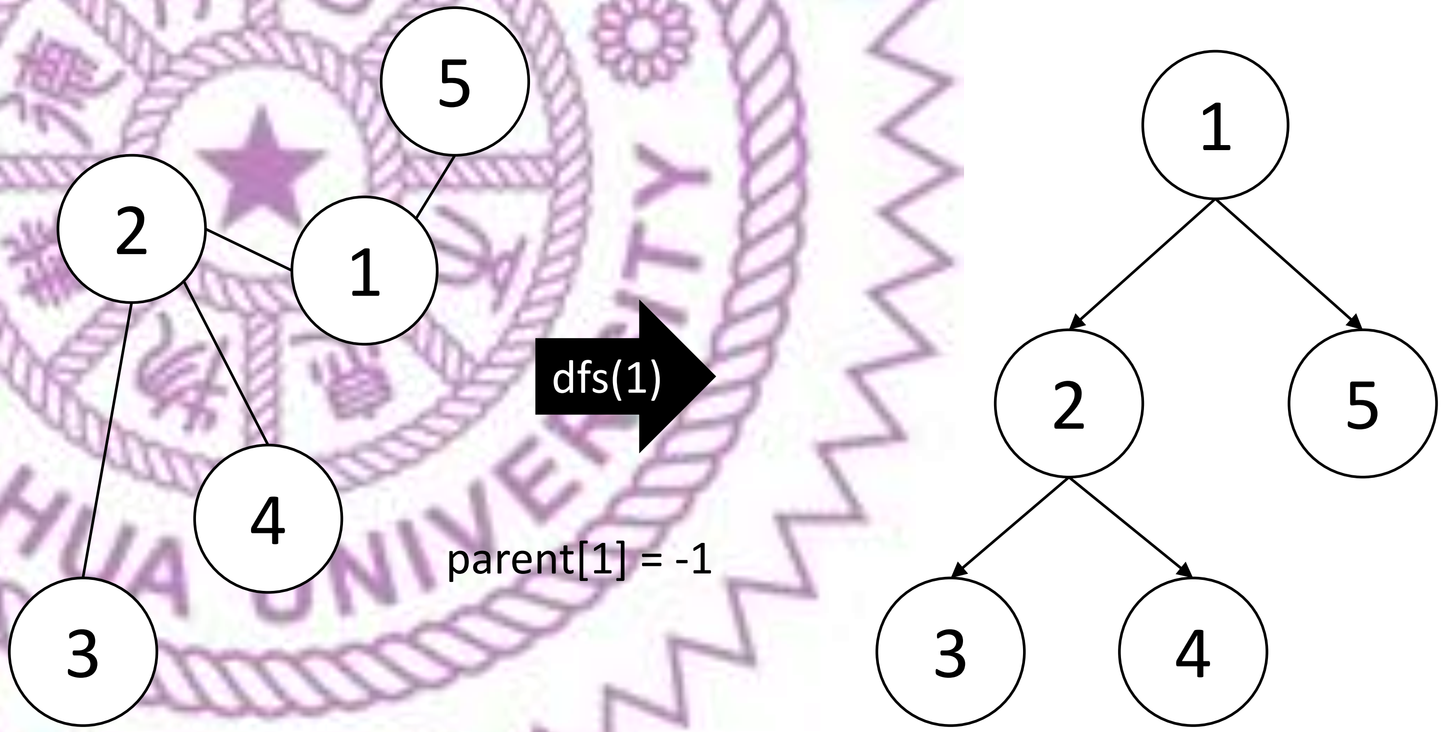
```
5
1 2
2 3
2 4
1 5
```

```
vector<vector<int>> Tree;
int n;
cin >> n;
Tree.assign(n + 1, {});
for (int i = 0; i < n - 1; ++i) {
  int u, v;
  cin >> u >> v;
  Tree[u].emplace_back(v);
  Tree[v].emplace_back(u);
}
```



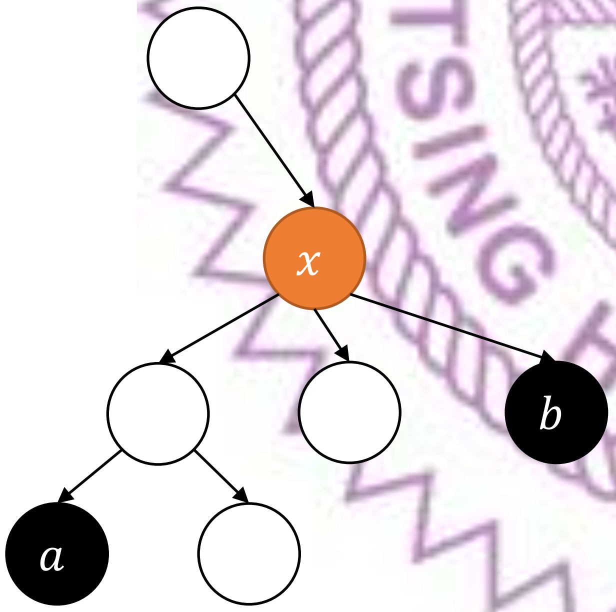
無根樹選一個點當根 ➔ 透過 dfs 走訪

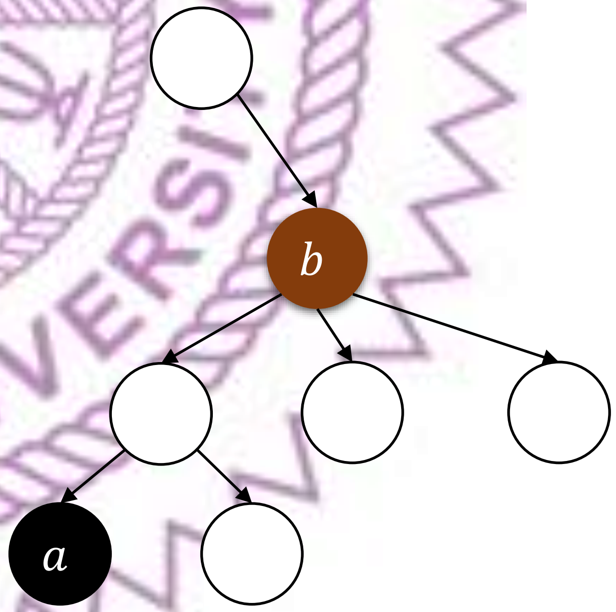
```
vector<int> parent;
void dfs(int u) {
  for (auto v : Tree[u]) {
    if (v == parent[u])
      continue;
    parent[v] = u;
    dfs(v);
  }
}
```



最近共同祖先 (LCA)

- 給你一棵有根樹 T，對於樹上任意兩點 a, b
  可以找到一個點 x 滿足：x 是 a, b 的祖先且深度最深
- 我們稱 x 是 a, b 的最近共同祖先 (LCA)



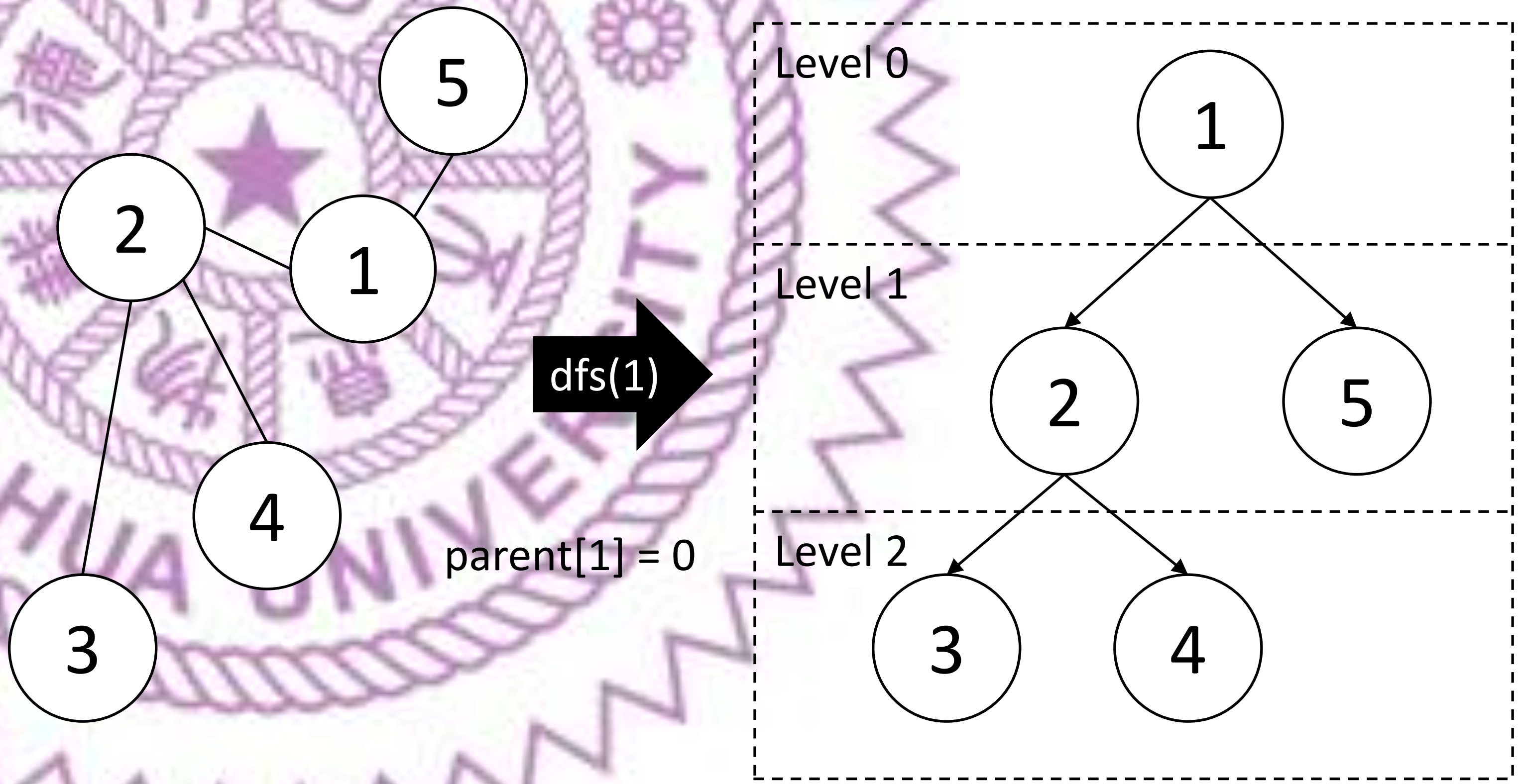


Lowest Common Ancestor - Sparse Table

LCA 倍增法

無根樹轉有根樹同時記錄深度

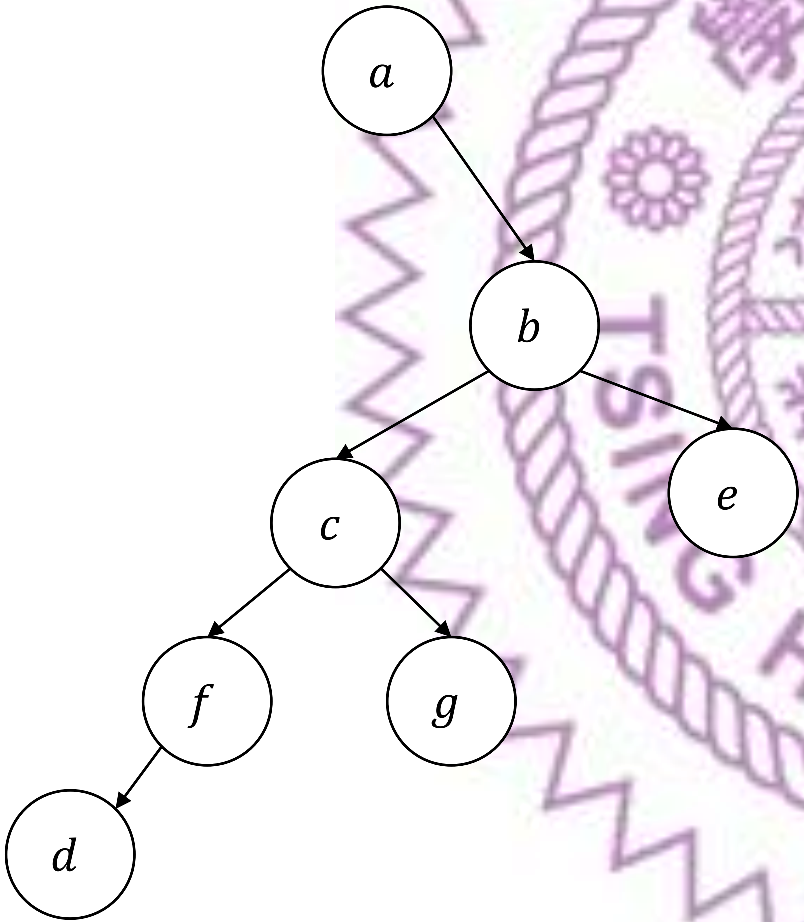
```cpp
vector<int> parent, level;
void dfs(int u, int L = 0) {
  level[u] = L;
  for (auto v : Tree[u]) {
    if (v == parent[u])
      continue;
    parent[v] = u;
    dfs(v, L + 1);
  }
}
```



$O(n)$ 暴力法

```
int LCA(int a, int b) {
  if (level[a] > level[b])
    swap(a, b);
  for (int T = level[b] - level[a]; T--;)
    b = parent[b];
  while (a != b)
    a = parent[a], b = parent[b];
  return a;
}
```

祖先的祖先



設 $anc2[x][k]$表示與節點 x 距離 2^k 的祖先

	a	b	c	d	e	f	g
0	0	a	b	f	b	c	c
1	0	0	a	c	a	b	b
2	0	0	0	a	0	0	0

$$anc2[x][k+1] = anc2\big[anc2[x][k]\big][k]$$

計算 anc2 $O(n \log n)$

```
vector<vector<int>> anc2;
void buildAnc2(int n) {
  int logN = log2(n) + 1;
  anc2.assign(n + 1, vector<int>(logN + 1, 0));
  anc2[0][0] = 0;
  for (int u = 1; u <= n; ++u)
    anc2[u][0] = parent[u];
  for (int k = 0; k < logN; ++k)
    for (int u = 1; u <= n; ++u)
      anc2[u][k + 1] = anc2[anc2[u][k]][k];
}
```

找出距離 x 為 ith 的祖先 $O(\log n)$

```
int ithAnc(int x, int ith) {
  for (size_t k = 0; k < anc2[x].size(); ++k) {
    if (ith & (1 << k))
      x = anc2[x][k];
  }
  return x;
}
```

暴力法稍微加速

```
int LCA(int a, int b) {
  if (level[a] > level[b])
    swap(a, b);
  for (int T = level[b] - level[a]; T--;)
    b = parent[b];
  while (a != b)
    a = parent[a], b = parent[b];
  return a;
}
```

```
int LCA(int a, int b) {
  if (level[a] > level[b])
    swap(a, b);
  b = ithAnc(b, level[b] - level[a]);
  while (a != b)
    a = parent[a], b = parent[b];
  return a;
}
```

暴力法稍微加速

```
int LCA(int a, int b) {
  if (level[a] > level[b])
    swap(a, b);
  b = ithAnc(b, level[b] - level[a]);
  while (a != b)
    a = parent[a], b = parent[b];
  return a;
}
```

這裡還是太慢

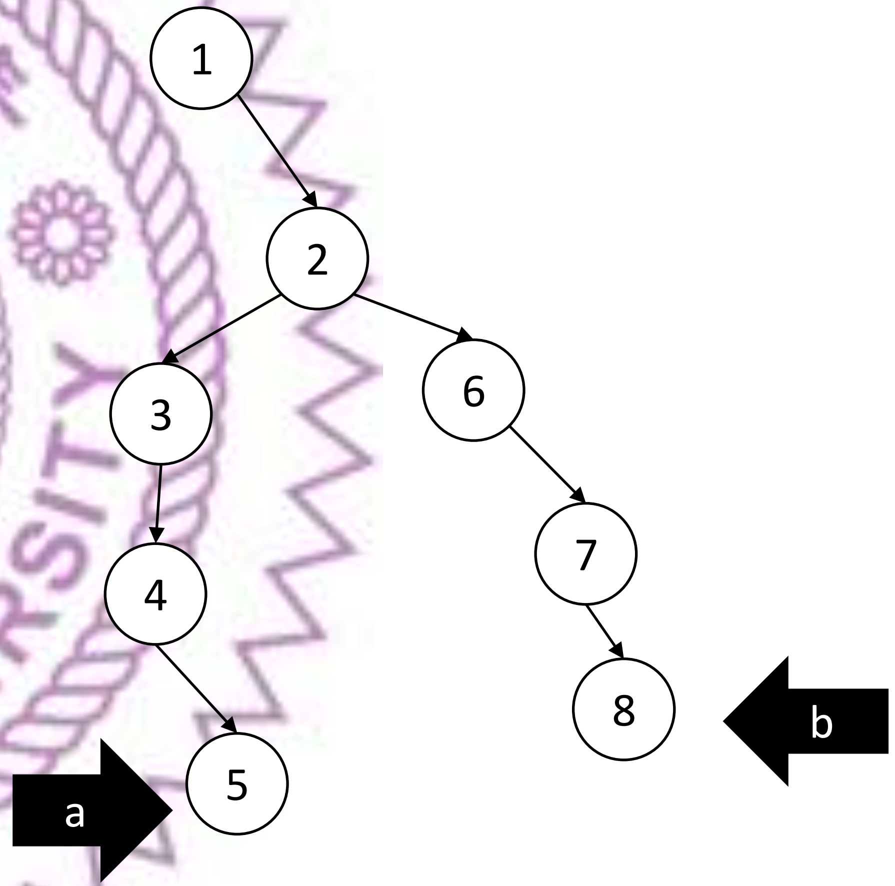
$O(\log n)$ 的方法

```
int LCA(int a, int b) {
  if (level[a] > level[b])
    swap(a, b);
  b = ithAnc(b, level[b] - level[a]);
  if (a == b) return a;
  for (int k = anc2[a].size() - 1; k >= 0; --k)
    if (anc2[a][k] != anc2[b][k])
      a = anc2[a][k], b = anc2[b][k];
  return parent[a];
}
```

利用
Binary Search

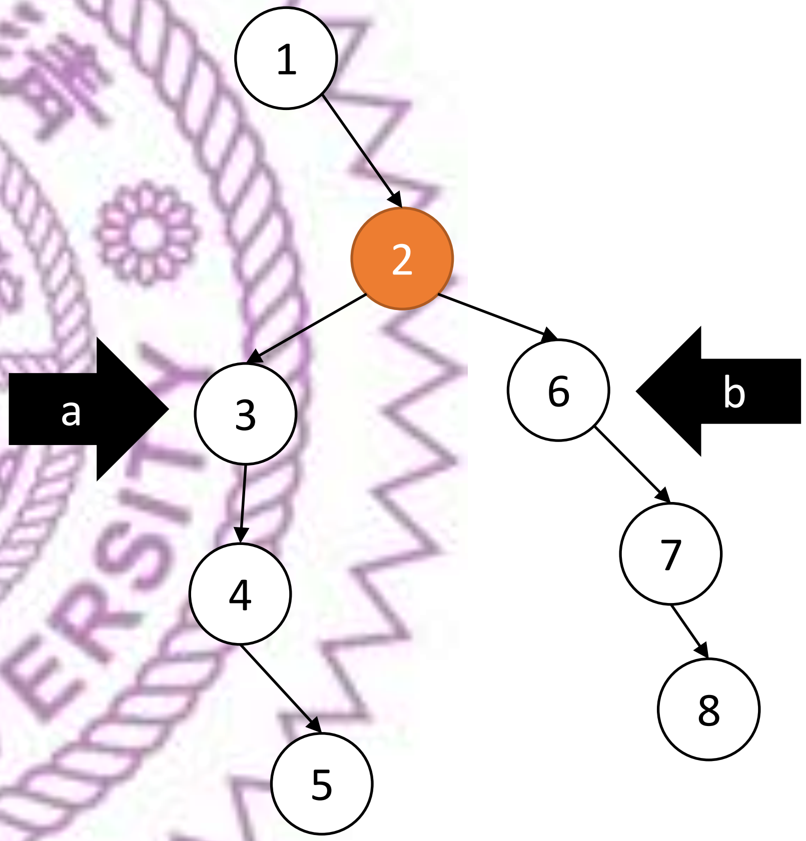
$O(\log n)$ 的方法

```
int LCA(int a, int b) {
  if (level[a] > level[b])
    swap(a, b);
  b = ithAnc(b, level[b] - level[a]);
  if (a == b) return a;
  for (int k = anc2[a].size() - 1; k >= 0; --k)
    if (anc2[a][k] != anc2[b][k])
      a = anc2[a][k], b = anc2[b][k];
  return parent[a];
}
```



$O(\log n)$ 的方法

```cpp
int LCA(int a, int b) {
  if (level[a] > level[b])
    swap(a, b);
  b = ithAnc(b, level[b] - level[a]);
  if (a == b) return a;
  for (int k = anc2[a].size() - 1; k >= 0; --k)
    if (anc2[a][k] != anc2[b][k])
      a = anc2[a][k], b = anc2[b][k];
  return parent[a];
}
```



Euler Tour Technique

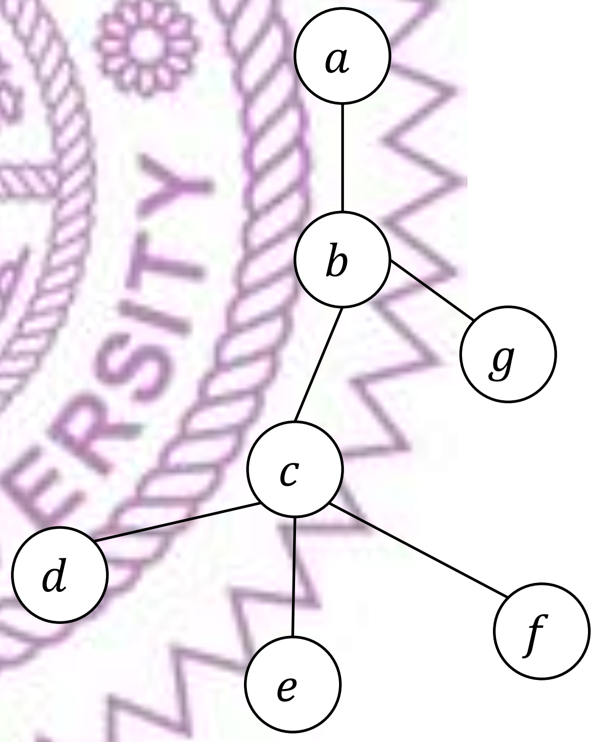
樹序列化

樹狀結構序列化

Arr

0	1	2	3	4	5	6
a	b	c	d	e	f	g

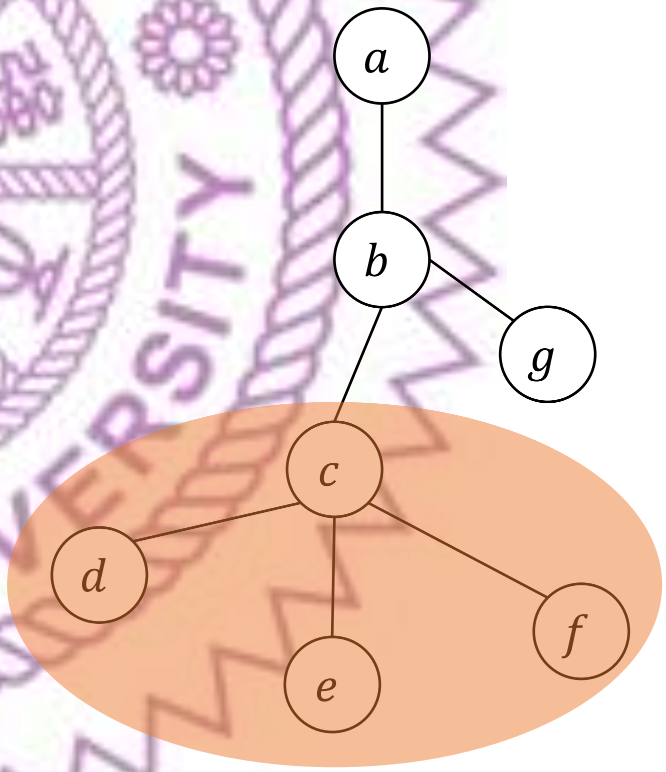
```cpp
vector<int> Arr;
void dfs(int u) {
  Arr.push_back(u);
  for (auto v : Tree[u]) {
    if (v == parent[u])
      continue;
    parent[v] = u;
    dfs(v);
  }
}
```



同一個子樹在序列中會連續

	0	1	2	3	4	5	6
Arr	a	b	c	d	e	f	g

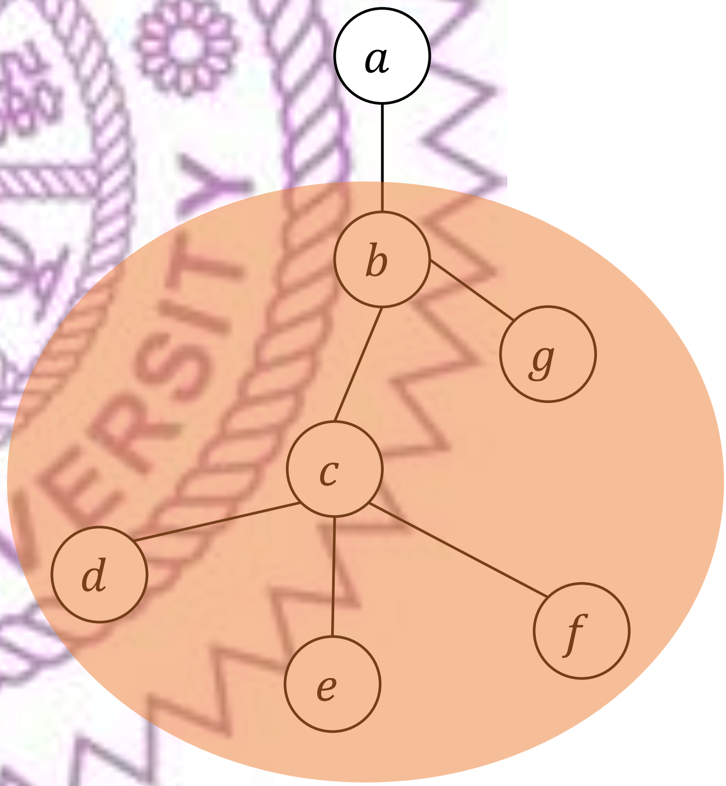
```
vector<int> Arr;
void dfs(int u) {
  Arr.push_back(u);
  for (auto v : Tree[u]) {
    if (v == parent[u])
      continue;
    parent[v] = u;
    dfs(v);
  }
}
```



同一個子樹在序列中會連續

Arr	0	1	2	3	4	5	6
	a	b	c	d	e	f	g

```cpp
vector<int> Arr;
void dfs(int u) {
  Arr.push_back(u);
  for (auto v : Tree[u]) {
    if (v == parent[u])
      continue;
    parent[v] = u;
    dfs(v);
  }
}
```

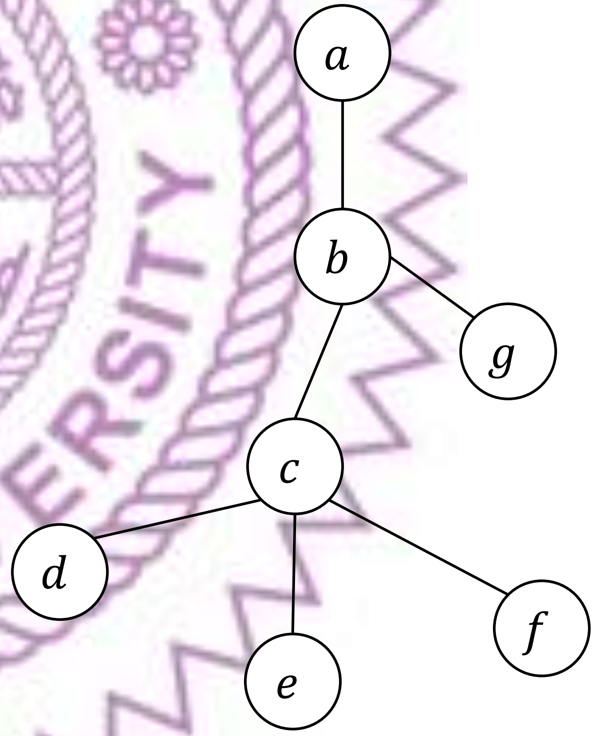


記錄子樹範圍

	a	b	c	d	e	f	g
In	0	1	2	3	4	5	6
Out	6	6	5	3	4	5	6

	0	1	2	3	4	5	6
Arr	a	b	c	d	e	f	g

```
vector<int> Arr;
vector<int> In, Out;
void dfs(int u) {
  Arr.push_back(u);
  In[u] = Arr.size() - 1;
  for (auto v : Tree[u]) {
    if (v == parent[u])
      continue;
    parent[v] = u;
    dfs(v);
  }
  Out[u] = Arr.size() - 1;
}
```

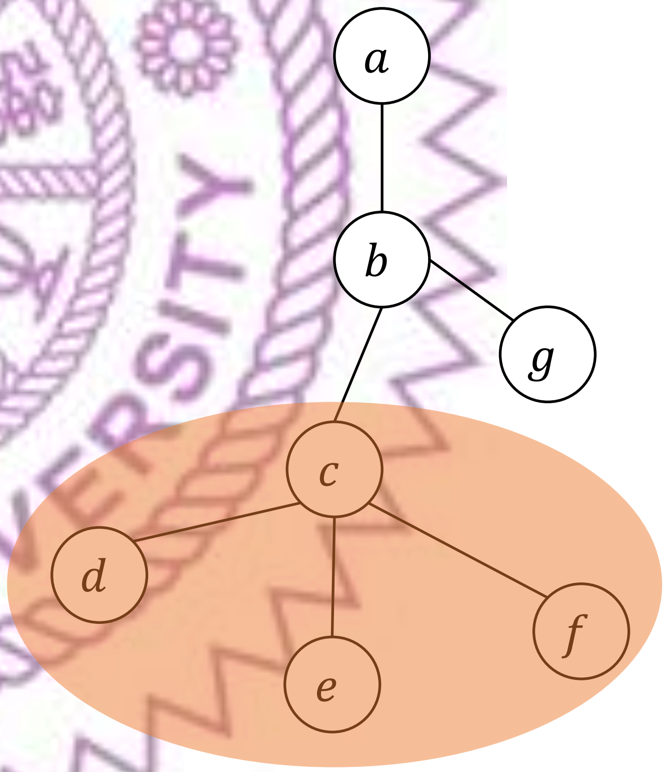


記錄子樹範圍 – 好像可以用線段樹維護?

	a	b	c	d	e	f	g
In	0	1	2	3	4	5	6
Out	6	6	5	3	4	5	6

	0	1	2	3	4	5	6
Arr	a	b	c	d	e	f	g

```
vector<int> Arr;
vector<int> In, Out;
void dfs(int u) {
  Arr.push_back(u);
  In[u] = Arr.size() - 1;
  for (auto v : Tree[u]) {
    if (v == parent[u])
      continue;
    parent[v] = u;
    dfs(v);
  }
  Out[u] = Arr.size() - 1;
}
```



特殊題

- 給你一個 n 個點的樹，編號 $1 \sim n$，每個點 u 有自己的權重 $cost[u]$
  再給你 q 個操作，操作有兩種：

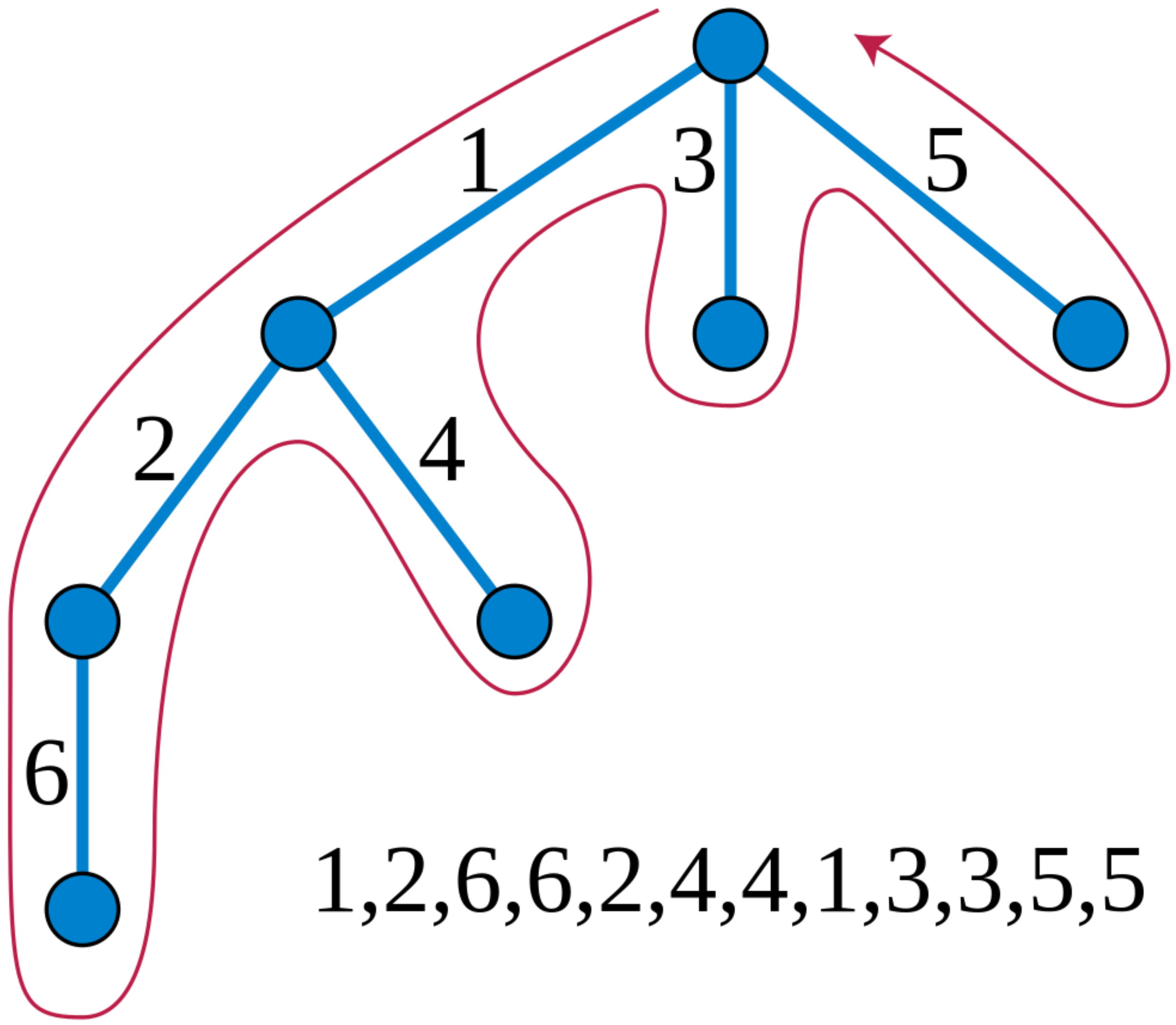
- $query(u)$:
  查詢以 u 為根的子樹的權重總和

- $move(a, b)$:
  將以 a 為根的子樹，移動到節點 b 底下，保證 b 不是 a 的後代

- $1 \leq n, q \leq 10^6$

$move(c, a)$

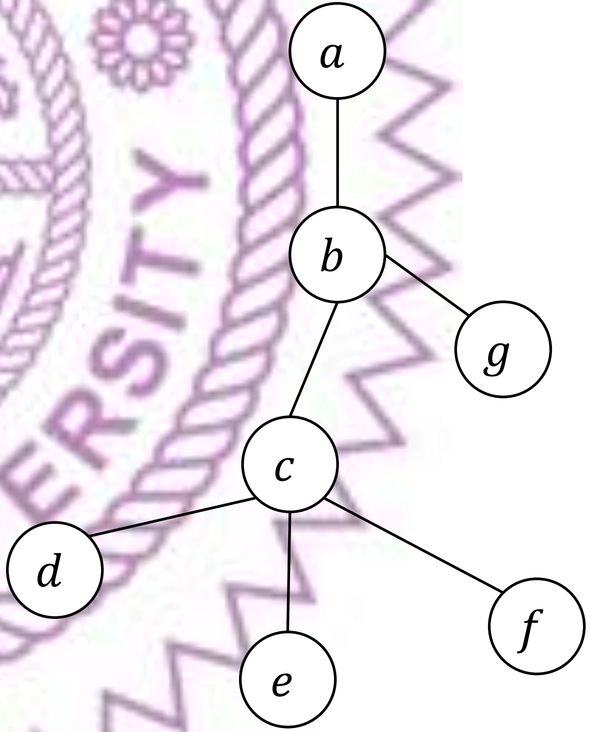
Stirling Permutation



進來出去都記錄 – 兩倍空間

a	b	c	d	d	e	e	f	f	c	g	g	b	a

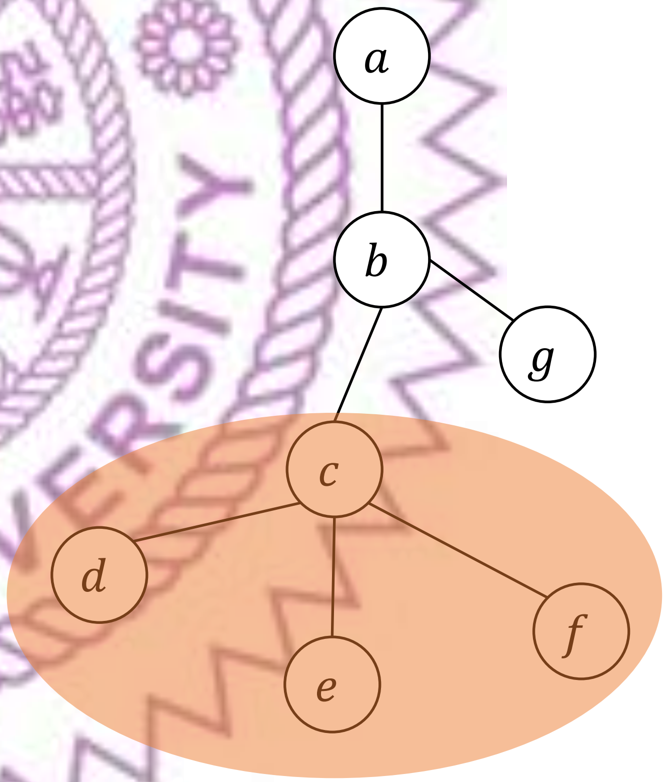
```cpp
vector<int> Arr;
void dfs(int u) {
  Arr.push_back(u);
  for (auto v : Tree[u]) {
    if (v == parent[u])
      continue;
    parent[v] = u;
    dfs(v);
  }
  Arr.push_back(u);
}
```



進來出去都記錄 – 子樹範圍更清楚

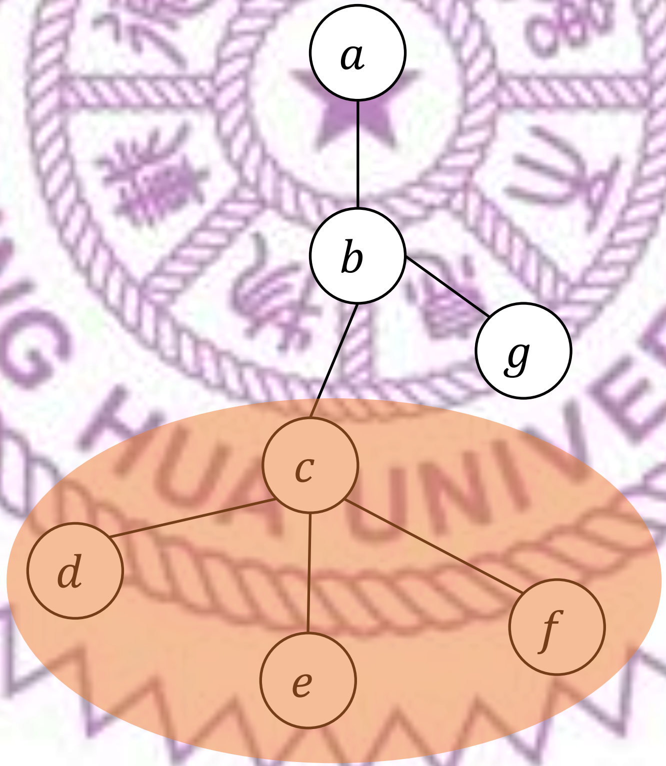
a	b	c	d	d	e	e	f	f	c	g	g	b	a

```cpp
vector<int> Arr;
void dfs(int u) {
  Arr.push_back(u);
  for (auto v : Tree[u]) {
    if (v == parent[u])
      continue;
    parent[v] = u;
    dfs(v);
  }
  Arr.push_back(u);
}
```

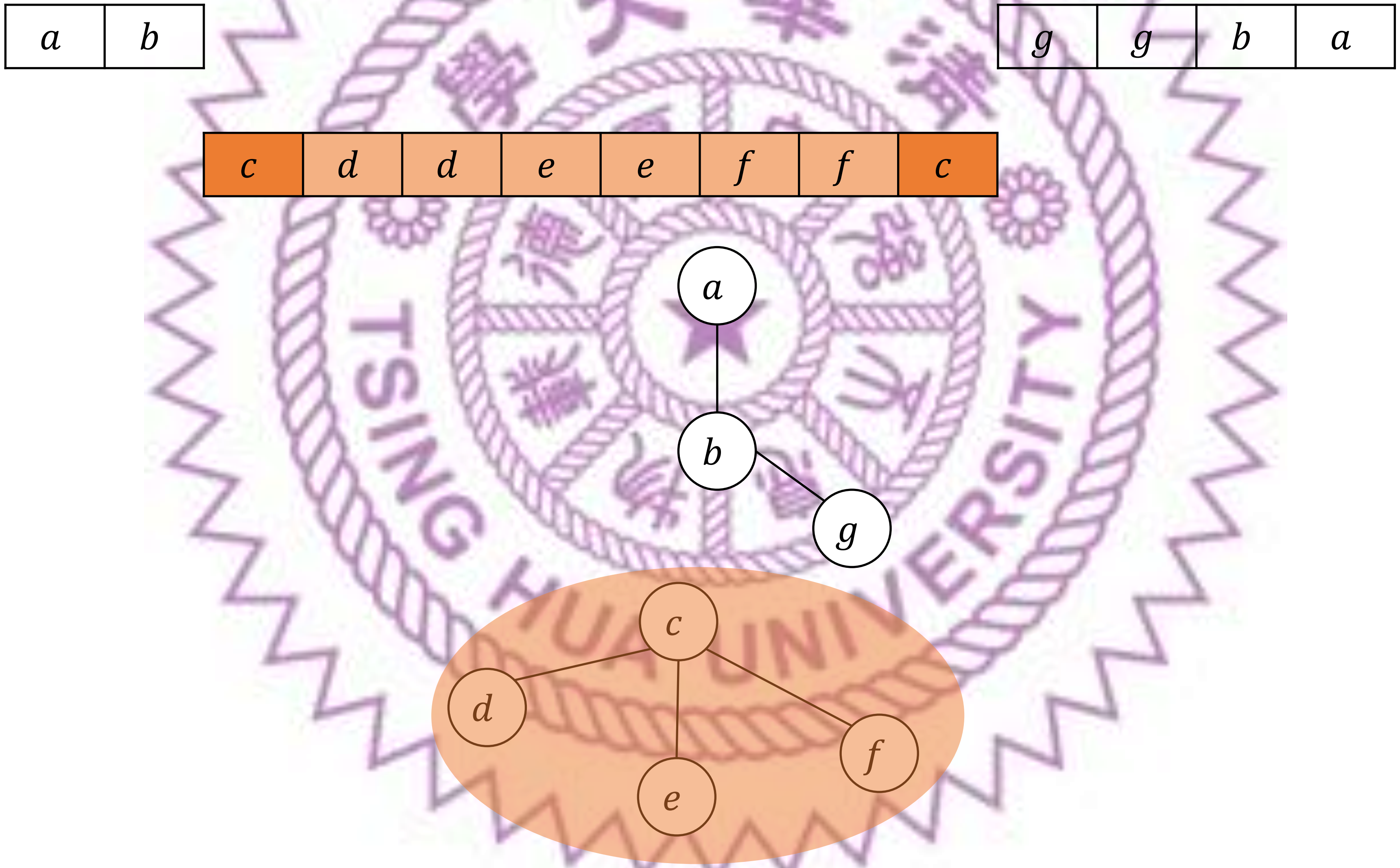


$move(c, a)$

a	b	c	d	d	e	e	f	f	c	g	g	b	a



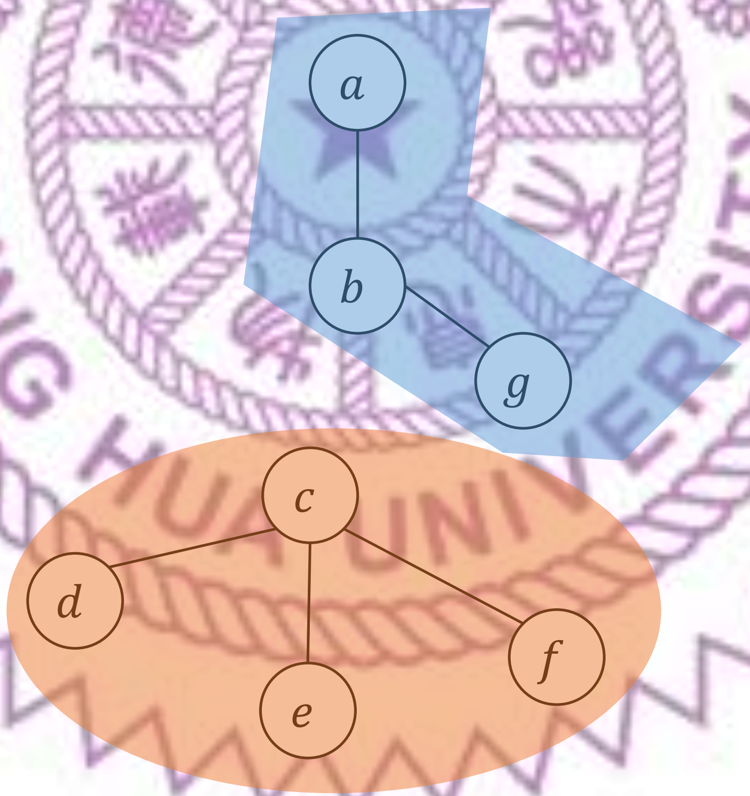
$move(c, a)$



$move(c,a)$

a	b	g	g	b	a

c	d	d	e	e	f	f	c

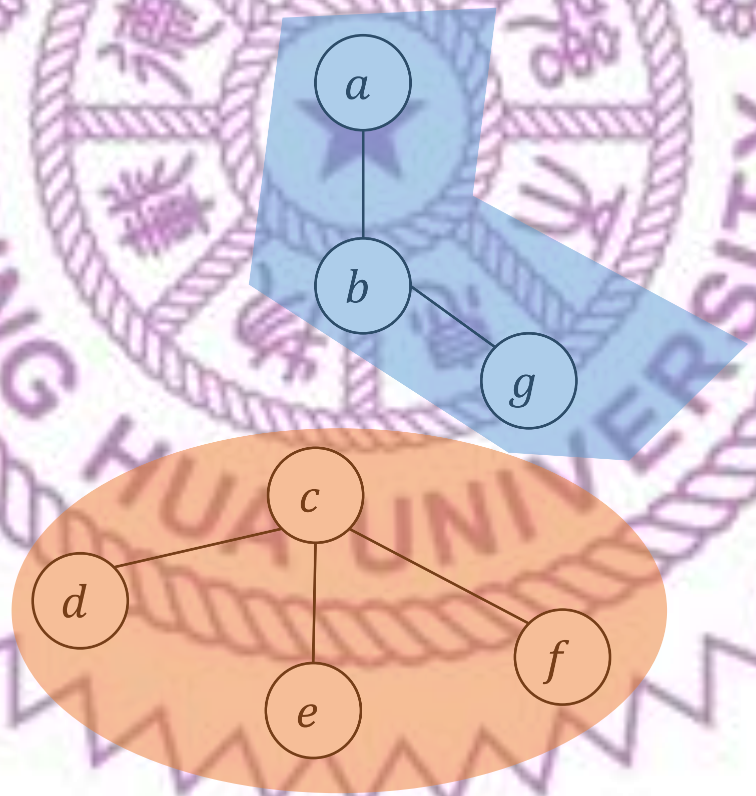


$$move(c,a)$$

a

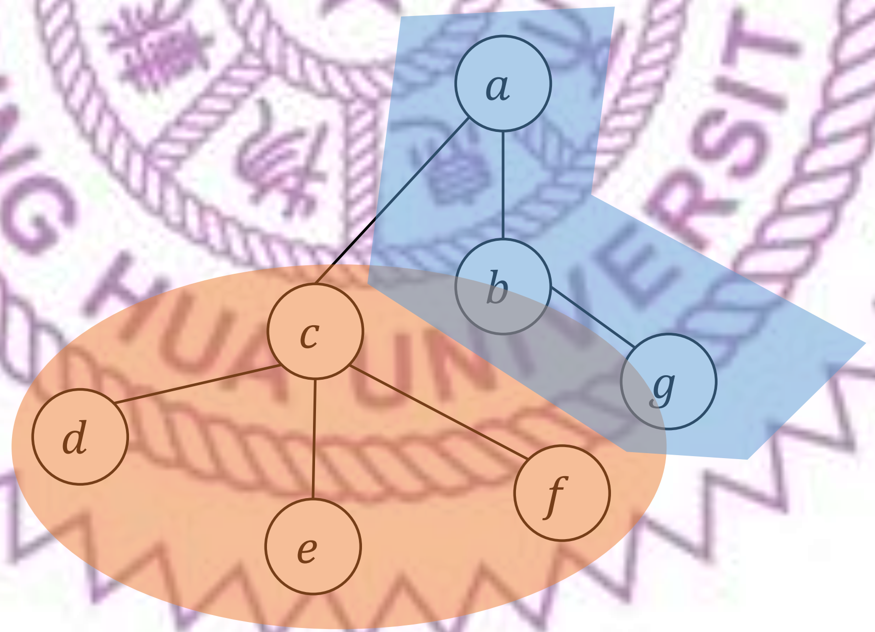
b	g	g	b	a

c	d	d	e	e	f	f	c



$move(c, a)$

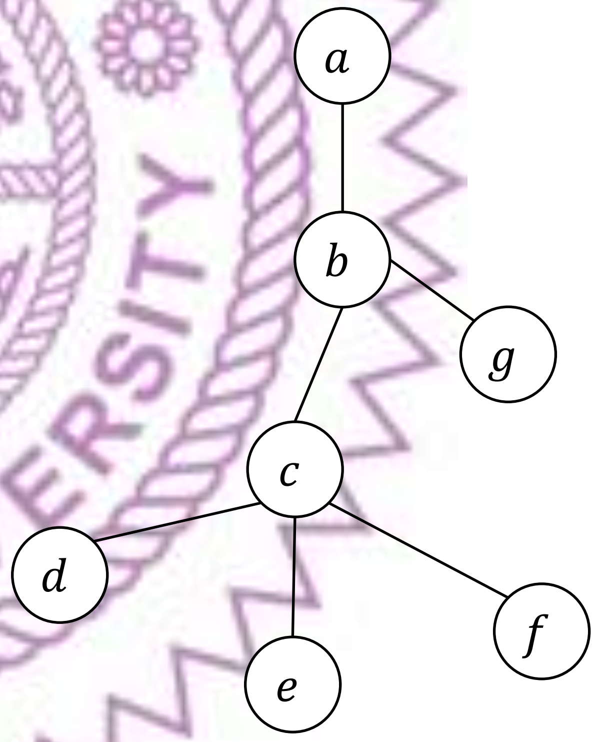
a	c	d	d	e	e	f	f	c	b	g	g	b	a



用 Treap 紀錄

a	b	c	d	d	e	e	f	f	c	g	g	b	a

```
Treap *root = nullptr;
vector<Treap *> In, Out;
void dfs(int u) {
  In[u] = new Treap(cost[u]);
  root = merge(root, In[u]);
  for (auto v : Tree[u]) {
    if (v == parent[u])
      continue;
    parent[v] = u;
    dfs(v);
  }
  Out[u] = new Treap(0);
  root = merge(root, Out[u]);
}
```



Treap 紀錄 parent

```cpp
struct Treap {
  Treap *lc = nullptr, *rc = nullptr;
  Treap *pa = nullptr;
  unsigned pri, size;
  long long Val, Sum;
  Treap(int Val):
    pri(rand()), size(1),
    Val(Val), Sum(Val) {}
  void pull();
};
```

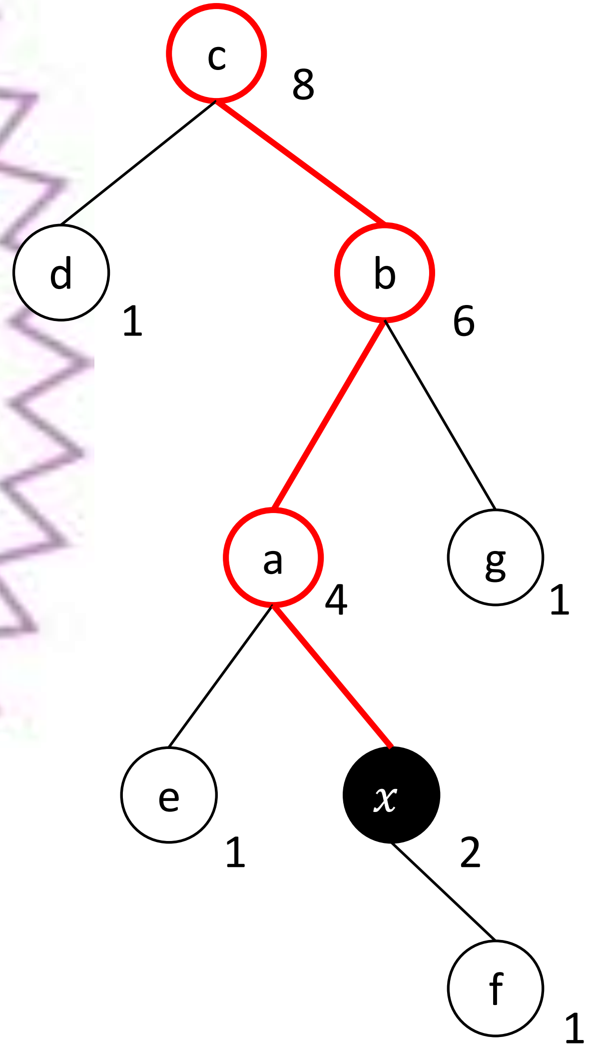
```cpp
void Treap::pull() {
  size = 1;
  Sum = Val;
  pa = nullptr;
  if (lc) {
    size += lc->size;
    Sum += lc->Sum;
    lc->pa = this;
  }
  if (rc) {
    size += rc->size;
    Sum += rc->Sum;
    rc->pa = this;
  }
}
```

找出節點在中序的編號

1	2	3	4	5	6	7	8
d	c	e	a	x	f	b	g

```
size_t getIdx(Treap *x) {
  assert(x);
  size_t Idx = 0;
  for (Treap *child = x->rc; x;) {
    if (child == x->rc)
      Idx += 1 + size(x->lc);
    child = x;
    x = x->pa;
  }
  return Idx;
}
```

$$
\begin{aligned}
getIdx(x) &= (size(x \to lc) + 1) \\
&+ (size(a \to lc) + 1) \\
&+ (size(c \to lc) + 1) \\
&= 1 + 2 + 2 \\
&= 5
\end{aligned}
$$



這樣就能切出想要的東西了

```cpp
void move(Treap *&root, int a, int b) {
  size_t a_in = getIdx(In[a]), a_out = getIdx(Out[a]);
  auto [L, tmp] = splitK(root, a_in - 1);
  auto [tree_a, R] = splitK(tmp, a_out - a_in + 1);
  root = merge(L, R);
  tie(L, R) = splitK(root, getIdx(In[b]));
  root = merge(L, merge(tree_a, R));
}
```

比較

Link-Cut Tree(不會教)	Euler Tour Technique
• 必須用複雜的 Splay Tree	• 可以用 線段樹 / Treap 維護
• 維護路徑訊息	• 維護子樹訊息
• 各種操作 $O(\log n)$	• 各種操作 $O(\log n)$

Heavy-Light Decomposition

輕重鏈剖分 (樹鏈剖分)

經典題

- 給你一個 n 個點的樹，編號 $1\sim n$，每個點 u 有自己的權重 $cost[u]$
  再給你 q 個操作，操作有兩種：

- $query(u,v)$:
  查詢 $u \rightarrow v$ 路徑上點的權重總和

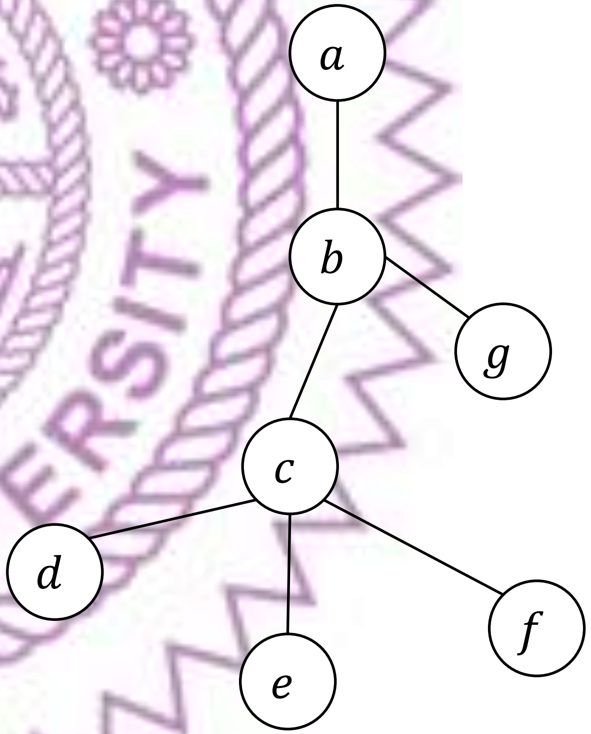
- $update(x,val)$:
  將結點 x 的權重改成 val

- $1 \leq n,q \leq 10^6$

出去時紀錄負的權重

a	b	c	d	$-d$	e	$-e$	f	$-f$	$-c$	g	$-g$	$-b$	$-a$

```
vector<int> Arr;
vector<int> In, Out;
void dfs(int u) {
  Arr.emplace_back(cost[u]);
  In[u] = Arr.size() - 1;
  for (auto v : Tree[u]) {
    if (v == parent[u])
      continue;
    parent[v] = u;
    dfs(v);
  }
  Arr.emplace_back(-cost[u]);
  Out[u] = Arr.size() - 1;
}
```

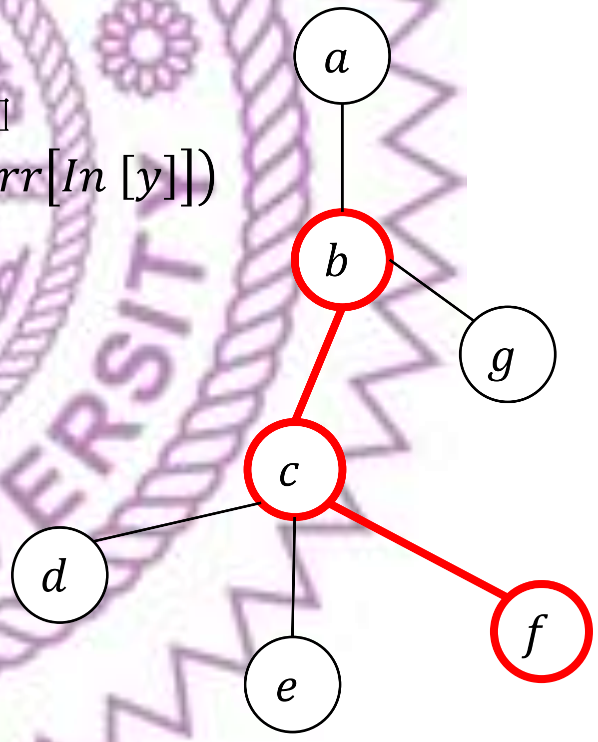


出去時紀錄負的權重

$In[b]$							$In[f]$						
a	b	c	d	$-d$	e	$-e$	f	$-f$	$-c$	g	$-g$	$-b$	$-a$

```cpp
vector<int> Arr;
vector<int> In, Out;
void dfs(int u) {
  Arr.emplace_back(cost[u]);
  In[u] = Arr.size() - 1;
  for (auto v : Tree[u]) {
    if (v == parent[u])
      continue;
    parent[v] = u;
    dfs(v);
  }
  Arr.emplace_back(-cost[u]);
  Out[u] = Arr.size() - 1;
}
```

若 x 是 y 的祖先
則 $x \rightarrow y$ 路徑上的權重和
等於 $sum(Arr[In[x]] \sim Arr[In[y]])$

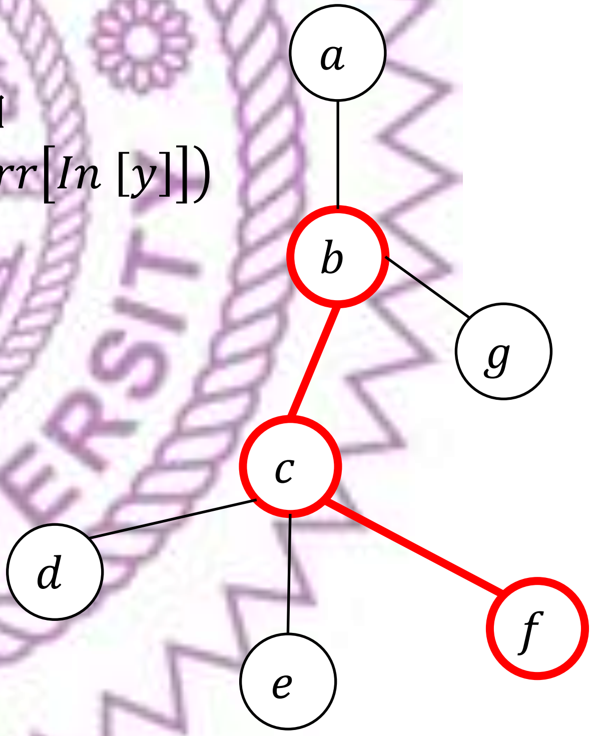


出去時紀錄負的權重

	$In[b]$						$In[f]$						
a	b	c	d	$-d$	e	$-e$	f	$-f$	$-c$	g	$-g$	$-b$	$-a$

```
vector<int> Arr;
vector<int> In, Out;
void dfs(int u) {
  Arr.emplace_back(cost[u]);
  In[u] = Arr.size() - 1;
  for (auto v : Tree[u]) {
    if (v == parent[u])
      continue;
    parent[v] = u;
    dfs(v);
  }
  Arr.emplace_back(-cost[u]);
  Out[u] = Arr.size() - 1;
}
```

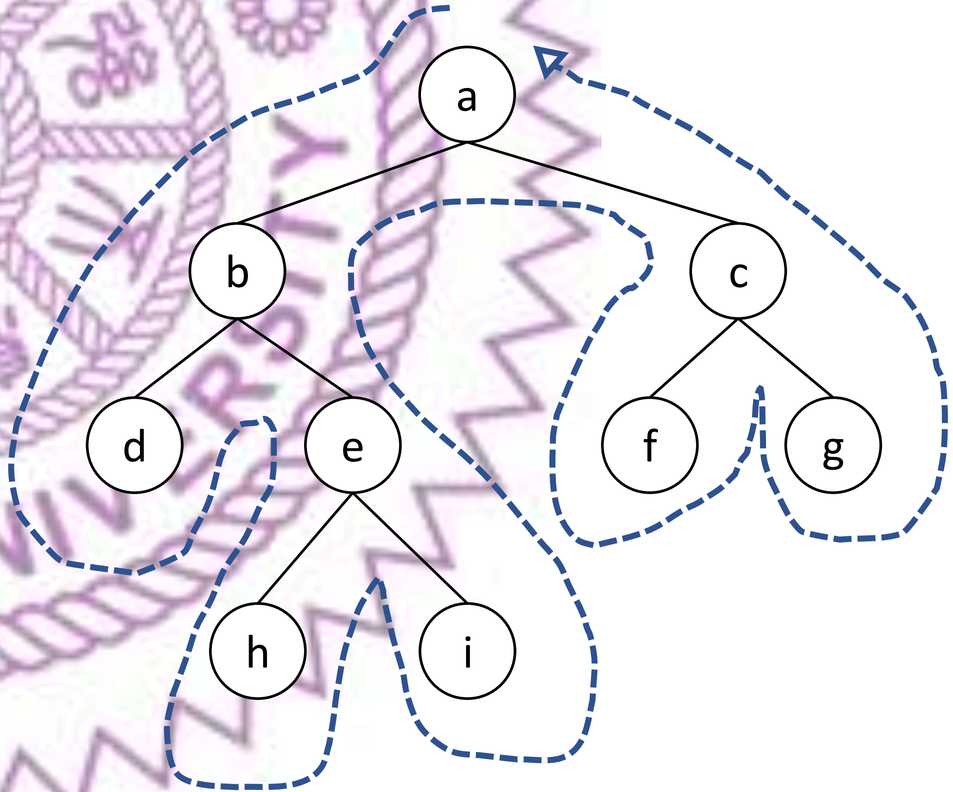
若 x 是 y 的祖先
則 $x \to y$ 路徑上的權重和
等於 $sum(Arr[In\ [x]] \sim Arr[In\ [y]])$



利用 RMQ 求 LCA

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	d	b	e	h	e	i	e	b	a	c	f	c	g	c	a
0	1	2	1	2	3	2	3	2	1	0	1	2	1	2	1	0

```
vector<pair<int, int>> Level;
vector<int> In, Out;
void dfs(int u, int L = 0) {
  Level.emplace_back(u, L);
  In[u] = Level.size() - 1;
  for (auto v : Tree[u]) {
    if (v == parent[u])
      continue;
    parent[v] = u;
    dfs(v, L + 1);
    Level.emplace_back(u, L);
  }
  Out[u] = Level.size() - 1;
}
```



利用 RMQ 求 LCA

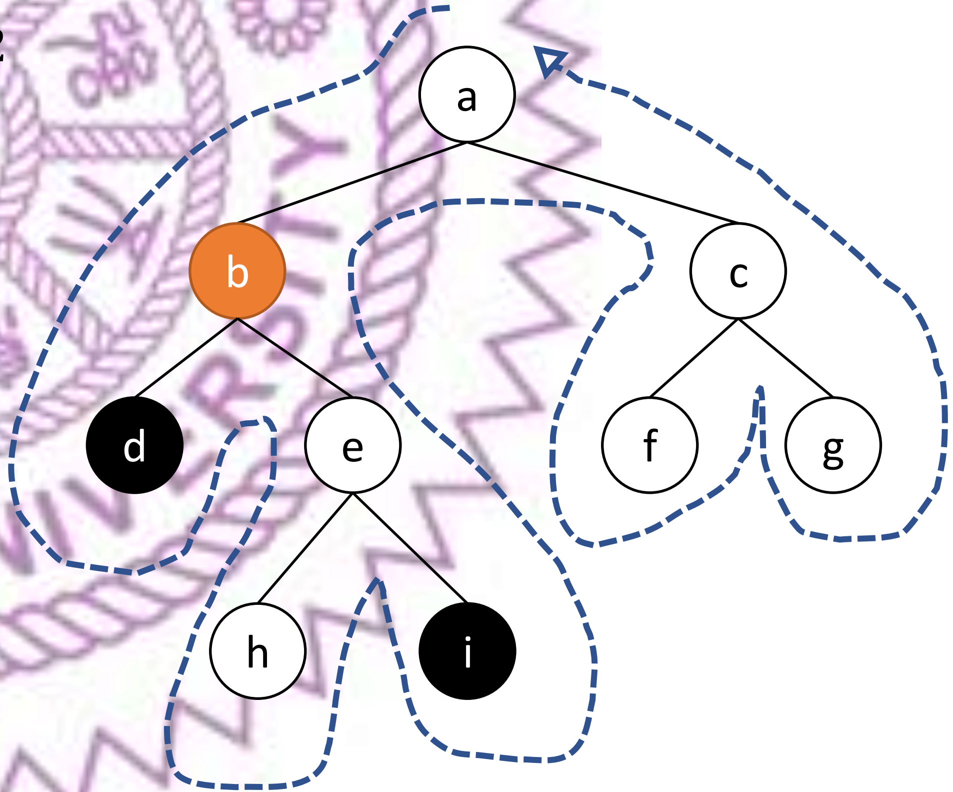
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	b	d	b	e	h	e	i	e	b	a	c	f	c	g	c	a
0	1	2	1	2	3	2	3	2	1	0	1	2	1	2	1	0

```
vector<pair<int, int>> Level;
vector<int> In, Out;
void dfs(int u, int L = 0) {
  Level.emplace_back(u, L);
  In[u] = Level.size() - 1;
  for (auto v : Tree[u]) {
    if (v == parent[u])
      continue;
    parent[v] = u;
    dfs(v, L + 1);
    Level.emplace_back(u, L);
  }
  Out[u] = Level.size() - 1;
}
```

$In[d] = 2$

$In[i] = 7$

變成 RMQ 查詢
區間最小值



$query(u,v) - O(\log n)$

- 找處 u, v 的 LCA x

- 計算 $x \rightarrow u$ 路徑上的權重和
  和 $x \rightarrow v$ 路徑上的權重和

- 相加後再扣掉 x 的權重值就行了！

經典題 2

- 給你一個 n 個點的樹，編號 $1\sim n$，每個點 u 有自己的權重 $cost[u]$
  再給你 q 個操作，操作有兩種：

- $query(u,v)$:
  查詢 $u \rightarrow v$ 路徑上點的權重最小值

- $update(x,val)$:
  將結點 x 的權重改成 val

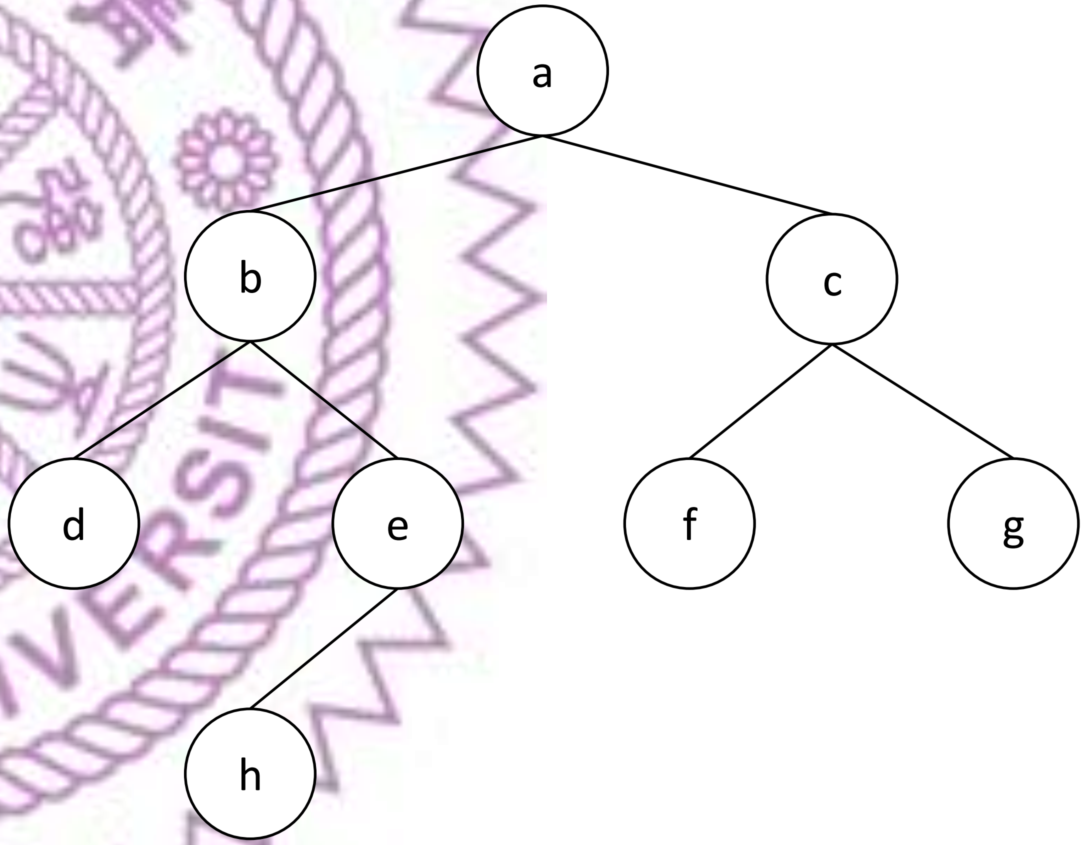
- $1 \leq n,q \leq 10^6$

樹鏈剖分

- 每個點找出 size 最大的小孩
  稱為**重小孩 (Heavy child)**

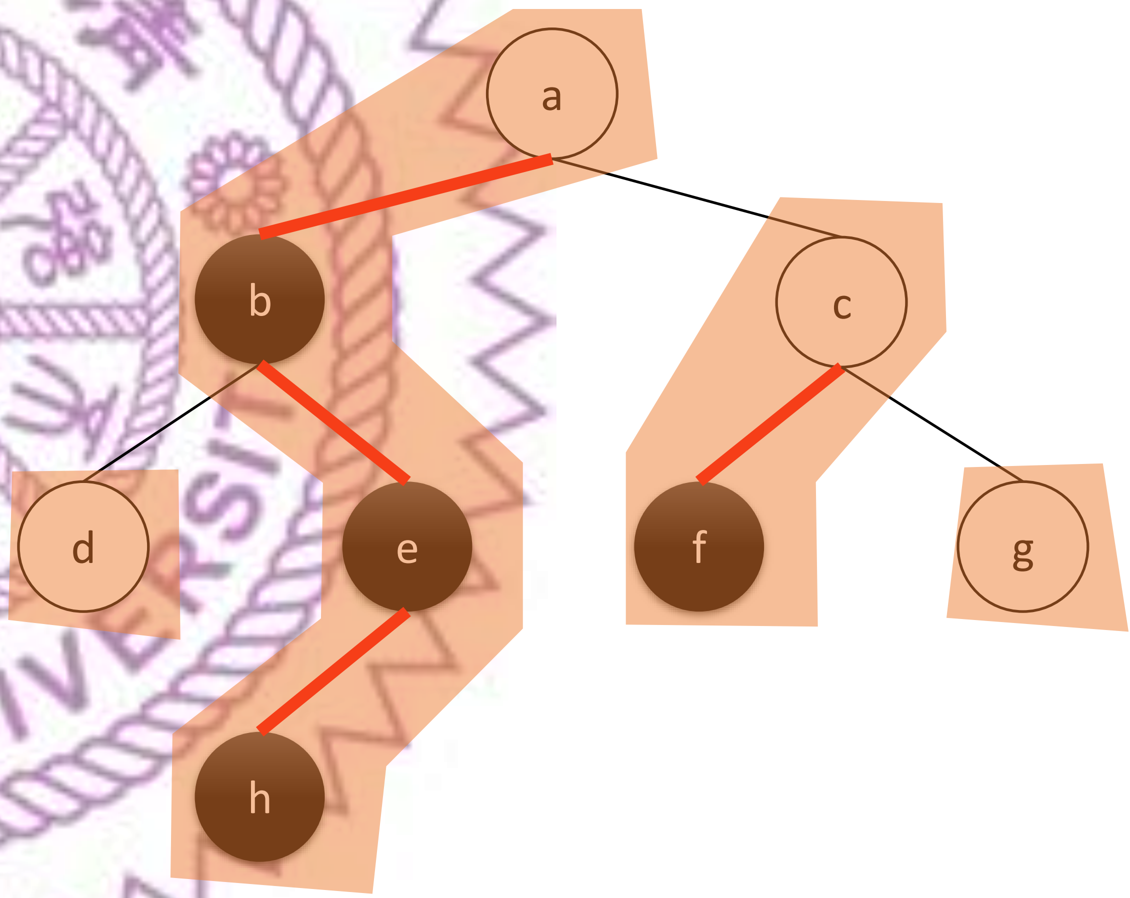
- 連向 Heavy Child 的邊
  被稱之為重邊 (Heavy edge)

- 反之為輕邊 (Light edge)



樹鏈剖分

- 每個點找出 size 最大的小孩
  稱為**重小孩 (Heavy child)**
- 連向 Heavy Child 的邊
  被稱之為重邊 (Heavy edge)
- 反之為輕邊 (Light edge)
- 重邊形成的路徑就是「鏈」



每個點紀錄最大的小孩

```cpp
vector<int> size, HeavyChild;
vector<int> parent, level;
void findHeavyChild(int u, int L = 0) {
  level[u] = L;
  size[u] = 1;
  HeavyChild[u] = -1;
  for (auto v : Tree[u]) {
    if (v == parent[u])
      continue;
    parent[v] = u;
    findHeavyChild(v, L + 1);
    if (HeavyChild[u] == -1 || size[v] > size[HeavyChild[u]])
      HeavyChild[u] = v;
    size[u] += size[v];
  }
}
```

樹鏈剖分

Arr

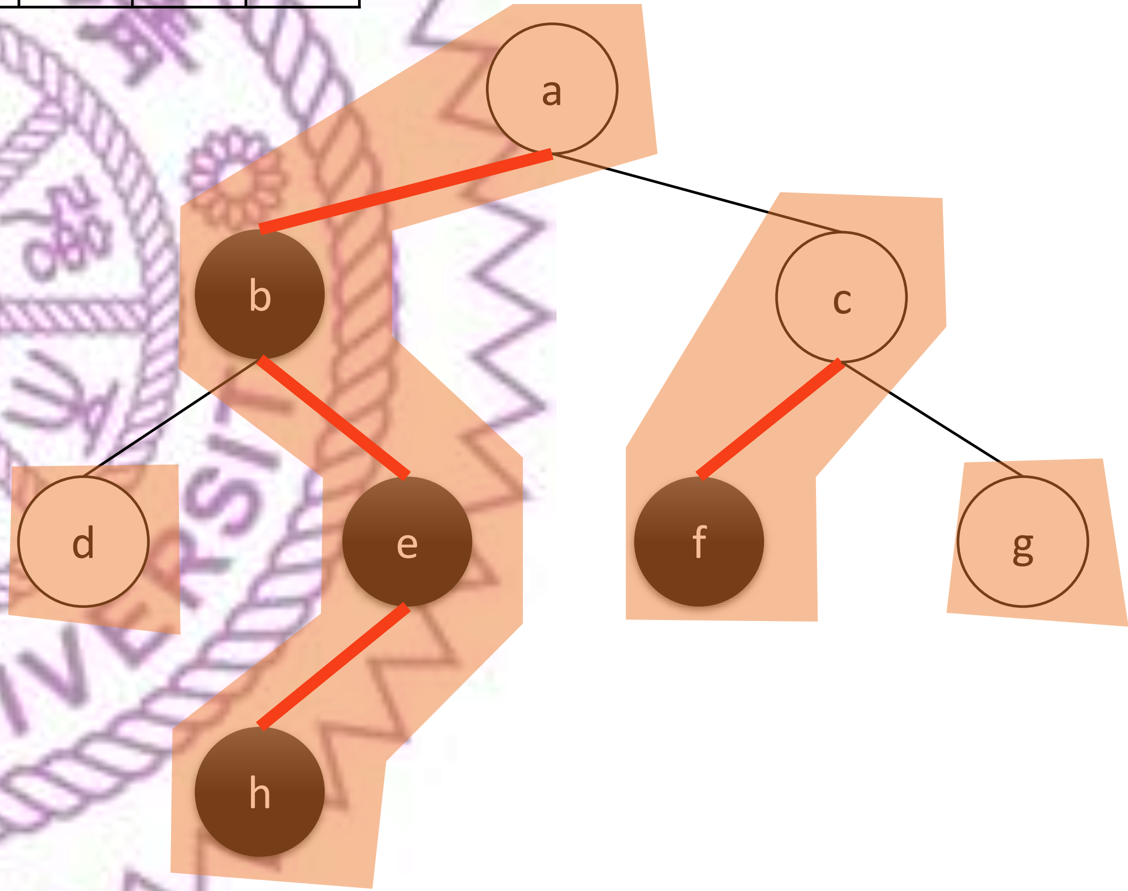
0	1	2	3	4	5	6	7
a	b	e	h	d	c	f	g

Top

a	b	c	d	e	f	g	h
a	a	c	d	a	c	g	a

- 每個鏈的點
  都記錄最上面的點是誰

- 在做樹序列化的時候
  同一個鏈的點要連續在一起



樹鏈剖分

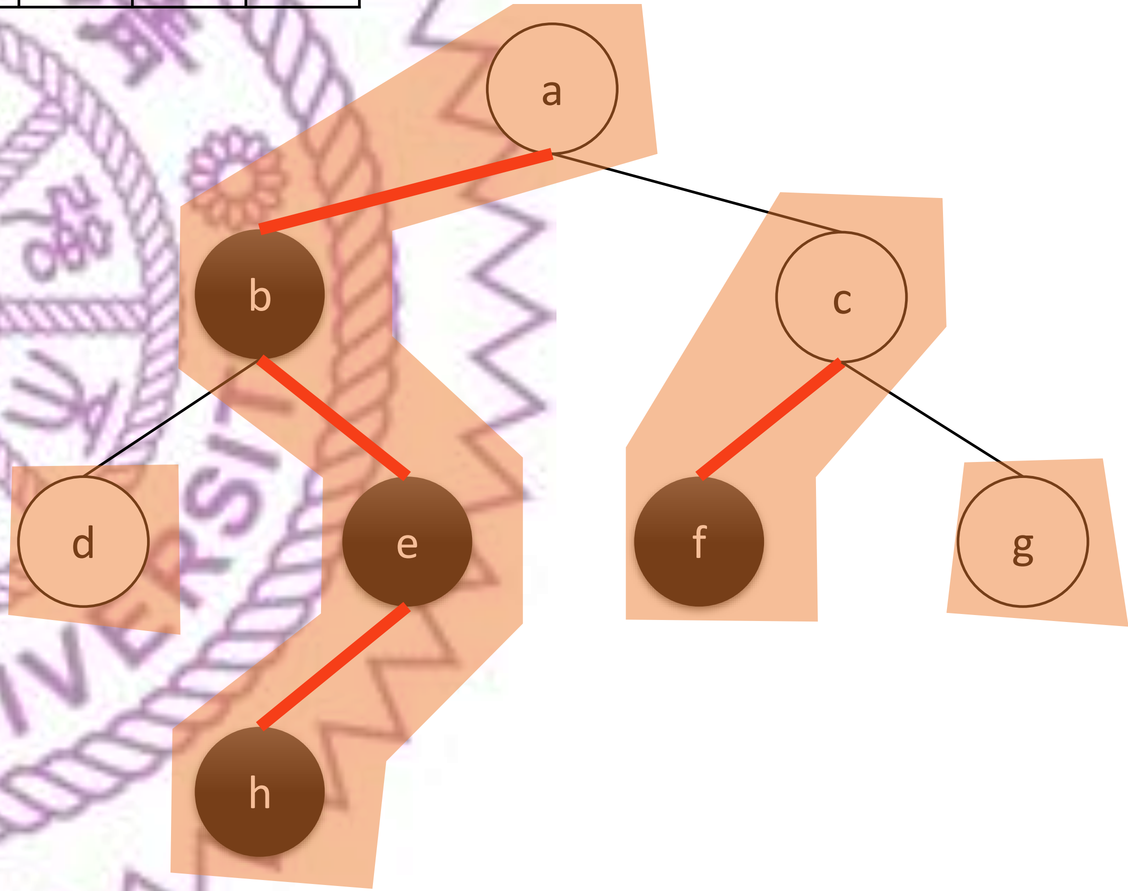
Arr

0	1	2	3	4	5	6	7
a	b	e	h	d	c	f	g

Top

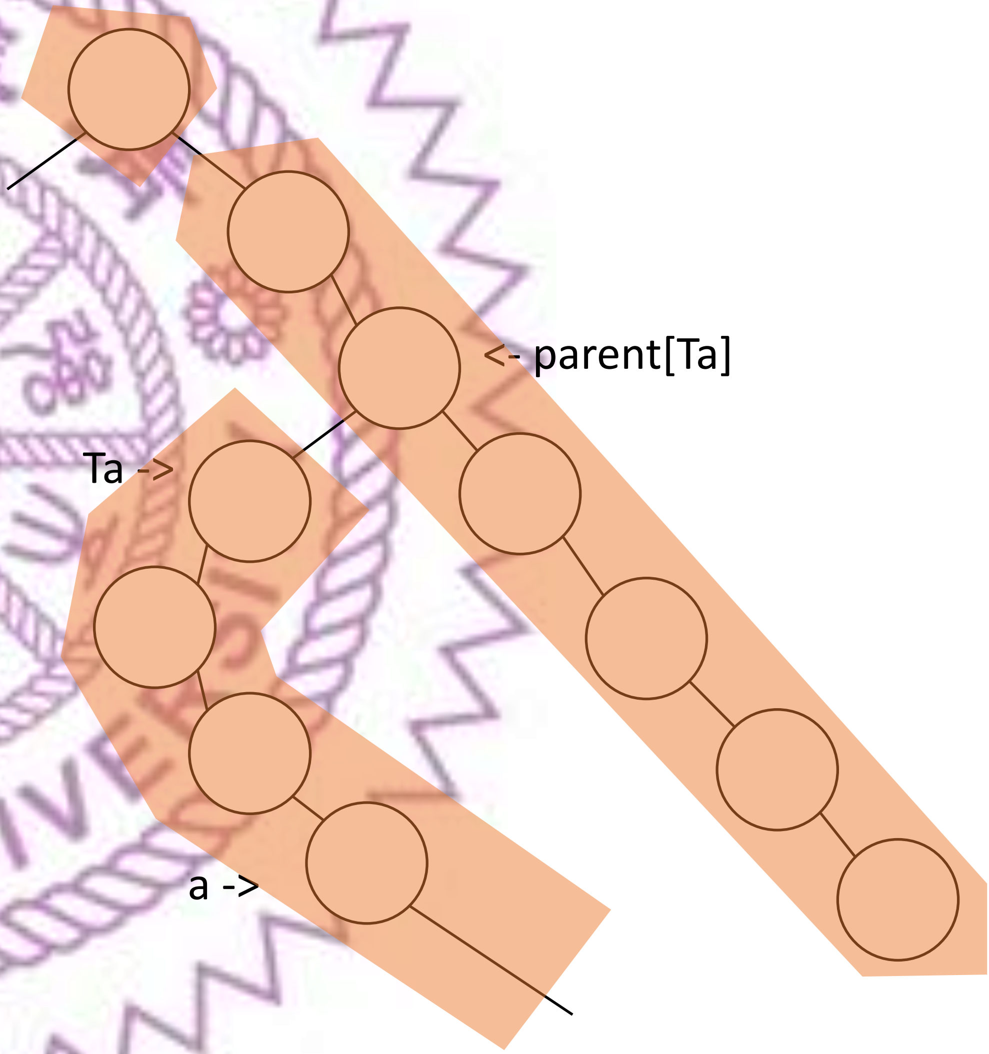
a	b	c	d	e	f	g	h
a	a	c	d	a	c	g	a

```
vector<int> Arr;
vector<int> Top, Idx;
void build_link(int u, int link_top) {
  Arr.emplace_back(u);
  Idx[u] = Arr.size() - 1;
  Top[u] = link_top;
  if (HeavyChild[u] == -1)
    return;
  build_link(HeavyChild[u], link_top);
  for (auto v : Tree[u]) {
    if (v == HeavyChild[u] || v == parent[u])
      continue;
    build_link(v, v);
  }
}
```



樹鏈剖分 – LCA

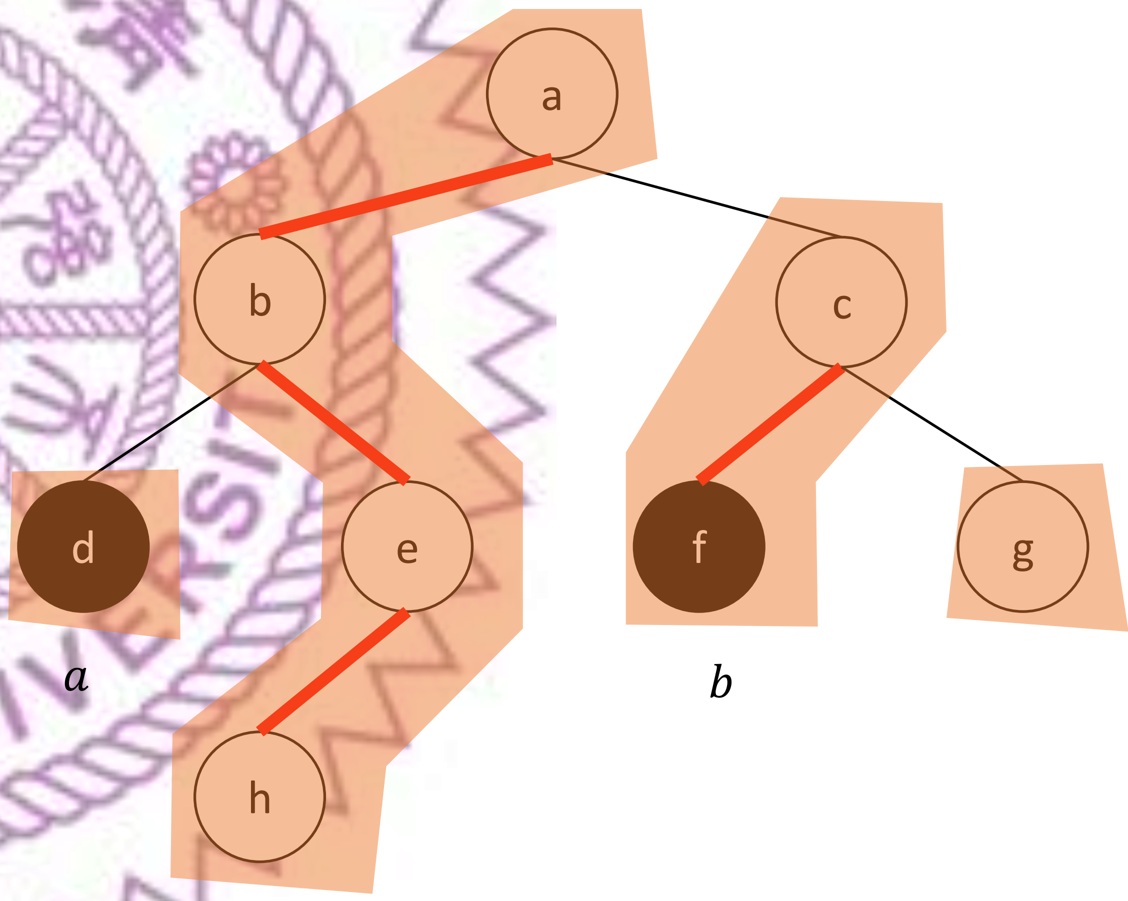
```
int getLCA(int a, int b) {
  int Ta = Top[a], Tb = Top[b];
  while (Ta != Tb) {
    if (level[Ta] < level[Tb]) {
      b = parent[Tb];
      Tb = Top[b];
    } else {
      a = parent[Ta];
      Ta = Top[a];
    }
  }
  if (level[a] > level[b])
    swap(a, b);
  return a;
}
```



樹鏈剖分 – 區間操作

Arr	0	1	2	3	4	5	6	7
	a	b	e	h	d	c	f	g

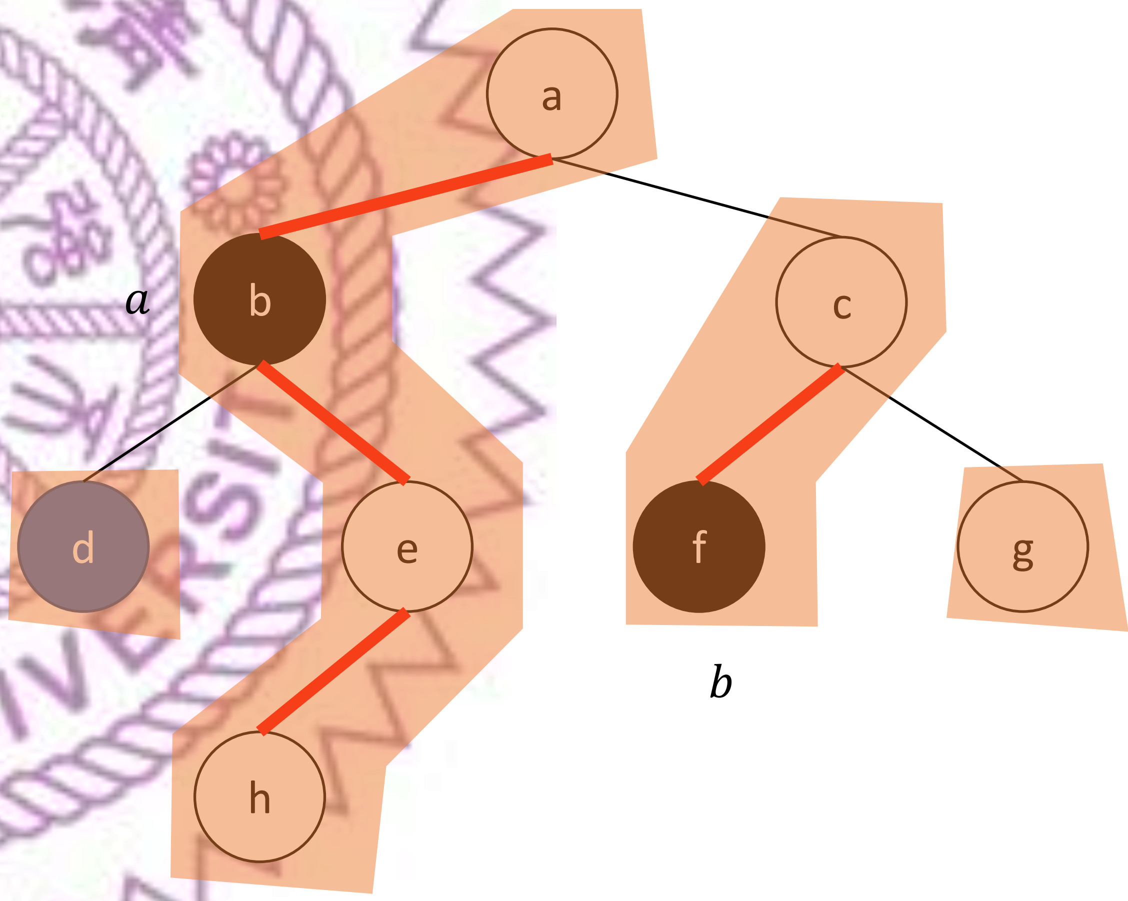
```
int getLCA(int a, int b, auto range_operator) {
  int Ta = Top[a], Tb = Top[b];
  while (Ta != Tb) {
    if (level[Ta] < level[Tb]) {
      range_operator(Idx[Tb], Idx[b]);
      b = parent[Tb];
      Tb = Top[b];
    } else {
      range_operator(Idx[Ta], Idx[a]);
      a = parent[Ta];
      Ta = Top[a];
    }
  }
  if (level[a] > level[b])
    swap(a, b);
  range_operator(a, b);
  return a;
}
```



樹鏈剖分 – 區間操作

```
int getLCA(int a, int b, auto range_operator) {
  int Ta = Top[a], Tb = Top[b];
  while (Ta != Tb) {
    if (level[Ta] < level[Tb]) {
      range_operator(Idx[Tb], Idx[b]);
      b = parent[Tb];
      Tb = Top[b];
    } else {
      range_operator(Idx[Ta], Idx[a]);
      a = parent[Ta];
      Ta = Top[a];
    }
  }
  if (level[a] > level[b])
    swap(a, b);
  range_operator(a, b);
  return a;
}
```

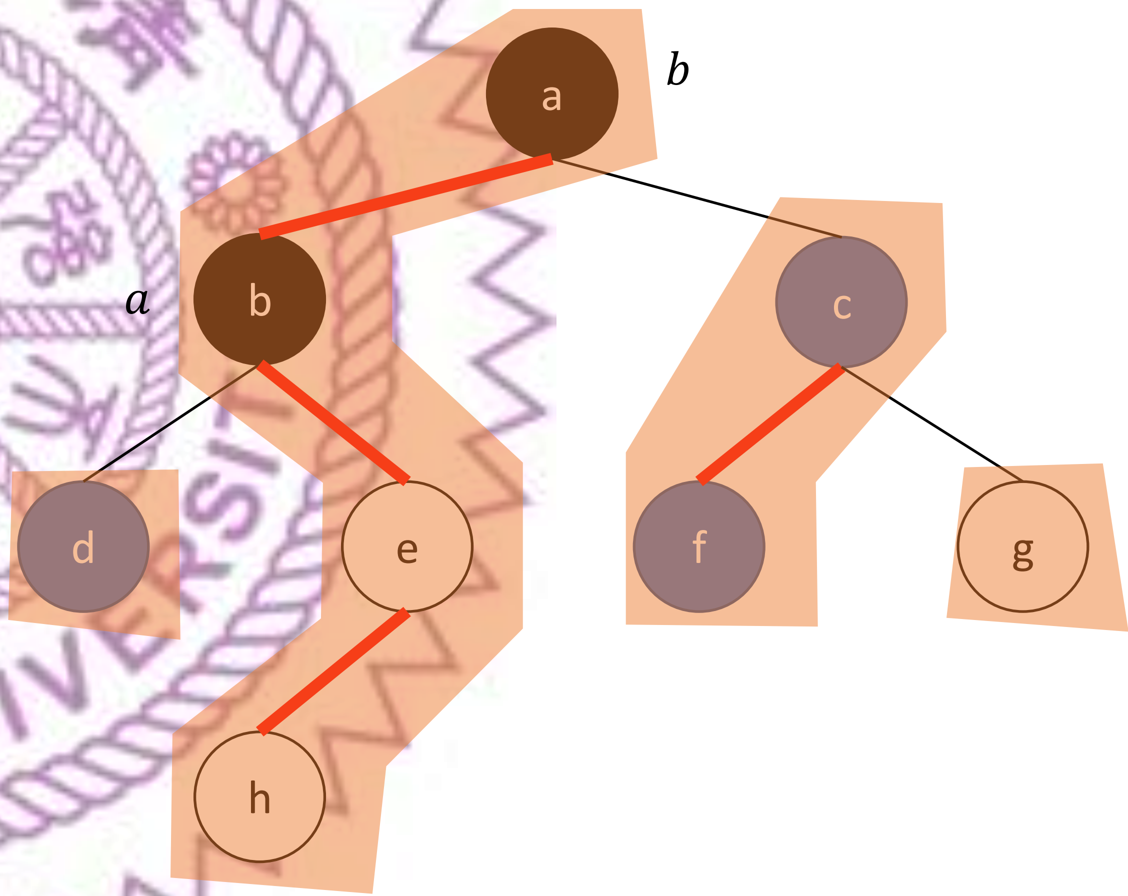
	0	1	2	3	4	5	6	7
Arr	a	b	e	h	d	c	f	g



樹鏈剖分 – 區間操作

Arr	0	1	2	3	4	5	6	7
	a	b	e	h	d	c	f	g

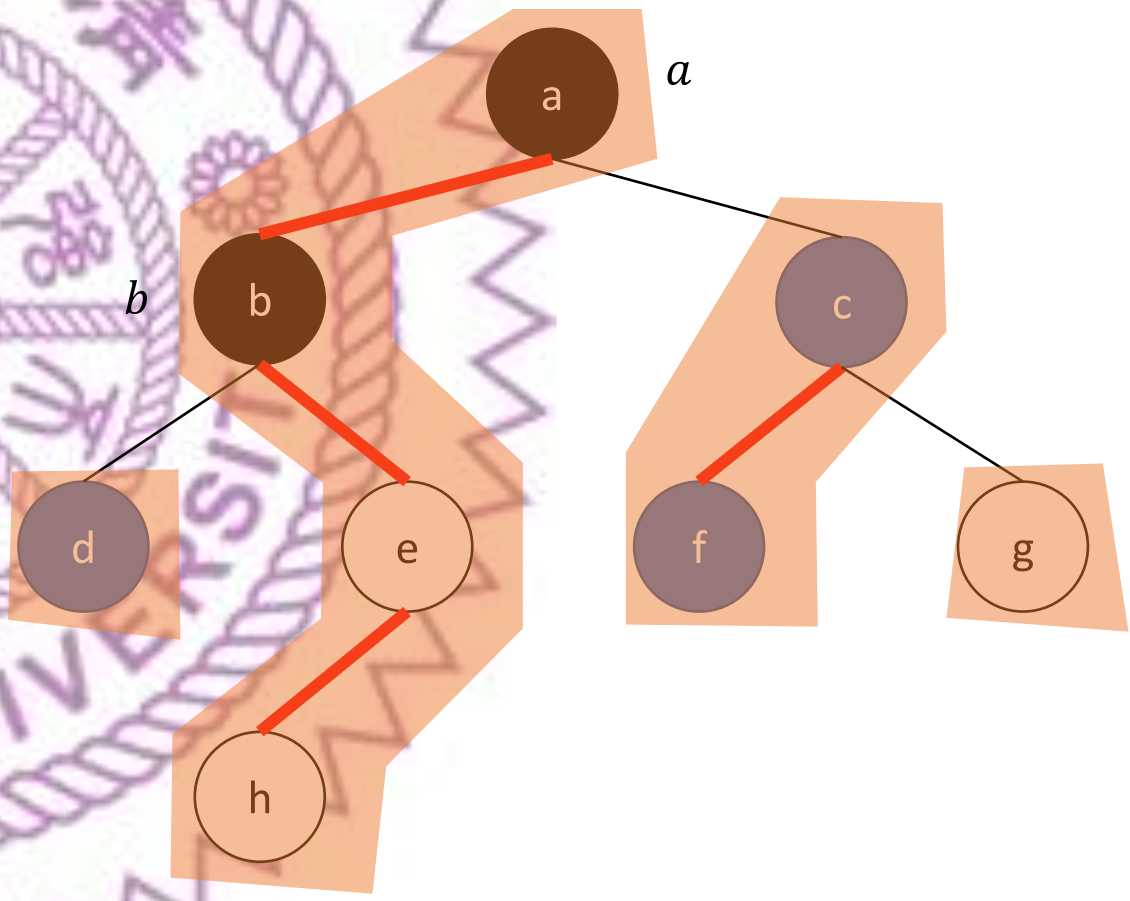
```cpp
int getLCA(int a, int b, auto range_operator) {
  int Ta = Top[a], Tb = Top[b];
  while (Ta != Tb) {
    if (level[Ta] < level[Tb]) {
      range_operator(Idx[Tb], Idx[b]);
      b = parent[Tb];
      Tb = Top[b];
    } else {
      range_operator(Idx[Ta], Idx[a]);
      a = parent[Ta];
      Ta = Top[a];
    }
  }
  if (level[a] > level[b])
    swap(a, b);
  range_operator(a, b);
  return a;
}
```



樹鏈剖分 – 區間操作

```cpp
int getLCA(int a, int b, auto range_operator) {
  int Ta = Top[a], Tb = Top[b];
  while (Ta != Tb) {
    if (level[Ta] < level[Tb]) {
      range_operator(Idx[Tb], Idx[b]);
      b = parent[Tb];
      Tb = Top[b];
    } else {
      range_operator(Idx[Ta], Idx[a]);
      a = parent[Ta];
      Ta = Top[a];
    }
  }
  if (level[a] > level[b])
    swap(a, b);
  range_operator(a, b);
  return a;
}
```

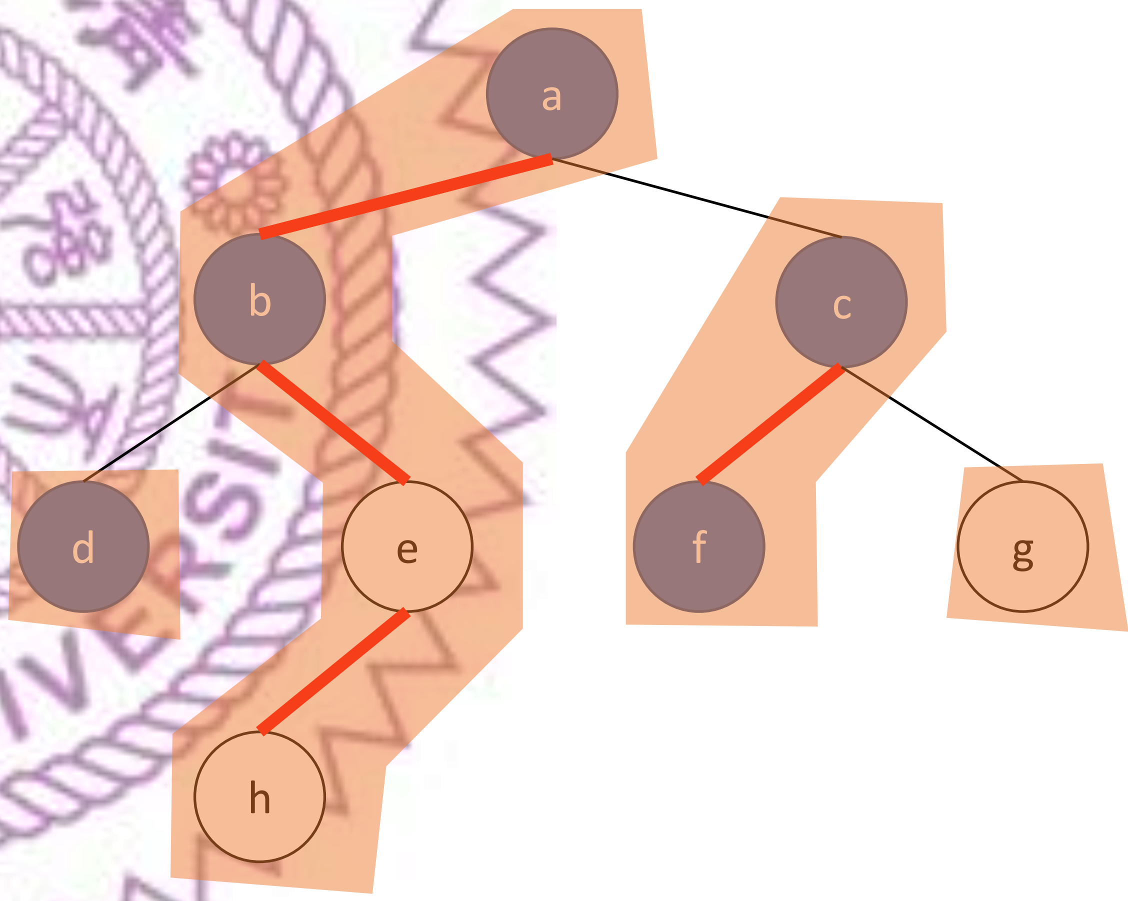
	0	1	2	3	4	5	6	7
Arr	a	b	e	h	d	c	f	g



樹鏈剖分 – 區間操作

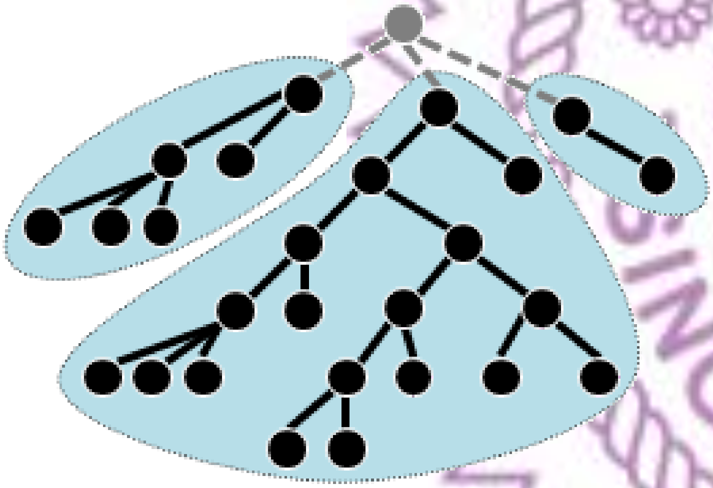
Arr	0	1	2	3	4	5	6	7
	a	b	e	h	d	c	f	g

```
int getLCA(int a, int b, auto range_operator) {
  int Ta = Top[a], Tb = Top[b];
  while (Ta != Tb) {
    if (level[Ta] < level[Tb]) {
      range_operator(Idx[Tb], Idx[b]);
      b = parent[Tb];
      Tb = Top[b];
    } else {
      range_operator(Idx[Ta], Idx[a]);
      a = parent[Ta];
      Ta = Top[a];
    }
  }
  if (level[a] > level[b])
    swap(a, b);
  range_operator(a, b);
  return a;
}
```



$\boxed{\text{利用線段樹維護每次操作 } \boldsymbol{O}(\log^2 n)}$

樹鏈剖分 – 複雜度



- 根到任意點的路徑上，最多只會經過 $\log_2 n$ 個輕邊
- 設 $size(x)$ 表示 x 為根的子樹節點數量。
- $size(HeavyChild[x])$ 是 x 所有小孩最大的那個
因此若 y 是 x 的小孩且 $y \neq HeavyChild[x]$ ，則：

$$size(y) \leq \frac{size(x)}{2}$$

$\boxed{\text{同時還能處理子樹資訊 } \boldsymbol{O}(\log n)}$

Centroid Decomposition

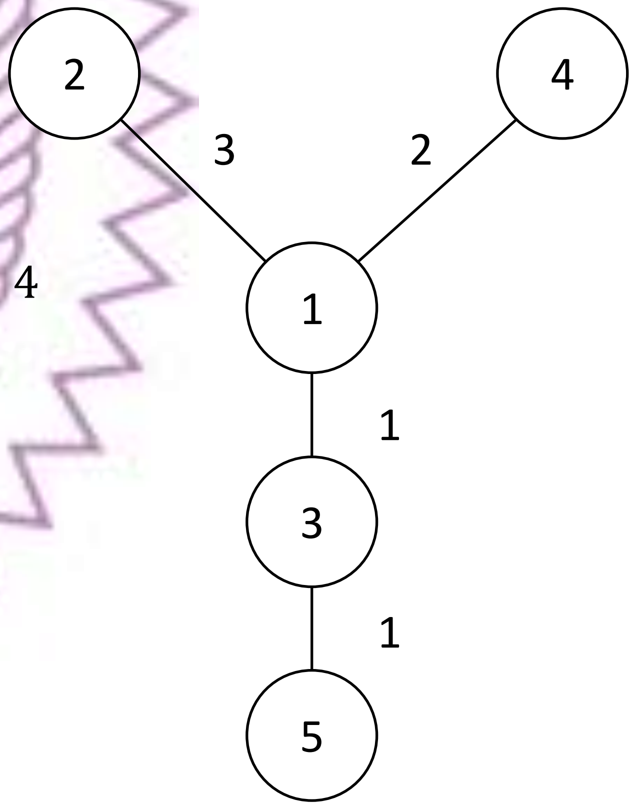
樹重心分治

經典題 POJ 1741

- 給你一個 n 個點的樹，編號 $1\sim n$，每條邊給定一個長度
- 問你整棵樹上長度不超過 k 的路徑有幾條
- $1 \le n, k \le 10^6$

有 8 條
$1 \rightarrow 2$
$1 \rightarrow 4$
$1 \rightarrow 3$
$1 \rightarrow 5$
$2 \rightarrow 3$
$3 \rightarrow 4$
$3 \rightarrow 5$
$4 \rightarrow 5$

$k = 4$



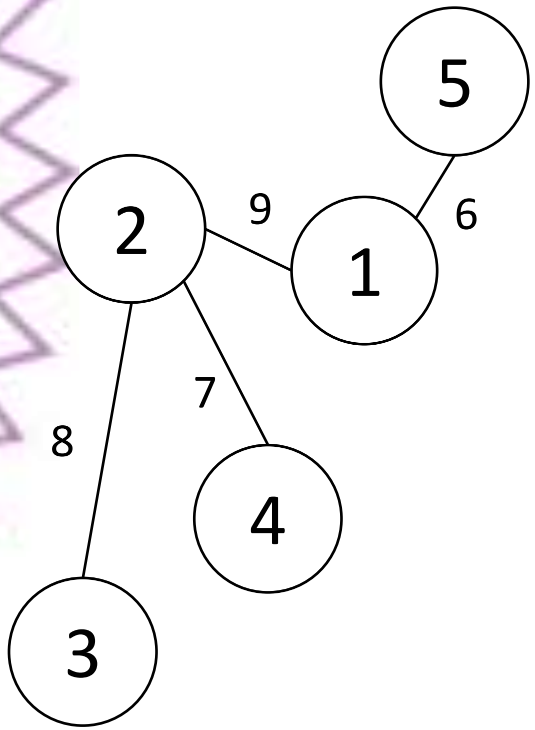
複習：如果邊有權重

n 個點

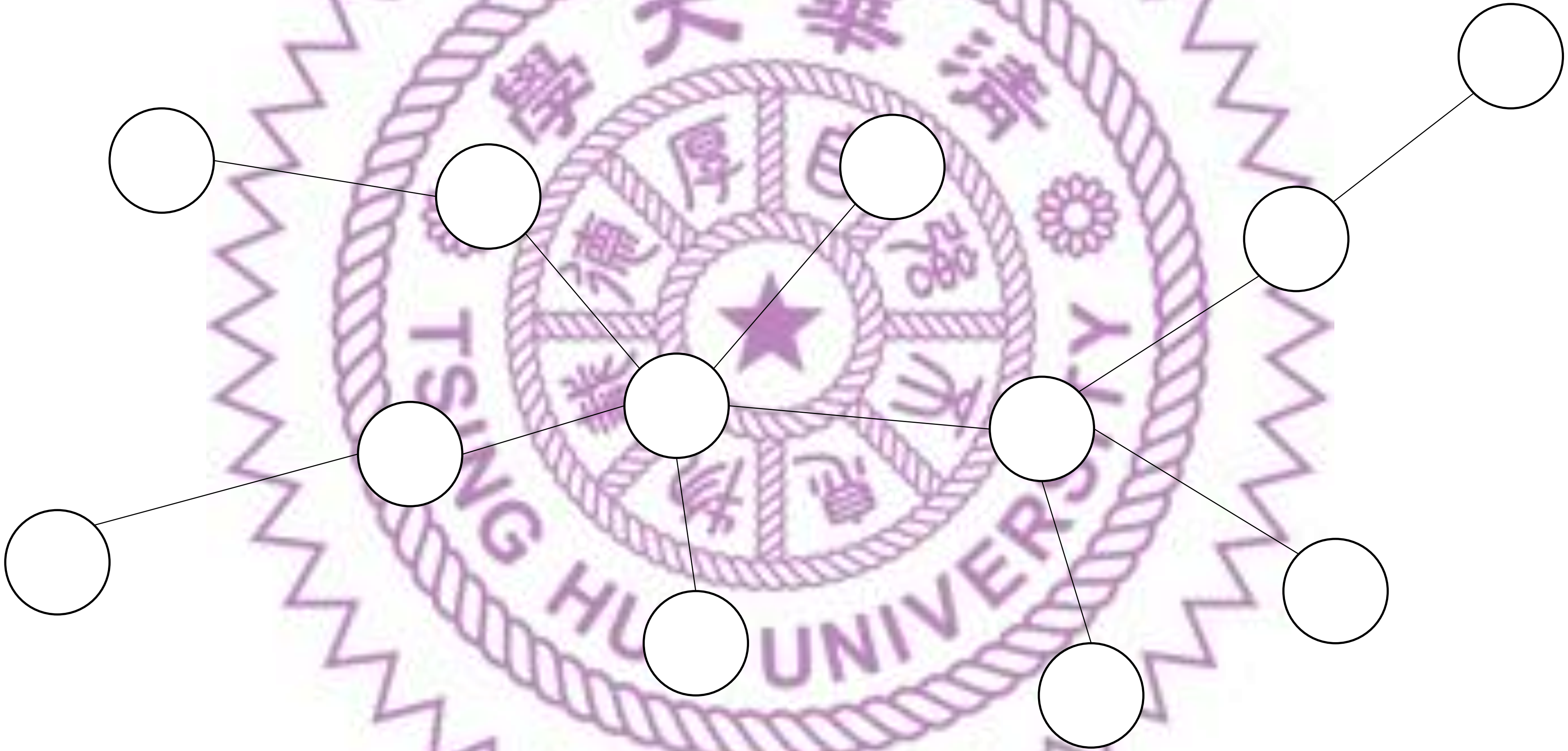
$n-1$ 條邊

```
5
1 2 9
2 3 8
2 4 7
1 5 6
```

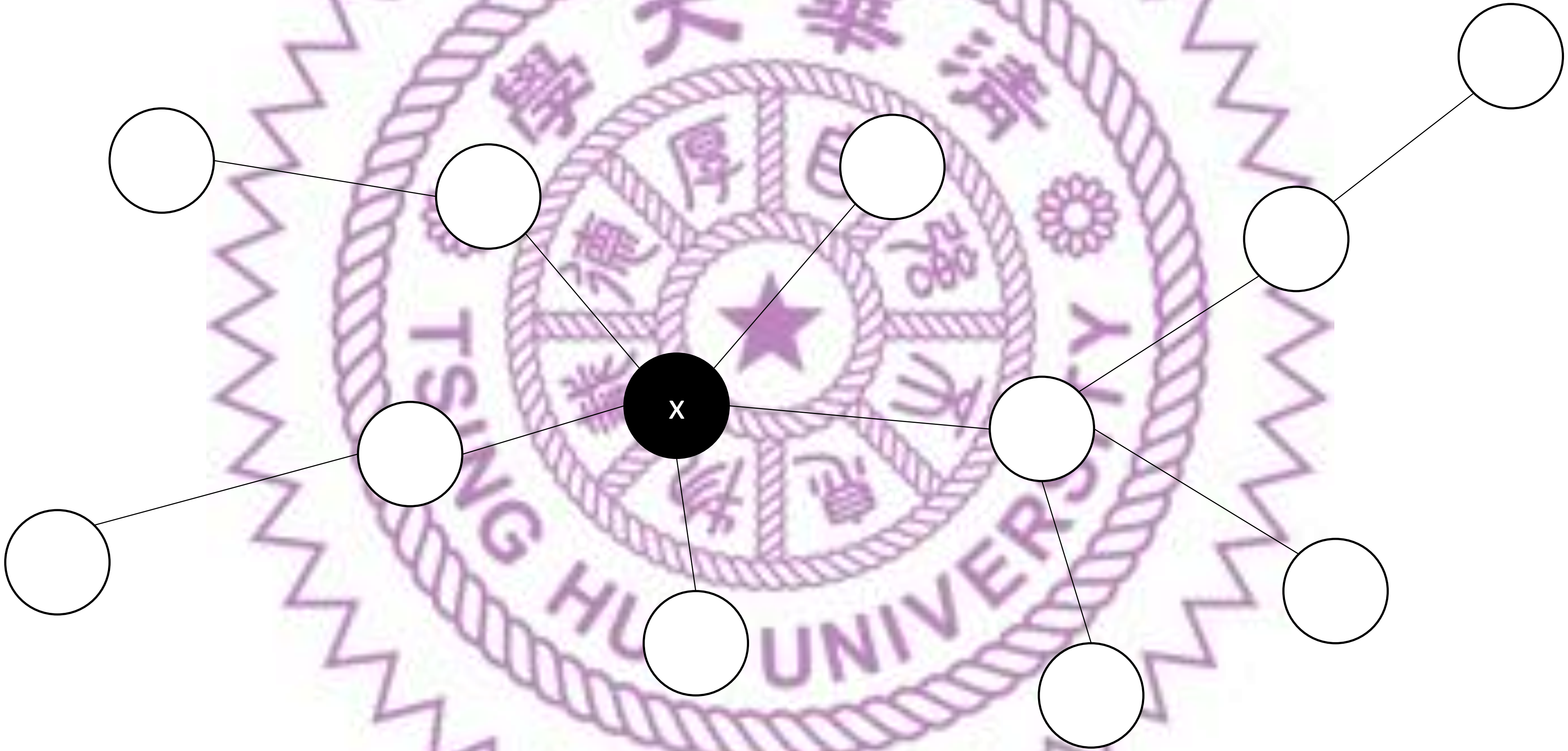
```cpp
vector<vector<pair<int, int>>> Tree;
int n;
cin >> n;
Tree.assign(n + 1, {});
for (int i = 0; i < n - 1; ++i) {
  int u, v, cost;
  cin >> u >> v >> cost;
  Tree[u].emplace_back(v, cost);
  Tree[v].emplace_back(u, cost);
}
```



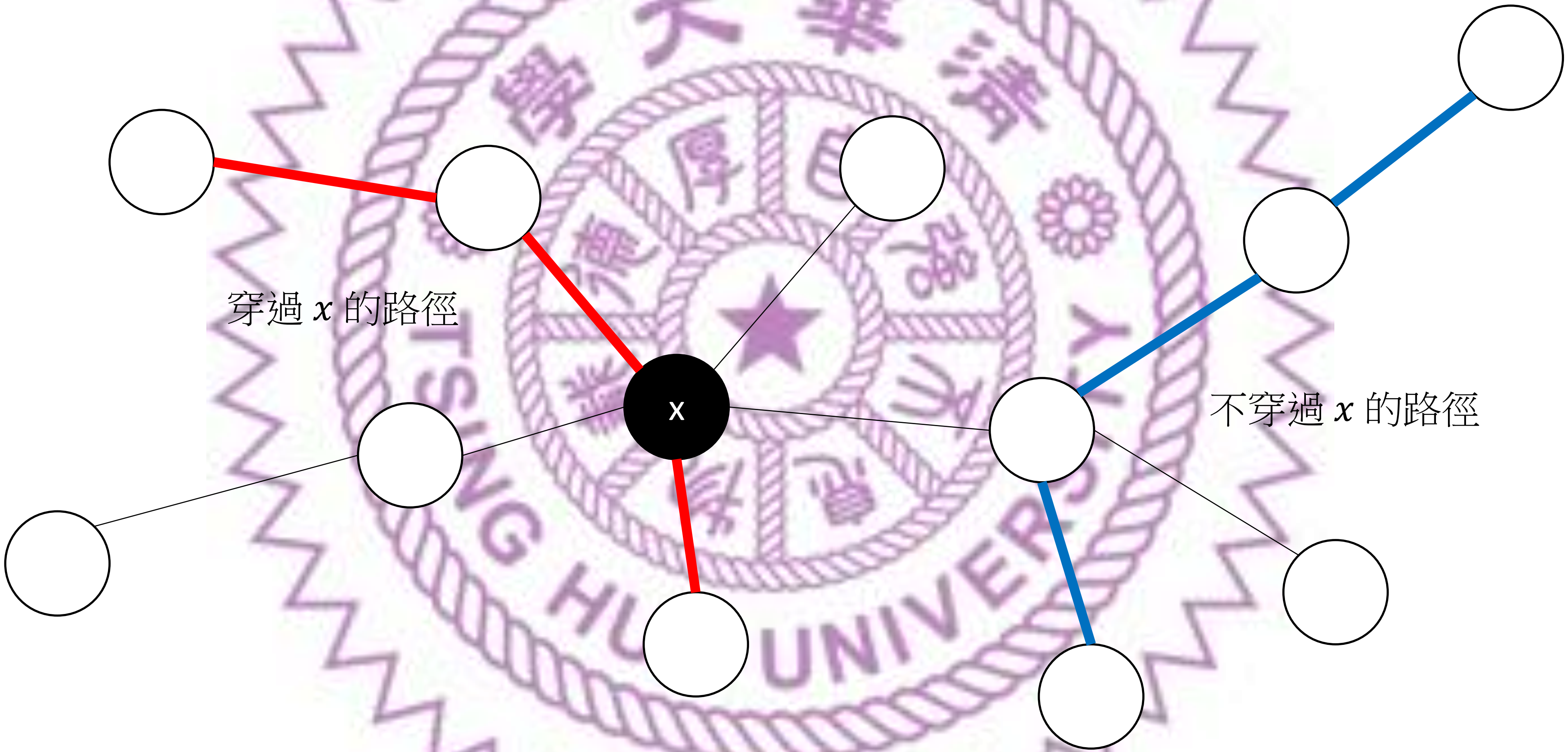
想法



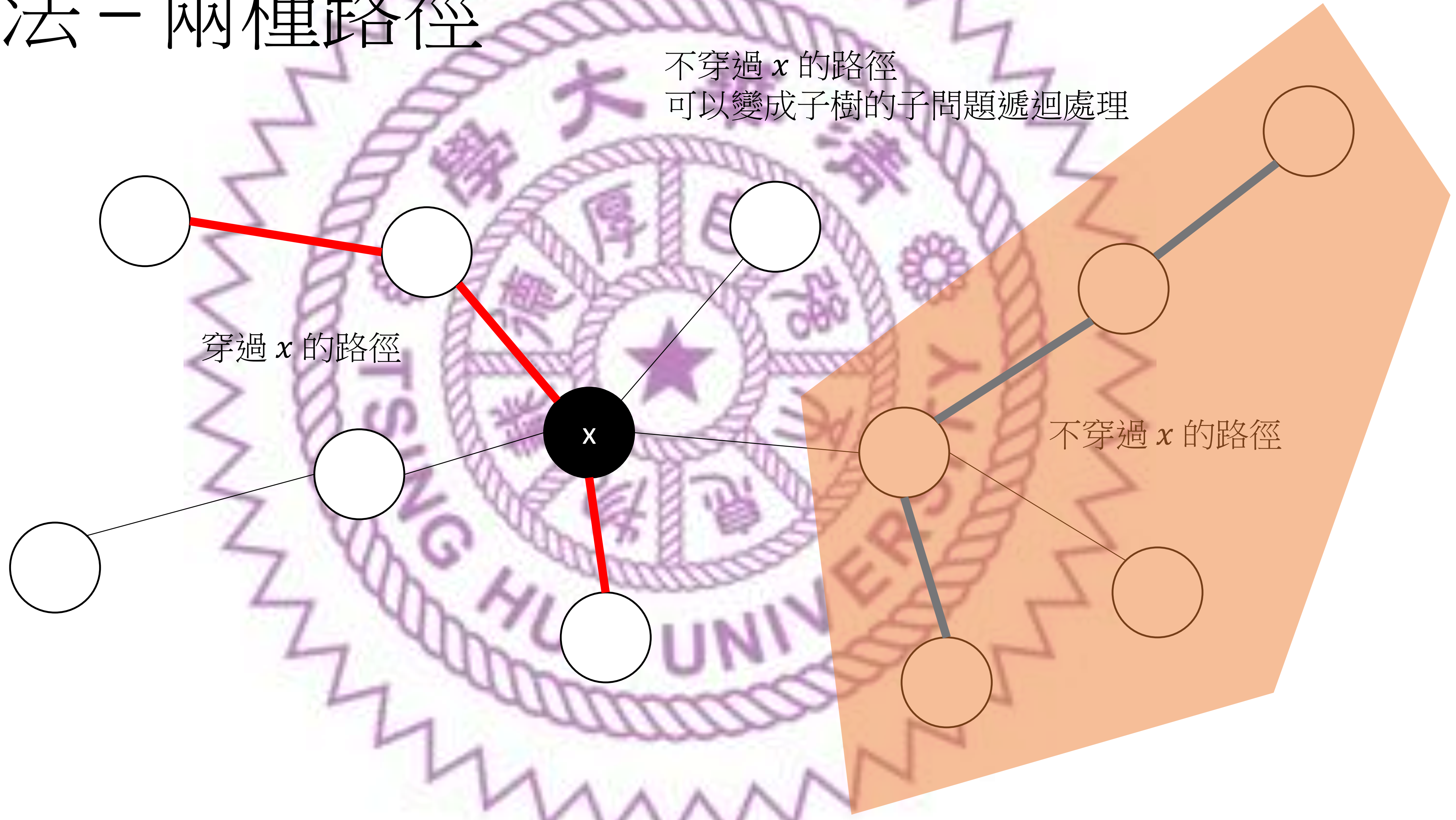
想法 – 隨便選一個點



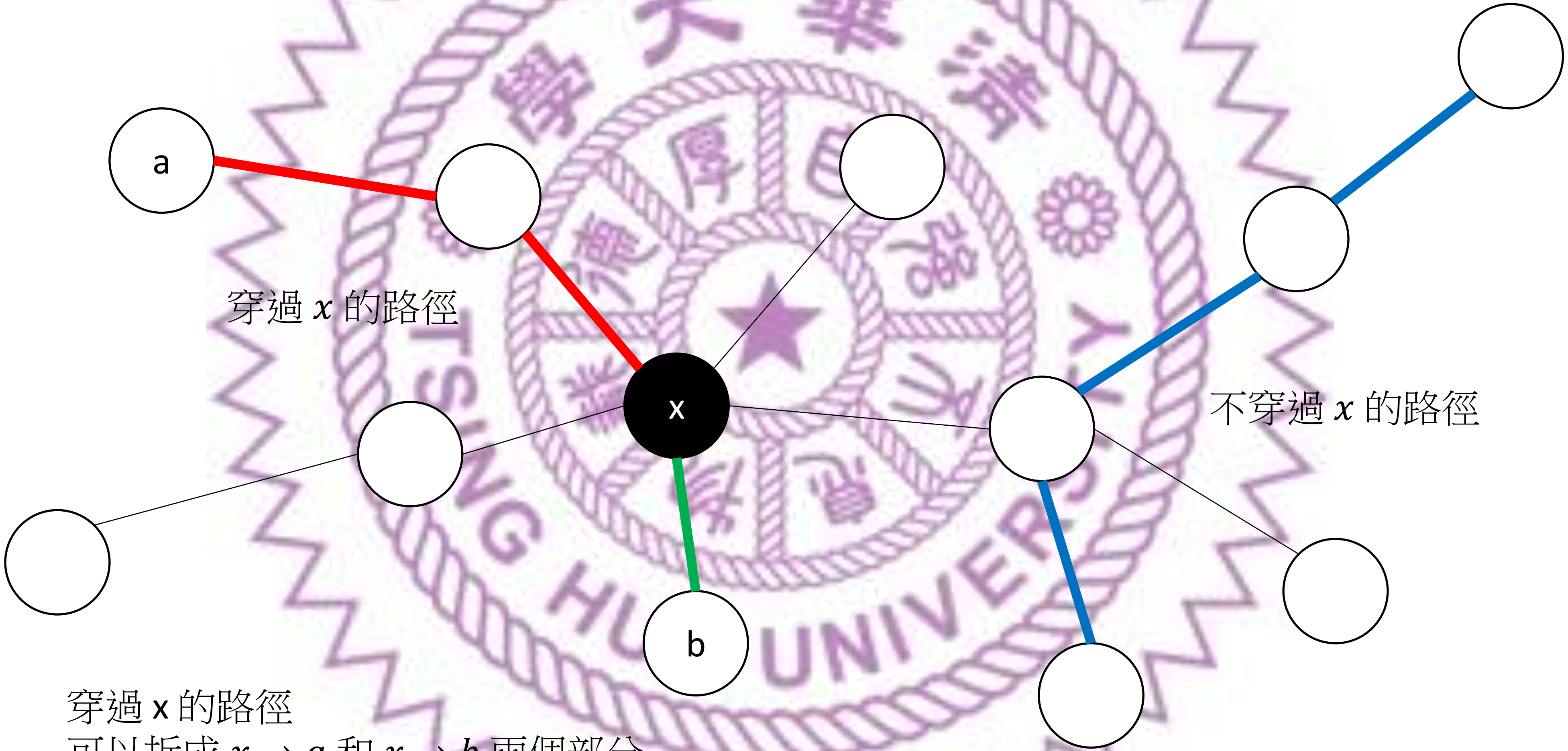
想法 – 兩種路徑



想法 – 兩種路徑



想法 – 兩種路徑



穿過 x 的路徑
可以拆成 $x \rightarrow a$ 和 $x \rightarrow b$ 兩個部分

找出從節點 u 出發到葉子的所有路徑

visit 限制了現在處理的樹的範圍
未來說明為何要限制

```
vector<int> Distance;
void getDistance(int D, int u, int parent = -1) {
  Distance.emplace_back(D);
  for (auto [v, cost] : Tree[u]) {
    if (v != parent && !visit[v])
      getDistance(D + cost, v, u);
  }
}
```

利用雙指標計算穿過 x 的合法路徑數

```
int cal(int x) {
  Distance.clear();
  getDistance(0, x);
  sort(Distance.begin(), Distance.end());
  int L = 0, R = Distance.size() - 1;
  int ans = 0;
  while (L < R) {
    while (L < R && Distance[L] + Distance[R] > k)
      --R;
    ans += R - (L++);
  }
  return ans;
}
```

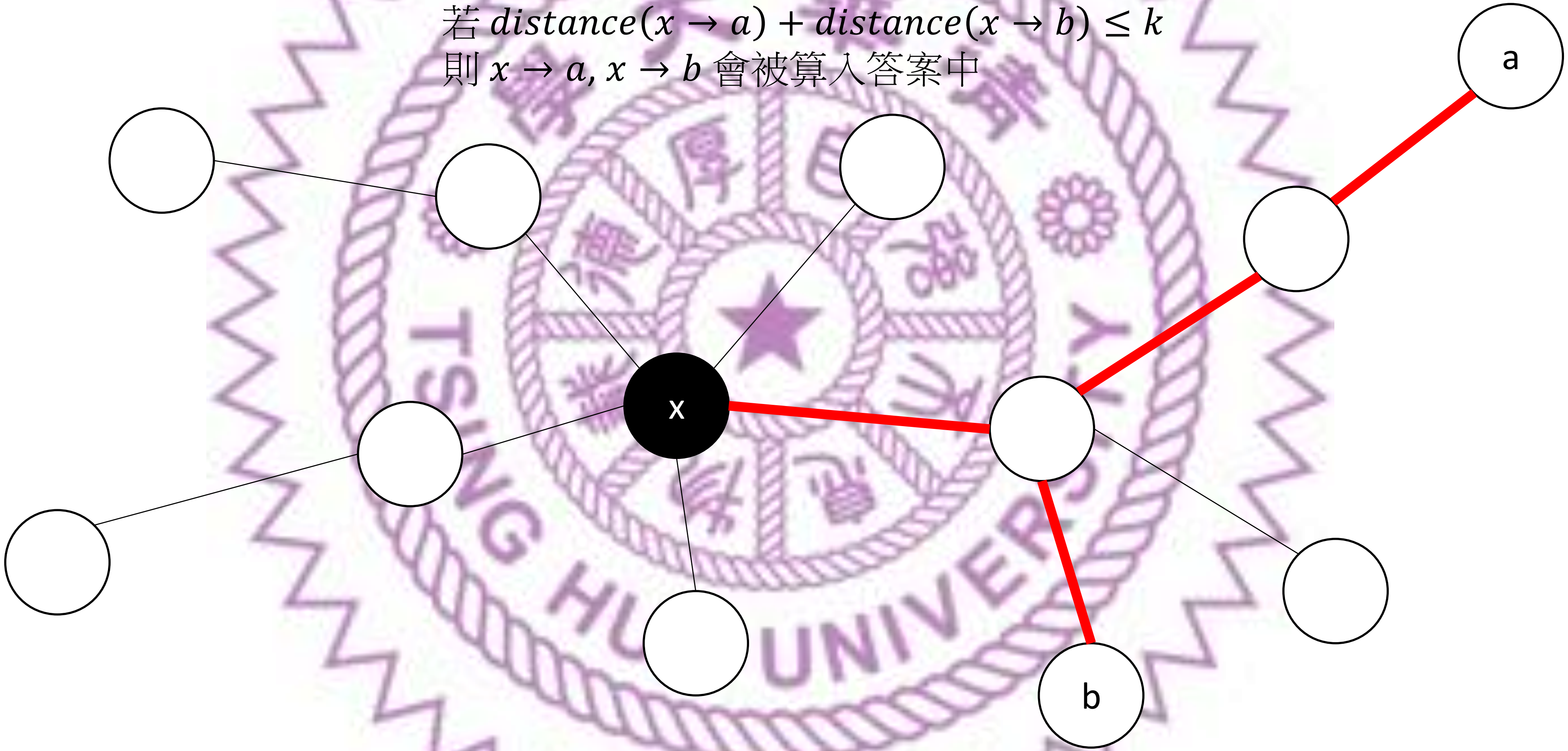
最後好好找 x 就能讓遞迴不會太深

```
int cen            position(
  int
  int an
  visit[x]
  for (auto [v              ) {
    if (vis
      co
    ar               eco
  }
  return
}
```

想法正確
但計算過程有問題

同一個子樹 x 連出去的邊會被重複計算

若 $distance(x \to a) + distance(x \to b) \le k$
則 $x \to a, x \to b$ 會被算入答案中



利用雙指標計算穿過 x 的合法路徑數

```
int cal(int x, int base) {
  Distance.clear();
  getDistance(base, x);
  sort(Distance.begin(), Distance.end());
  int L = 0, R = Distance.size() - 1;
  int ans = 0;
  while (L < R) {
    while (L < R && Distance[L] + Distance[R] > k)
      --R;
    ans += R - (L++);
  }
  return ans;
}
```

正確的遞迴

```
int centroid_decomposition(int u) {
  int x = 找出讓遞迴不會太深的x();
  int ans = cal(x, 0);
  visit[x] = true;
  for (auto [v, cost] : Tree[x]) {
    if (visit[v])
      continue;
    ans -= cal(v, cost);
    ans += centroid_decomposition(v);
  }
  return ans;
}
```

找出讓遞迴不會太深的 x

- 有一種點以其為根，會使得最大的子樹節點數量最小 $\left(\leq \frac{n}{2}\right)$
- 我們以前學過的樹重心

計算樹重心 (儲存在 second)

要輸入當前子樹的總節點數量

```
vector<int> size;
pair<int, int> tree_centroid(int u, int parent, const int sz) {
  size[u] = 1;
  pair<int, int> ans(INT_MAX, -1);
  int max_size = 0;
  for (auto [v, cost] : Tree[u]) {
    if (v == parent || visit[v])
      continue;
    ans = min(ans, tree_centroid(v, u, sz));
    size[u] += size[v];
    max_size = max(max_size, size[v]);
  }
  max_size = max(max_size, sz - size[u]);
  return min(ans, make_pair(max_size, u));
}
```

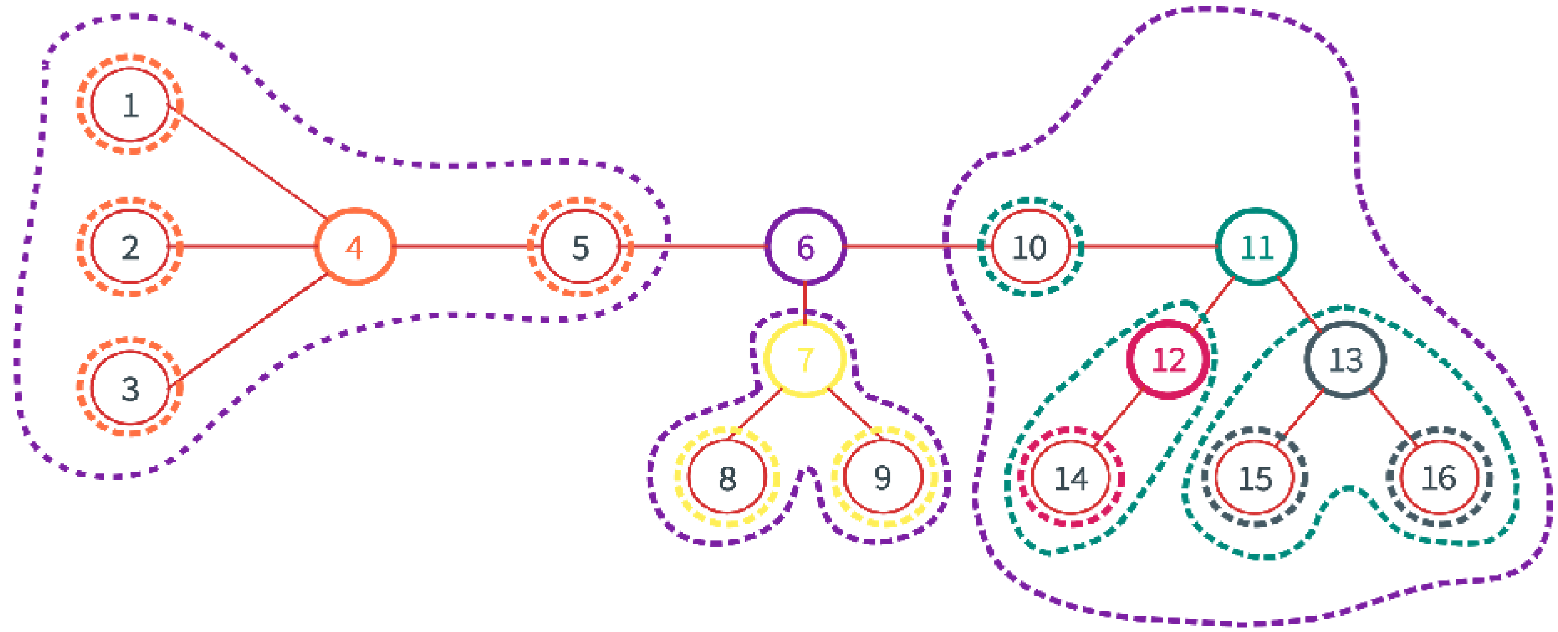
最終關鍵程式碼

```cpp
int centroid_decomposition(int u, int sz) {
  int center = tree_centroid(u, -1, sz).second;
  int ans = cal(center, 0);
  visit[center] = true;
  for (auto [v, cost] : Tree[center]) {
    if (visit[v])
      continue;
    ans -= cal(v, cost);
    ans += centroid_decomposition(v, size[v]);
  }
  return ans;
}
```

```cpp
centroid_decomposition(1,n);
```

樹重心性質

將重心刪除後，切出來的每棵樹節點數量都 $\leq \frac{n}{2}$



樹重心分治複雜度

- 根據樹重心性質，遞迴最多有 $\log_2 n$ 層
- 每一層都花 $O(n + n\log n)$ 計算答案
- 該問題總時間複雜度是 $O(n\log^2 n)$