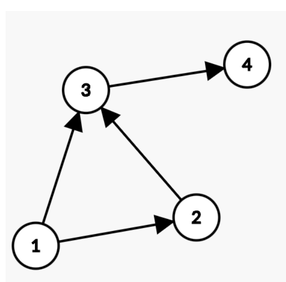
2024 DS Summer Lab9 : Graph and Tree

一、圖的基礎概念

1. **有向圖、無向圖：**
   無向圖中的邊為「無向邊」。無向邊 (u,v) 代表從 u 可以到達 v，反之也可到達。(u,v) 也可記為 $u \to v$。
   有向圖中的邊為「有向邊」。有向邊 (u,v) 代表從 u 可以到達 v。

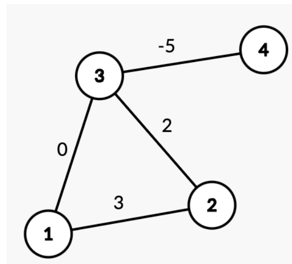


有向圖

2. 相鄰（adjacent）：當節點 u,v 間存在一條邊，則稱 u,v 相鄰
3. 度數：點 u 的度數為相鄰節點的數量
   a. 入度：點 u 的入度為指向 u 的邊的數量

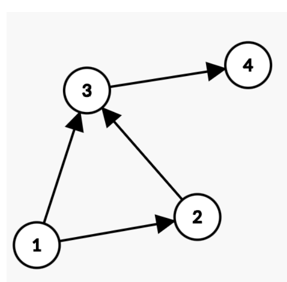
b. 出度：點 v 的出度為指向 v 的邊的數量

4. **邊權：一條邊可以有數值 w，稱 w 為該邊的「邊權」。邊權可能代表兩點間的距離、花費、最大流量等，其意義視用途而定。**



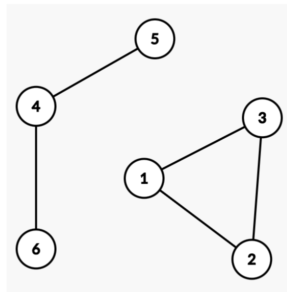
有邊權的無向圖

5. **點權：一個節點可以有數值 w，稱 w 為該點的「點權」。點權可能代表該點的花費等，其意義視用途而定。**
6. 重邊：當點 u, v 間有不只一條邊，稱 u, v 間有「重邊」。
7. **路徑：有起點 s 和終點 t，從 s 走到 t 途中經過的邊的序列。以下圖為例，** $(1 \to 2, 2 \to 3, 3 \to 4)$ 為 1 到 3 的路徑之一。



8. **環：一個路徑的起點和終點相同，則該路徑稱為「環」。**
9. 迴路：一個路徑經過了所有的點洽一次，且起點和終點相同，則該路徑稱為「迴路」。
10. 自環：一條邊的兩端為相同節點，稱為「自環」

11. **連通：若點 s 存在一條路徑到點 t，則稱 s 和 t 「連通」。**



$\{1,2,3\},\{4,5,6\}$ 各為一個連塊

1. **連通塊：如果點集 S 中的所有節點都互相連通，稱 S 為「連通塊」。**
2. **簡單圖：若圖 G 不存在自環和重邊，則稱 G 為「簡單圖」。**
3. 子圖：若圖 G 的邊和節點，皆為圖 G 的子集，則稱 C 為 G 的「子圖」。
4. 反圖：將有向圖 G 的每一條邊反過來（原本是 $u \to v$ 改成 $v \to u$），則新的圖和 G 互為反圖。
5. 完全圖：若無向簡單圖 G 滿足任意兩節點皆存在一條邊，稱 G 為完全圖。
6. 補圖：圖 G 的補圖 cG，和 G 有一樣的節點，而圖 cG 中 (u,v) 有邊若且唯若 (u,v) 在 G 中沒有邊。把 G 和 cG 加在一起，可以得到一張完全圖。
7. 二分圖：一張圖可以將節點拆成互斥的集合 U,V，使得集合中沒有相鄰的節點，則該圖為二分圖。

二、圖的儲存

常見圖的儲存有三種方式：直接存邊、鄰接矩陣和鄰接串列。

無向圖中的邊 (u,v) 代表 u 和 v 可以互相抵達，因此儲存時我們會將邊無向邊 (u,v) 拆解成兩條有向邊 $u \to v$ 和 $v \to u$。

鄰接矩陣

用一個 $|V| \times |V|$ 的陣列 G 表示圖。若存在邊 $u \to v$，記 $G[u][v] = 1$，否則 $G[u][v] = 0$。若 $u \to v$ 有邊權 w，可記 $G[u][v] = w$。

```
const int N = 1e3; // 該題測資中，最多 N 個節點

int n; // |V| = n
int G[N][N]; // 若 G[i][j] = 1，代表 i -> j
```

加入一條邊 `u -> v`：

```
G[u][v] = 1;
```

遍歷 `u` 的相鄰節點，複雜度：

```
for (int v = 1; v<=n; v++){
    if (G[u][v] == 1){
        // do something...
    }
}
```

實際存法可以依照需求調整，例如若有 $u \to v$ 且邊權為 w，記 $G[u][v] = w$；否則 $G[u][v] = INF$。

鄰接串列

用一個陣列 G 表示圖， $G[u]$ 為所有和 u 相鄰的點。通常 $G[u]$ 為一個可變長度陣列，實作上使用 `vector`。

```
const int N = 1e5; // 該題測資中，最多 N 個節點

int n; // |V| = n
vector<int> G[N]; // G[i] 為一 vector，vetor 當中儲存所有和 i 相鄰的
```

加入一條邊 `u -> v` ：

```
G[u].push_back(v);
```

遍歷 `u` 的相鄰節點：

```
for (auto v: G[u]){
    // do something...
}
```

直接存邊

直接將每一條邊的儲存在陣列裡。

```
struct Edge {
    int u, v; // u -> v
```

```
};

vector<Edge> e;
```

加入一條邊 `u -> v` ：

```
e.push_back( {u, v} );
```

我們通常不使用此方法來尋找 `u` 的相鄰節點。

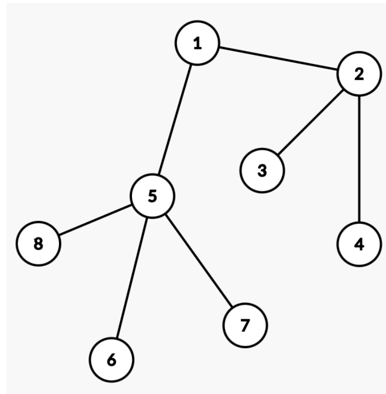
方法優劣分析

最常見的需求是將一張圖的每個點、每條邊遍歷過一次。若使用鄰接矩陣，對於每個點都需要 $O|V|$ 的時間，總複雜度為 $O|V|^2$。若使用鄰接矩陣，儘管對於一個點，遍歷所有鄰居的最糟時間為 $O(|V|)$，但經觀察可發現，每條邊至多被存取 1 到 2 次，因此總複雜度為 $O(|V|+|E|)$。**大部分情況下，我們使用鄰接串列來儲存一張圖。**

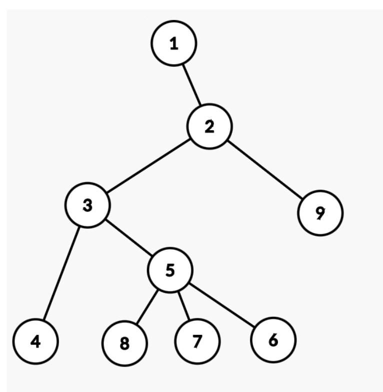
三、樹

「無根樹」為無向圖的一個特例。當一個連通無向圖沒有環，便稱「樹」。無根樹還有以下的性質，並且這些性質同時可以為樹的定義：

- 有 n 個節點， $n-1$ 條邊的連通圖
- 任兩節點間有恰一條簡單路徑
- 沒有環，並且在加入邊 $(u,v), u \neq v$ 後一定會形成一個環



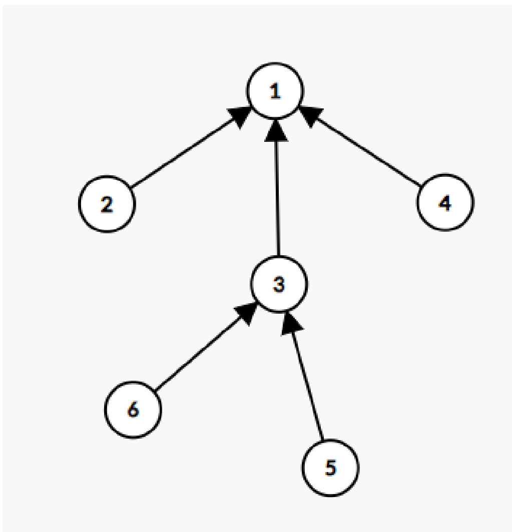
我們可以為樹定義一個根，稱為「有根樹」，當規定了根節點，節點間就會產生上下階層的關係。下圖是根節點為 1 的有根樹，我們以節點 3 為例說明有根樹的概念和名詞：



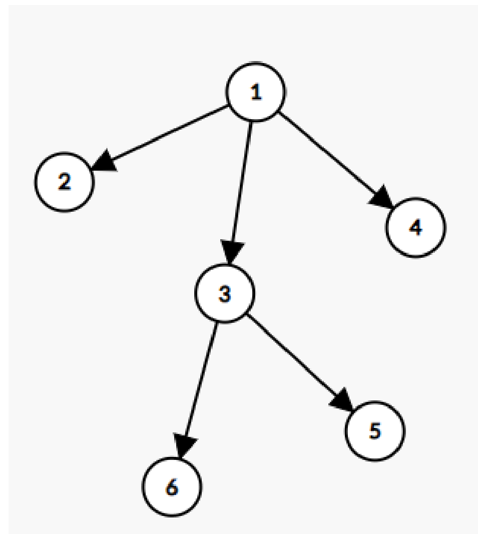
- 2 為 3 的「父節點」
- $4, 5$ 為 3 的「子節點」
- $4, 5, 6, 7, 8$ 皆為 3 的「後代」，而 3 和其後代構成 3 的「子樹」
- 「葉節點」為沒有後代的節點，如 $4, 6, 7, 8, 9$
- 「深度」為根節點到自己的距離，例如 3 的深度為 2
- 「高度」為後代中到自己的最大距離，例如 3 的高度為 2

樹的儲存

前面說到，樹是無向圖的特例，而無向圖的儲存方式是一條的兩個方向各存一次。在儲存樹「有根樹」的時候，我們不一定要這樣做，而是：一、只存指向父的邊，二、只存指向子的邊。那麼這時樹就變成類似有向樹的東西，如圖所示。



指向父親的邊



指向兒子的邊

只存指向父親的邊，也就是每個節點都只要記錄自己的父親是誰，而父親只有一個。因此可以用一個陣列 $p[i]$ 來代表樹。這種方式常常在輸入中來表示一棵有根樹。

加入一條邊 (u, v)， v 為父親：

```
p[u] = v
```

對於第二種方式，只存指向兒子的邊，可以利用上面提到的相鄰串列就可以了。