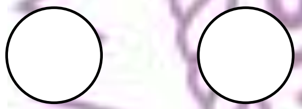
Graph

日月卦長

定義

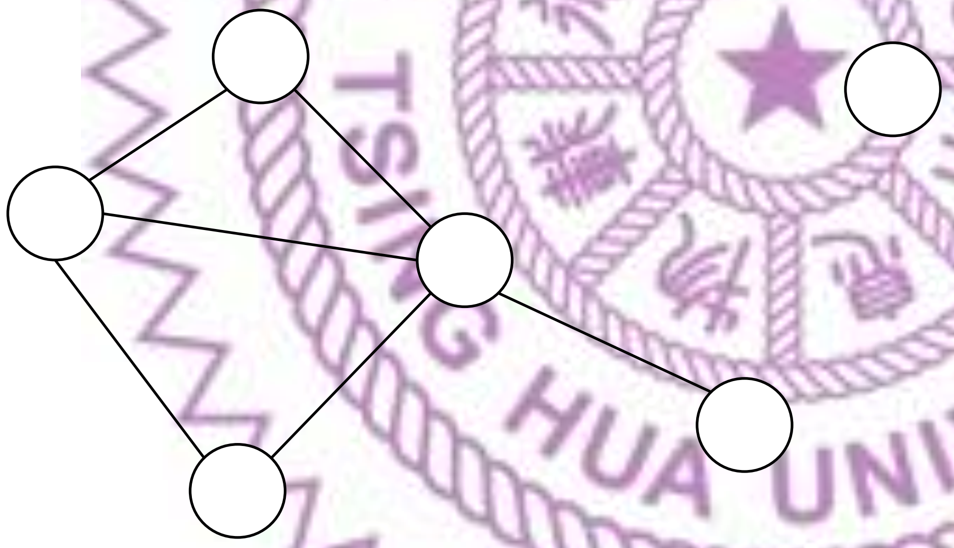
- 基本元素
  - 點和邊
- 點(vertex)



- 邊(edge)

定義

- 圖：點的集合加邊的集合
  $G = (V, E)$，這裡 V 是點集合、E 是邊集合



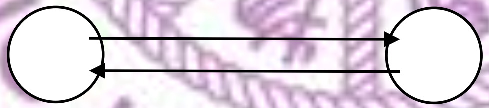
定義

- 有向邊、無向邊

定義

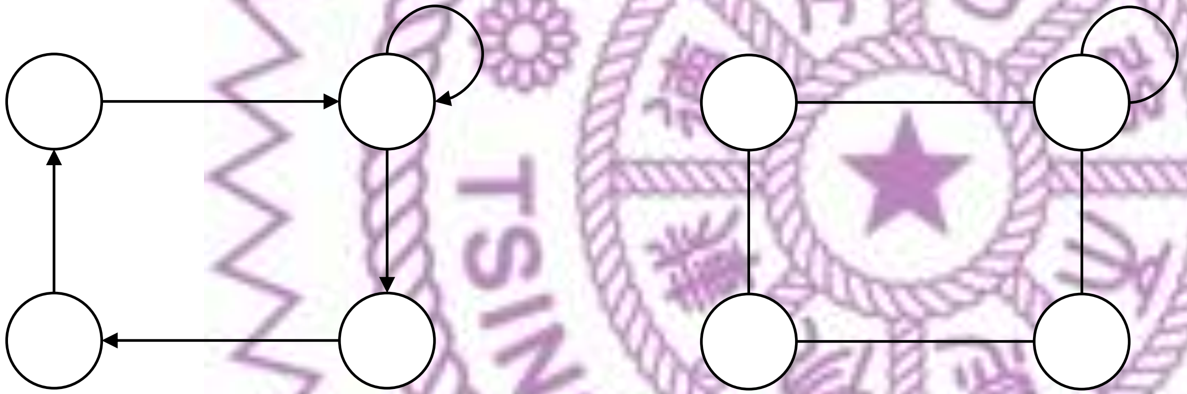
- 有向邊、無向邊

可以想成兩個方向相反的有向邊



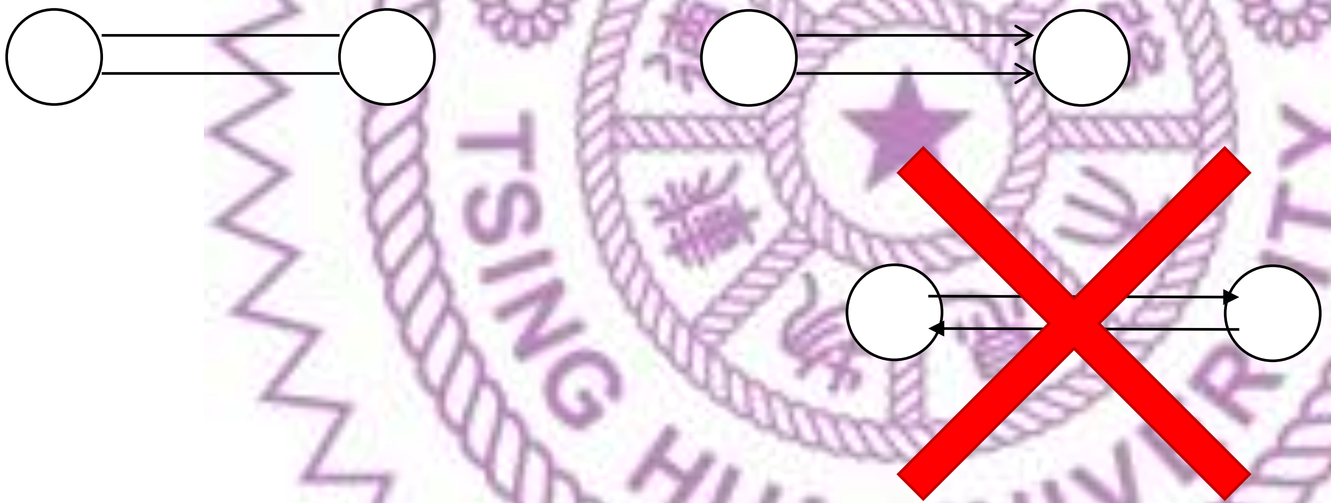
定義

- 有向圖、無向圖



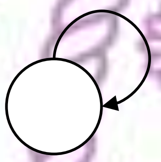
定義

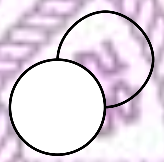
- 重邊



定義

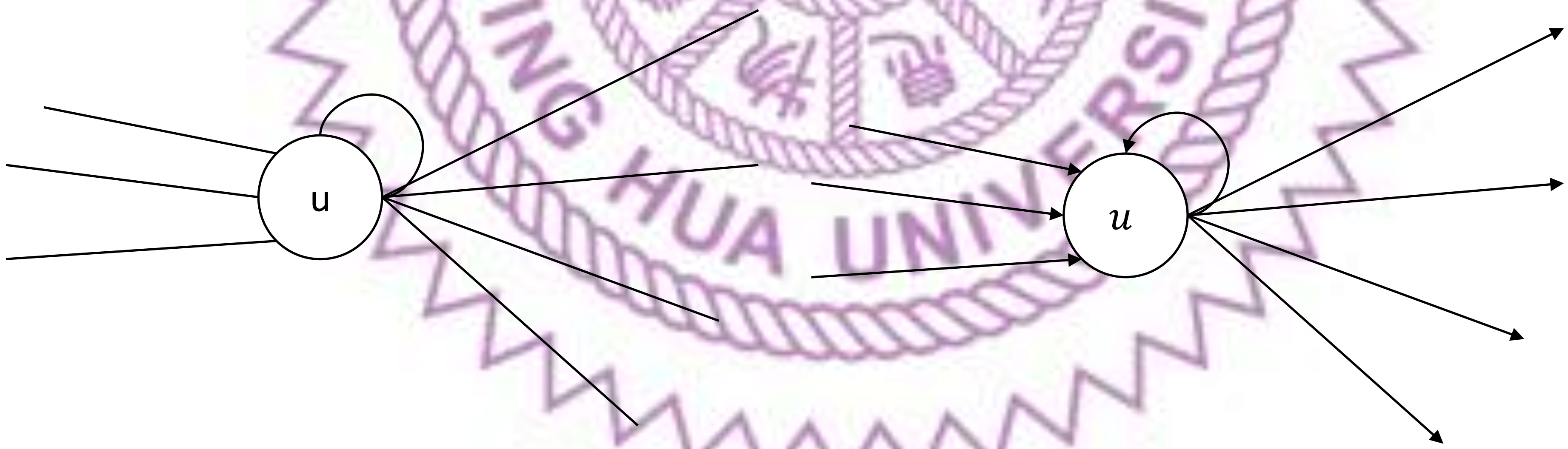
- 自環 (loop)





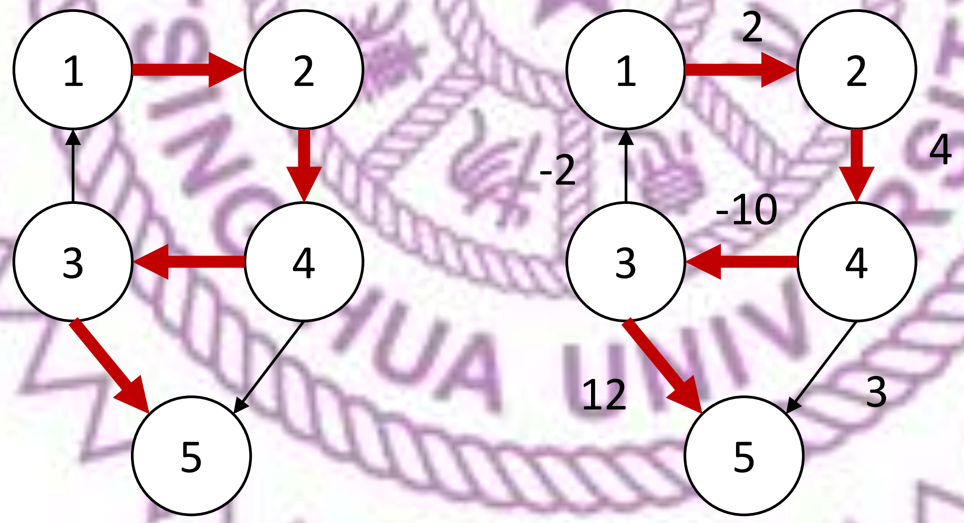
定義

- 度(degree)：和一個點 u 有相關的點的數量
  - 無向圖： 連到 u 這個點的邊數
- 入度(in-degree)：終點為 u 的邊數
- 出度(out-degree)：起點為 u 的邊數



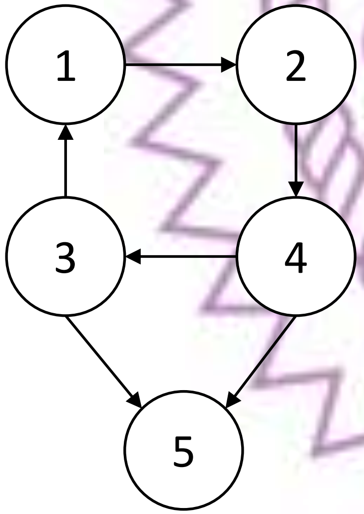
定義

- 路徑(Path)：由頭尾相連的邊組合成的集合
- 路徑長：路徑上邊的數量或邊的權重總和



定義

- 簡單路徑(Simple Path)：
  一條路徑中，起終點可以為同一個點，但其他頂點皆為不相同



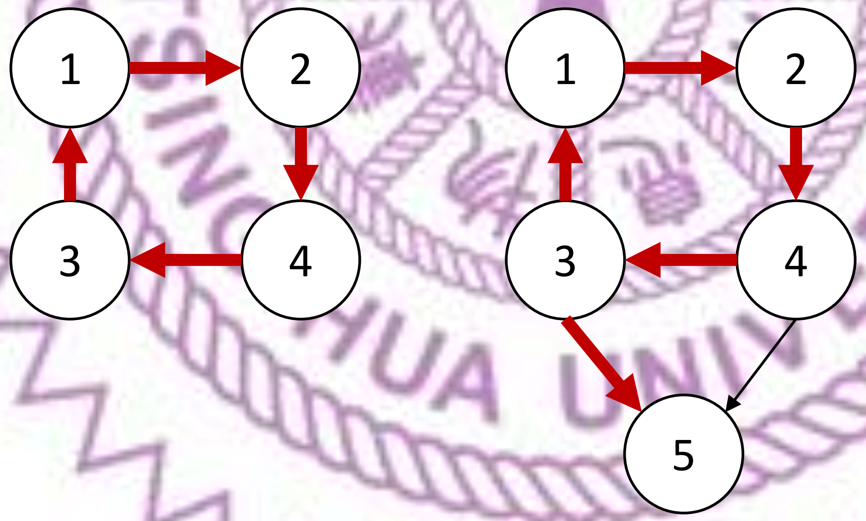
$1 \to 2 \to 4 \to 3 \to 1$ 是簡單路徑

$1 \to 2 \to 4 \to 3 \to 5$ 是簡單路徑

$1 \to 2 \to 4 \to 3 \to 1 \to 2 \to 4 \to 5$ 不是簡單路徑

定義

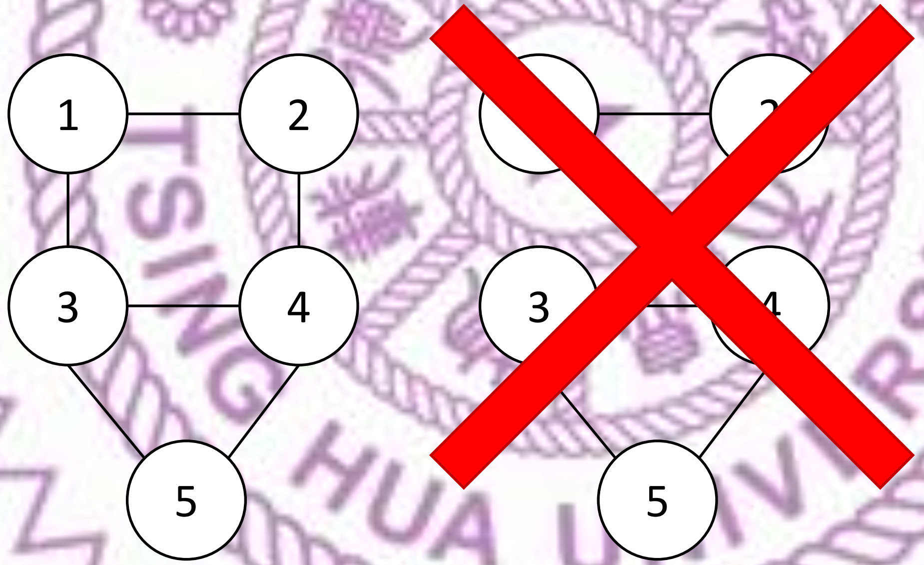
- 環(cycle)：起點和終點為同一點的路徑
- 沒特別說明的話，路徑也可以包含環



$1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5$ 包含環

定義

- 連通圖(無向圖)：圖上任相異兩點必定存在一條路徑



如何存圖

- Adjacency List
  - 每個點紀錄自己連向誰
  - 需要的空間是 $O(|V|+|E|)$
  - 支援重邊
  - 大多數圖論題目適用
- Adjacency Matrix
  - 二維陣列 G ，若存在一條邊 (u,v) 則 $G[u][v]=true$
  - 需要的空間是 $O(|V|^2)$
  - 不支援重邊
  - 由於走訪速度很慢，只有在少數演算法會被使用

無向圖的輸入 – Adjacency List

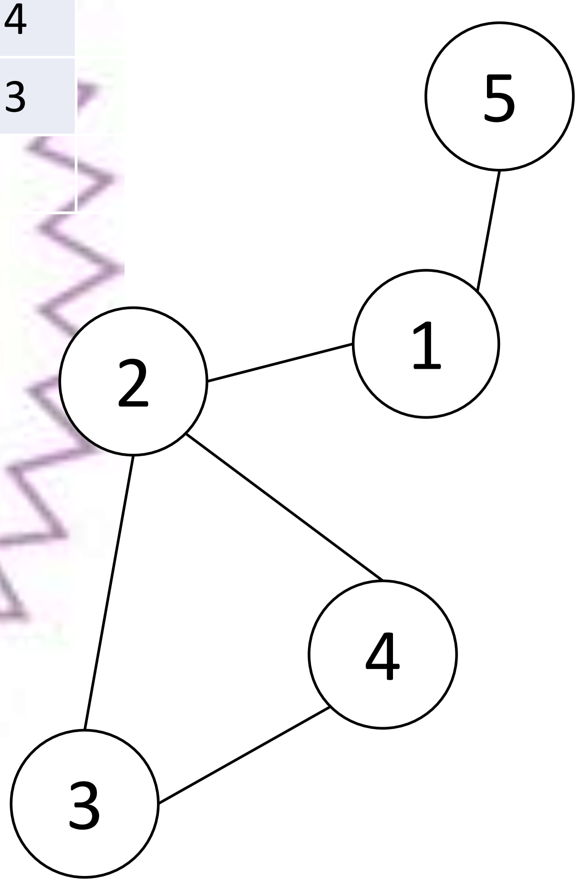
n 個點, m 條邊

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

m 條邊

```cpp
vector<vector<int>> G;
int n, m;
cin >> n >> m;
G.assign(n + 1, {});
while (m--) {
  int u, v;
  cin >> u >> v;
  G[u].emplace_back(v);
  G[v].emplace_back(u);
}
```

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



有向圖的輸入 – Adjacency List

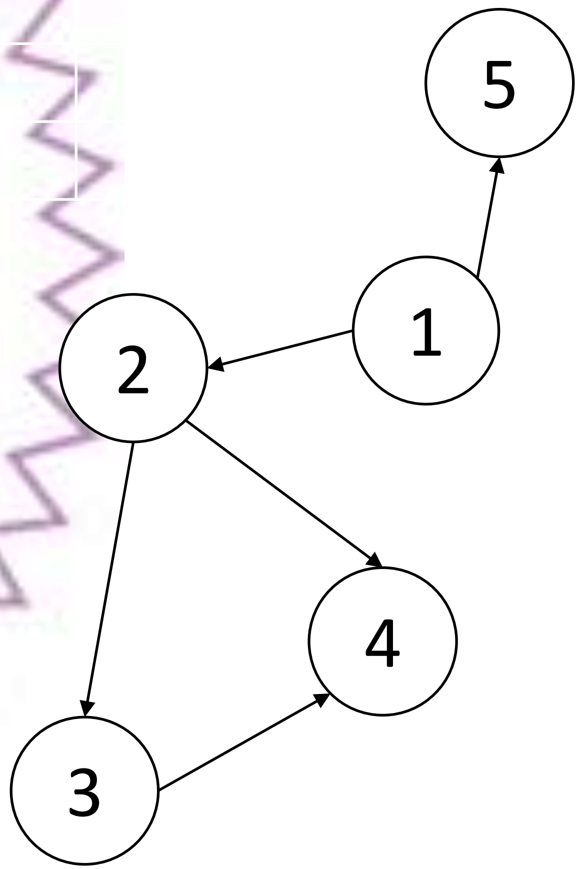
n 個點, m 條邊

m 條邊

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

```
vector<vector<int>> G;
int n, m;
cin >> n >> m;
G.assign(n + 1, {});
while (m--) {
  int u, v;
  cin >> u >> v;
  G[u].emplace_back(v);
}
```

1	2	5
2	3	4
3	4	
4		
5		



無向圖的輸入 – Adjacency Matrix

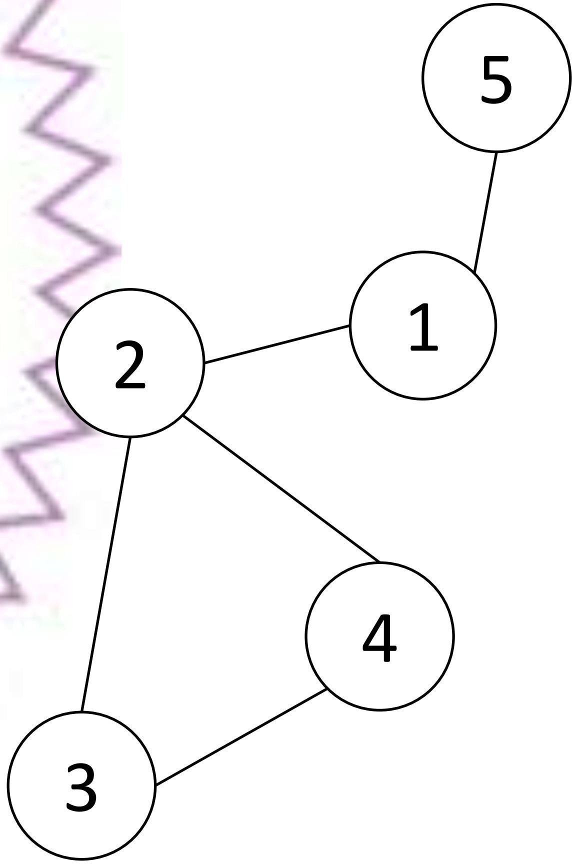
	1	2	3	4	5
1		1			1
2	1		1	1	
3		1		1	
4		1	1		
5	1				

n 個點, m 條邊

m 條邊

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

```cpp
vector<vector<int>> G;
int n, m;
cin >> n >> m;
G.assign(n + 1, vector<int>(n + 1));
while (m--) {
  int u, v;
  cin >> u >> v;
  G[u][v] = G[v][u] = 1;
}
```



有向圖的輸入 – Adjacency Matrix

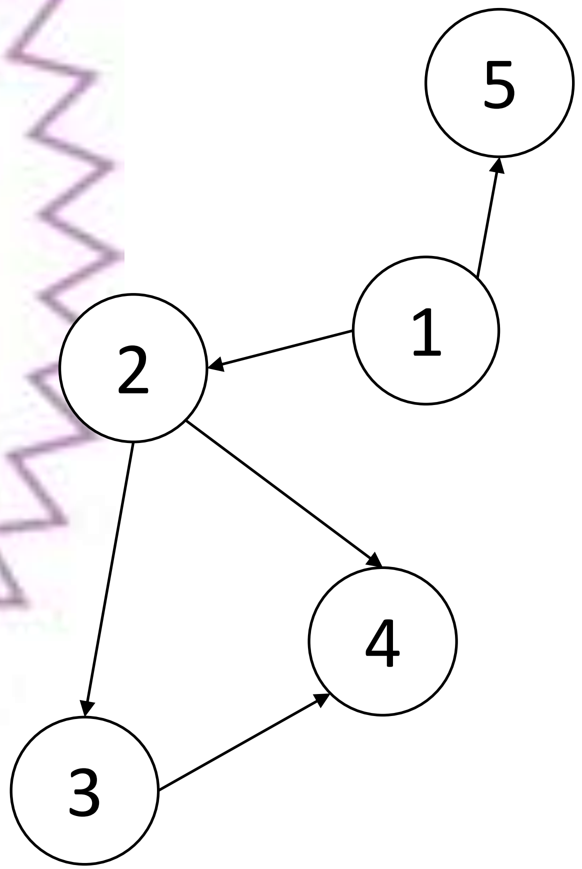
n 個點, m 條邊

m 條邊

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

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

```
vector<vector<int>> G;
int n, m;
cin >> n >> m;
G.assign(n + 1, vector<int>(n + 1));
while (m--) {
  int u, v;
  cin >> u >> v;
  G[u][v] = 1;
}
```



圖的邊數、點數

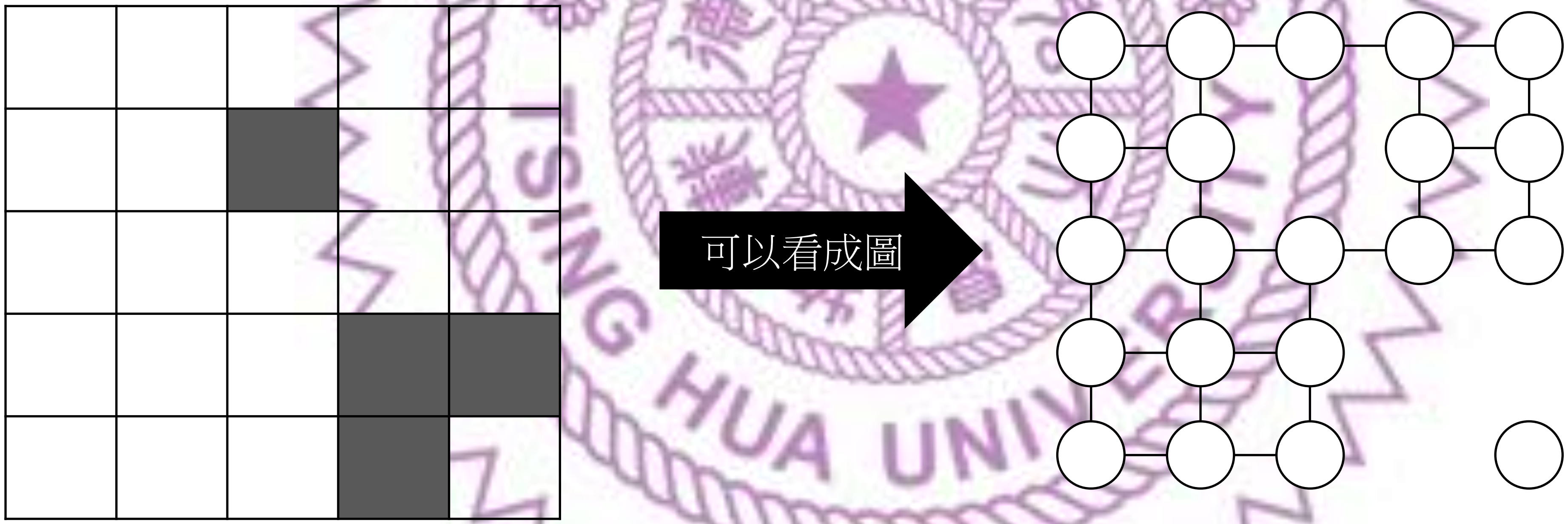
- 若沒有重邊、自環，則

$$|E| \leq \frac{|V| \times (|V| - 1)}{2}$$

- 對於 $|E| \approx |V|^2$ 的圖稱之為稠密圖(dense graph)
- 反之稱為稀疏圖(sparse graph)

圖上的 DFS、BFS

- 如同 Flood-fill ，DFS、BFS 會走過所有連通的點



圖上的 DFS (Adjacency List) $O(|V| + |E|)$

```
pair<int, int> Dxy[4] =
    {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};
void dfs(int x, int y) {
  if (grid[x][y]) return;
  grid[x][y] = true;
  for (auto [dx, dy] : Dxy)
    dfs(x + dx, y + dy);
}
```

```
vector<bool> visit;

void dfs(int u) {
  if (visit[u]) return;
  visit[u] = true;
  for (auto v : G[u]) {
    dfs(v);
  }
}
```

圖上的 BFS (Adjacency List) $O(|V| + |E|)$

```cpp
pair<int, int> Dxy[4] =
    {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};
void bfs(int x, int y) {
  queue<pair<int,int>> Q;
  Q.emplace(x, y);
  while(Q.size()) {
    tie(x, y) = Q.front();
    Q.pop();
    if(grid[x][y]) continue;
    grid[x][y] = true;
    for (auto [dx, dy] : Dxy) {
        Q.emplace(x + dx, y + dy);
    }
  }
}
```

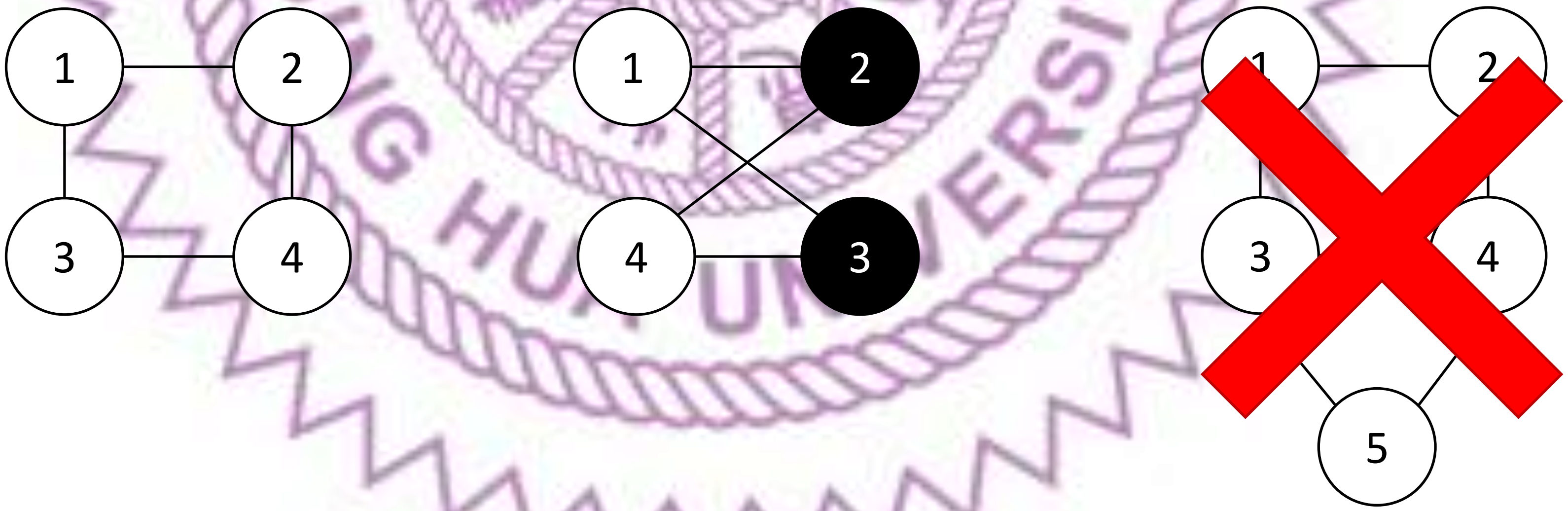
```cpp
vector<bool> visit;

void bfs(int u) {
  queue<int> Q;
  Q.emplace(u);
  while (!Q.empty()) {
    u = Q.front();
    Q.pop();
    if (visit[u]) continue;
    visit[u] = true;
    for (auto v : G[u]) {
      Q.emplace(v);
    }
  }
}
```

特殊的圖

- 樹 (Tree)
- 二分圖 (Bipartite Graph)
- 有向無環圖 (Directed Acyclic Graph, DAG)
- 平面圖 (Planar Graph)
- 弦圖 (Chordal Graph)

二分圖

- 二分圖
  - 一個無向圖的頂點可以分成兩個集合，使的同集合中的點不相鄰
- 黑白染色 (二分圖色)
  - 將圖中的點圖成黑或白，使得每條邊的兩端必不同色



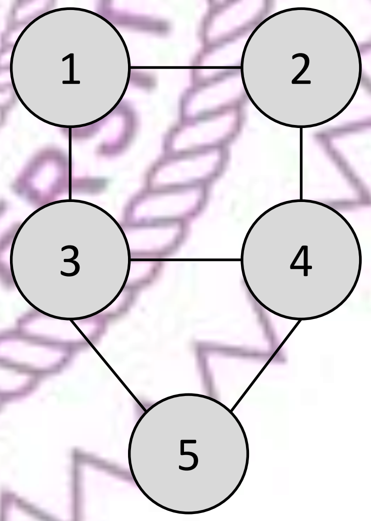
二分圖

- 性質

- G 是二分圖 $\leftrightarrow$ G 可以被二分圖色

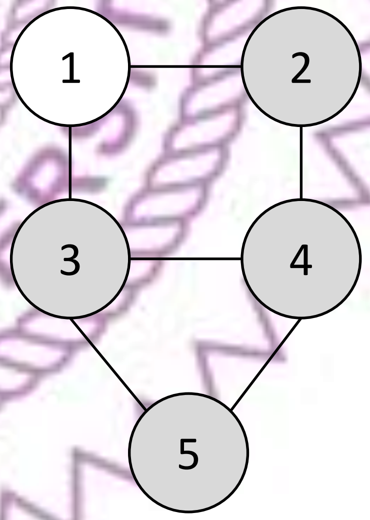
二分圖判定

- 如何判定一個無向圖是否為二分圖呢？
  - 1. 隨便找一個點，塗成黑或是白
  - 2. 從這個點 DFS or BFS，並將相鄰的點塗成相異的顏色
  - 3. 如果塗色的過程中發生矛盾，就不是二分圖，否則就是二分圖



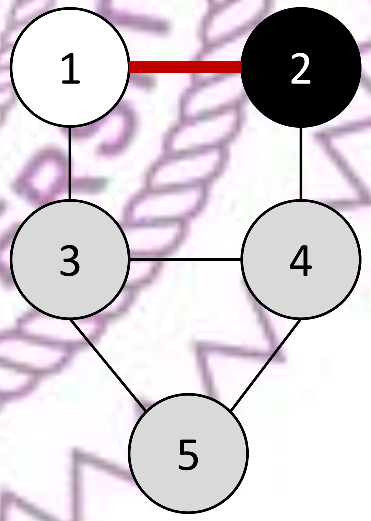
二分圖判定

- 如何判定一個無向圖是否為二分圖呢？
  - 1. 隨便找一個點，塗成黑或是白
  - 2. 從這個點 DFS or BFS，並將相鄰的點塗成相異的顏色
  - 3. 如果塗色的過程中發生矛盾，就不是二分圖，否則就是二分圖



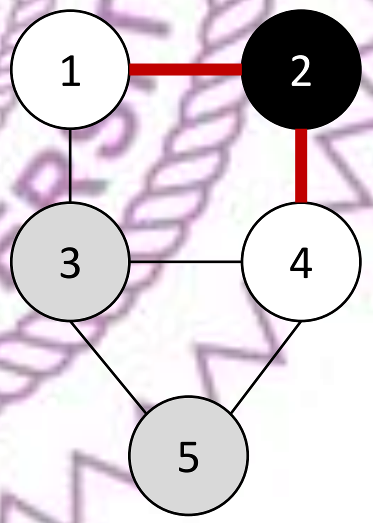
二分圖判定

- 如何判定一個無向圖是否為二分圖呢？
  - 1. 隨便找一個點，塗成黑或是白
  - 2. 從這個點 DFS or BFS，並將相鄰的點塗成相異的顏色
  - 3. 如果塗色的過程中發生矛盾，就不是二分圖，否則就是二分圖



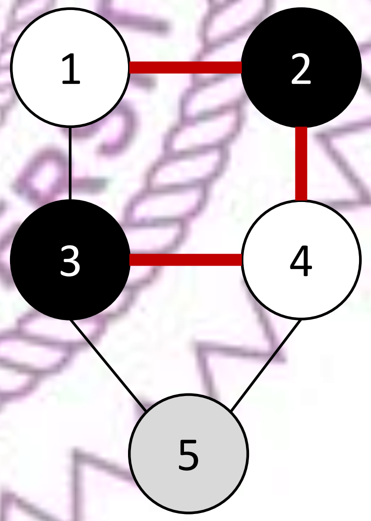
二分圖判定

- 如何判定一個無向圖是否為二分圖呢？
  - 1. 隨便找一個點，塗成黑或是白
  - 2. 從這個點 DFS or BFS，並將相鄰的點塗成相異的顏色
  - 3. 如果塗色的過程中發生矛盾，就不是二分圖，否則就是二分圖



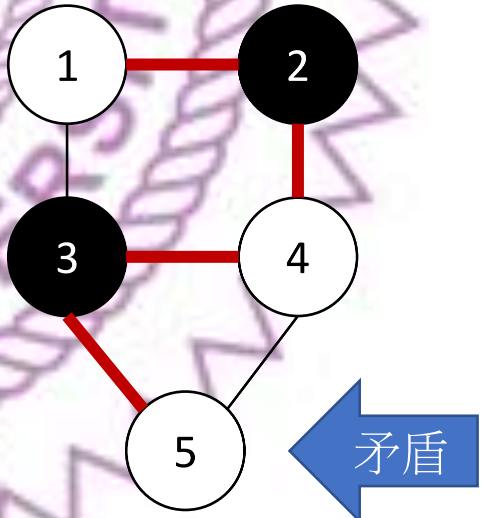
二分圖判定

- 如何判定一個無向圖是否為二分圖呢？
  - 1. 隨便找一個點，塗成黑或是白
  - 2. 從這個點 DFS or BFS，並將相鄰的點塗成相異的顏色
  - 3. 如果塗色的過程中發生矛盾，就不是二分圖，否則就是二分圖



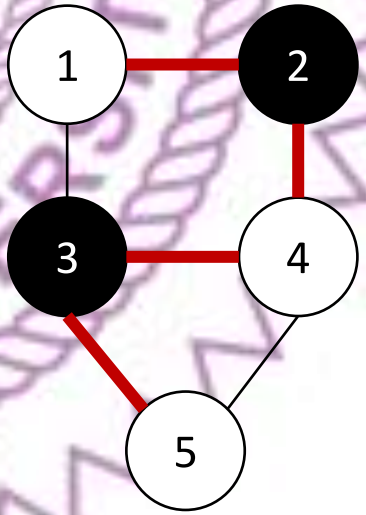
二分圖判定

- 如何判定一個無向圖是否為二分圖呢？
  - 1. 隨便找一個點，塗成黑或是白
  - 2. 從這個點 DFS or BFS，並將相鄰的點塗成相異的顏色
  - 3. 如果塗色的過程中發生矛盾，就不是二分圖，否則就是二分圖

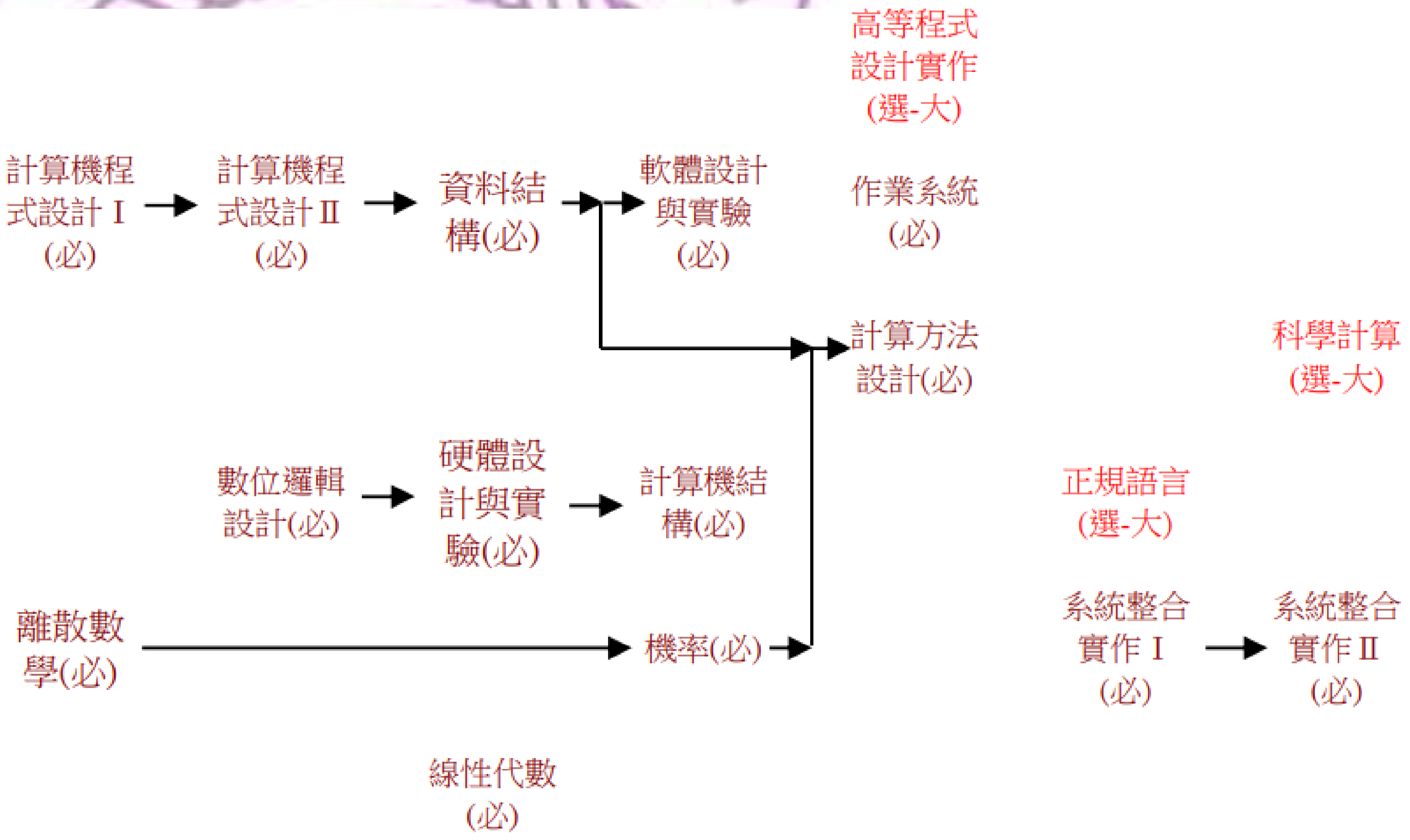


二分圖判定

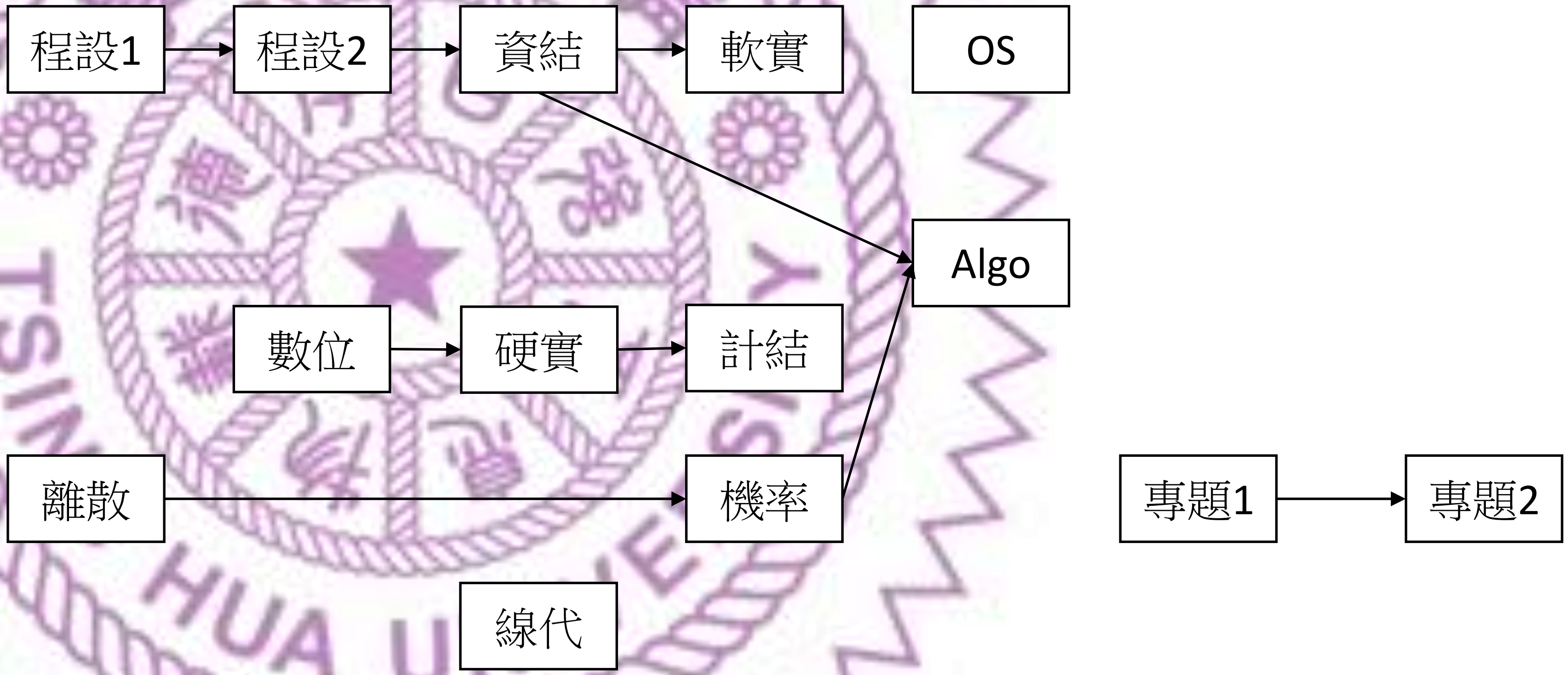
```cpp
vector<int> color; // 一開始初始化都是 0

bool dfs(int u, int c = 1) {
  if (color[u])
    return color[u] == c;
  color[u] = c;
  for (auto v : G[u])
    if (!dfs(v, c * -1))
      return false;
  return true;
}
```



清大資工課程地圖

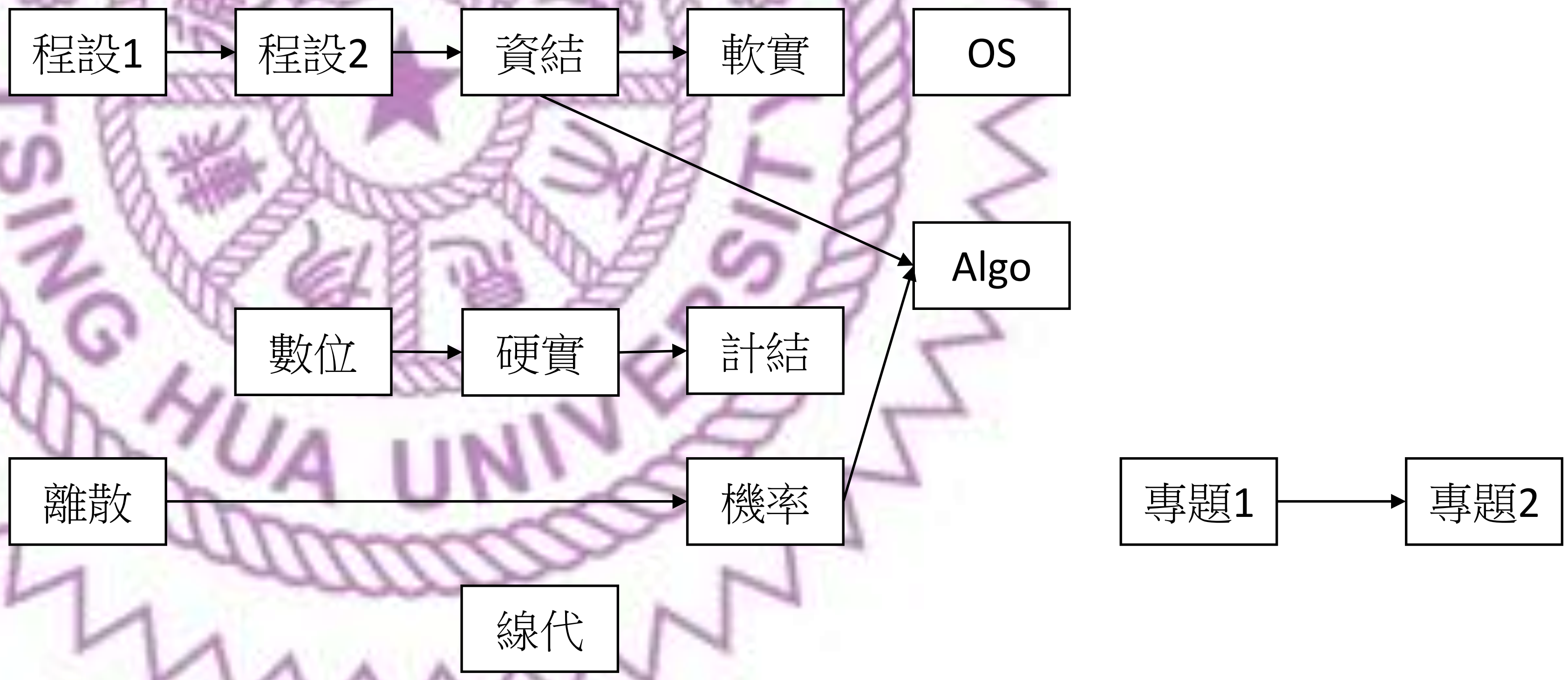

高等程式設計實作(選-大)
計算機程式設計 I (必)
計算機程式設計 II (必)
資料結構(必)
軟體設計與實驗(必)
作業系統(必)
計算方法設計(必)
科學計算(選-大)
數位邏輯設計(必)
硬體設計與實驗(必)
計算機結構(必)
正規語言(選-大)
離散數學(必)
機率(必)
系統整合實作 I (必)
系統整合實作 II (必)
線性代數(必)

變成圖論



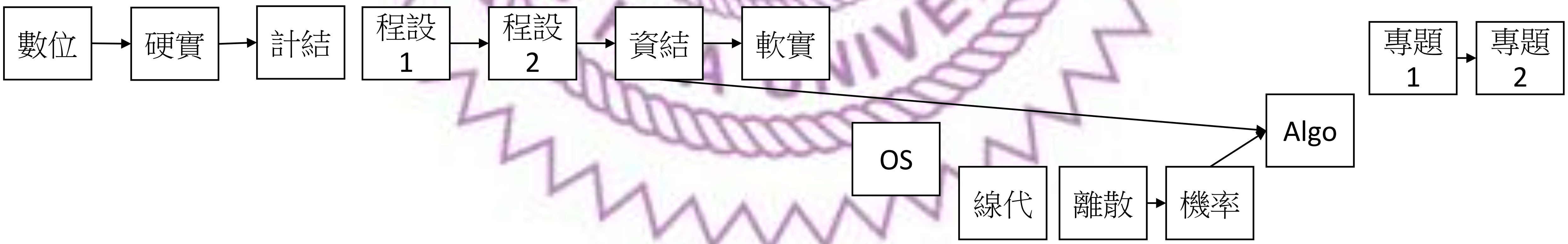
有向無環圖

顧名思義，是有向圖且沒有環



拓樸排序 Topological sort

- 假設一次只能修一堂課，必須為課程安排先後順序
- $A \rightarrow B$ 表示 A 要排在 B 前面
- 得到的順序稱為拓樸排序
  - 可能有多種排法



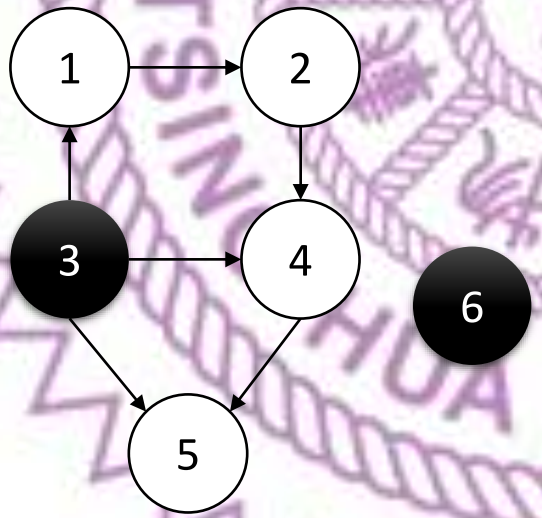
拓樸排序 Topological sort

- 性質

- G 是有向無環圖 $\leftrightarrow$ G 可以被拓樸排序

誰有資格排在第一位

- 沒有連進來的邊的點可以排在第一位
- 也就是 in-degree 為 0 的點



$3 \rightarrow 6 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5$

$6 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5$

有向圖記錄 in-degree

1	2	3	4	5
0	1	1	2	1

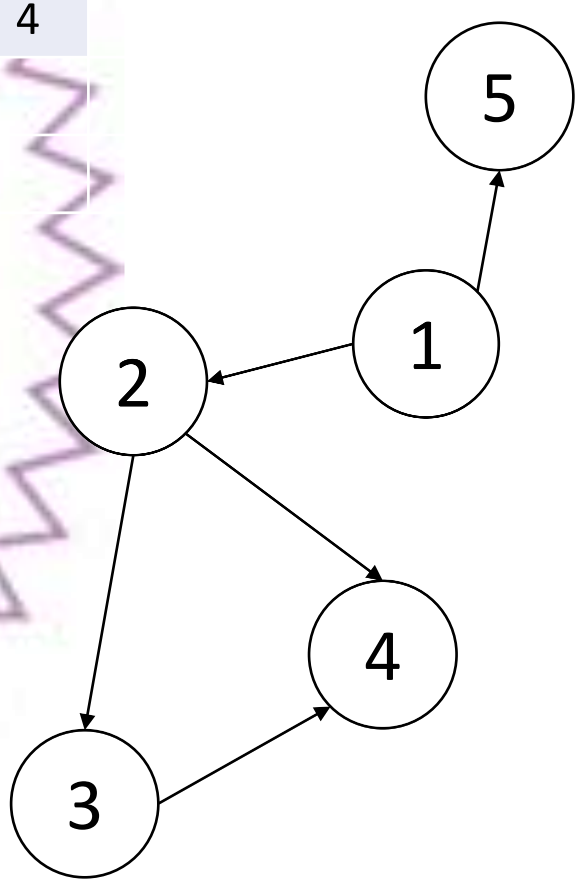
1	2	5
2	3	4
3	4	
4		
5		

n 個點, m 條邊

m 條邊

```
5 6
1 2
2 3
2 4
1 5
3 4
```

```cpp
vector<vector<int>> G;
vector<int> in;
int n, m;
cin >> n >> m;
G.assign(n + 1, {});
in.assign(n + 1, 0);
while (m--) {
  int u, v;
  cin >> u >> v;
  G[u].emplace_back(v);
  ++in[v];
}
```

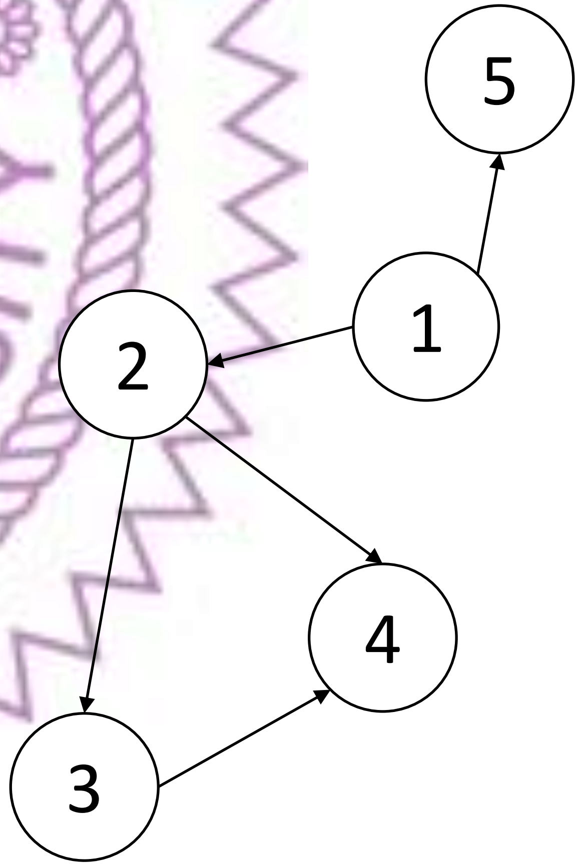


Kahn 演算法

Queue

- 不斷找出 in-degree 是 0 的點並刪掉
- 刪除的順序就是拓樸排序
- 可以用 queue、priority_queue 等容器維護刪除的順序

1	2	3	4	5
0	1	1	2	1

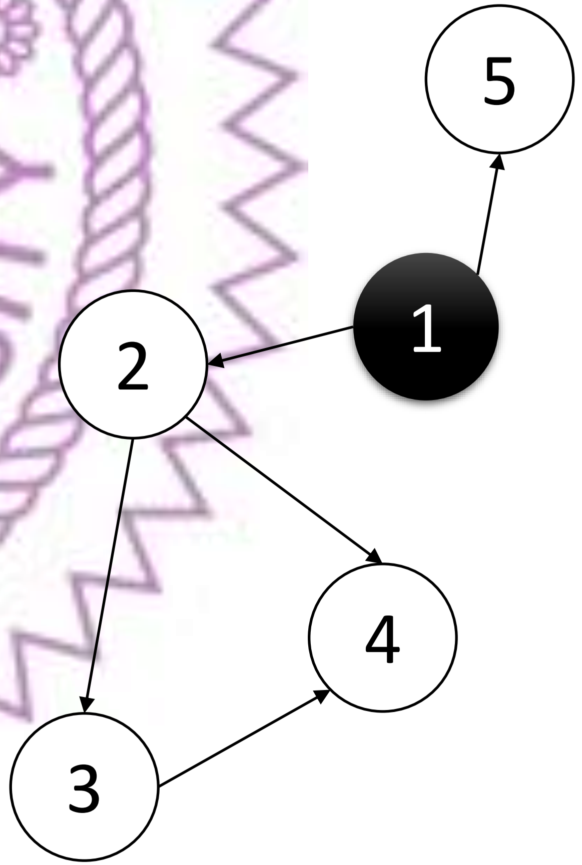


Kahn 演算法

Queue 1

- 不斷找出 in-degree 是 0 的點並刪掉
- 刪除的順序就是拓樸排序
- 可以用 queue、priority_queue 等容器維護刪除的順序

1	2	3	4	5
0	1	1	2	1



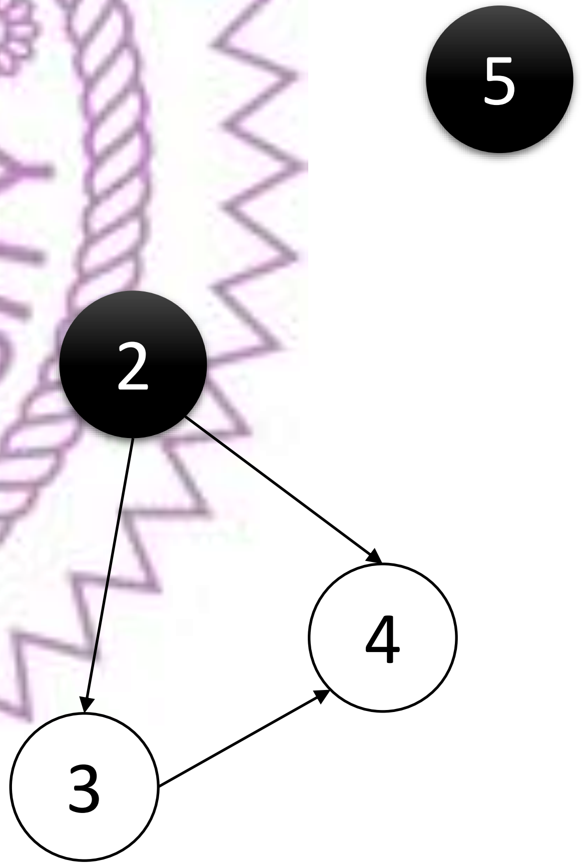
Kahn 演算法

Queue

1	2	5

- 不斷找出 in-degree 是 0 的點並刪掉
- 刪除的順序就是拓樸排序
- 可以用 queue、priority_queue 等容器維護刪除的順序

1	2	3	4	5
0	0	1	2	0



Kahn 演算法

- 不斷找出 in-degree 是 0 的點並刪掉
- 刪除的順序就是拓樸排序
- 可以用 queue、priority_queue 等容器維護刪除的順序

Queue				
	1	2	5	3

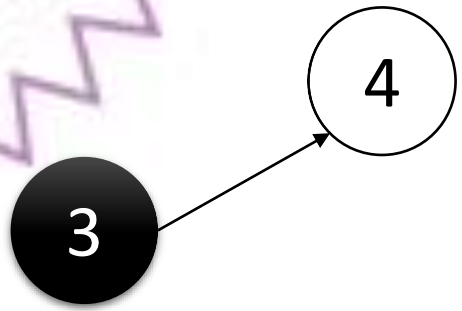
1	2	3	4	5
0	0	0	1	0

Kahn 演算法

- 不斷找出 in-degree 是 0 的點並刪掉
- 刪除的順序就是拓樸排序
- 可以用 queue、priority_queue 等容器維護刪除的順序

Queue				
	1	2	5	3

1	2	3	4	5
0	0	0	1	0



Kahn 演算法

Queue

1	2	5	3	4

- 不斷找出 in-degree 是 0 的點並刪掉
- 刪除的順序就是拓樸排序
- 可以用 queue、priority_queue 等容器維護刪除的順序

1	2	3	4	5
0	0	0	0	0

4

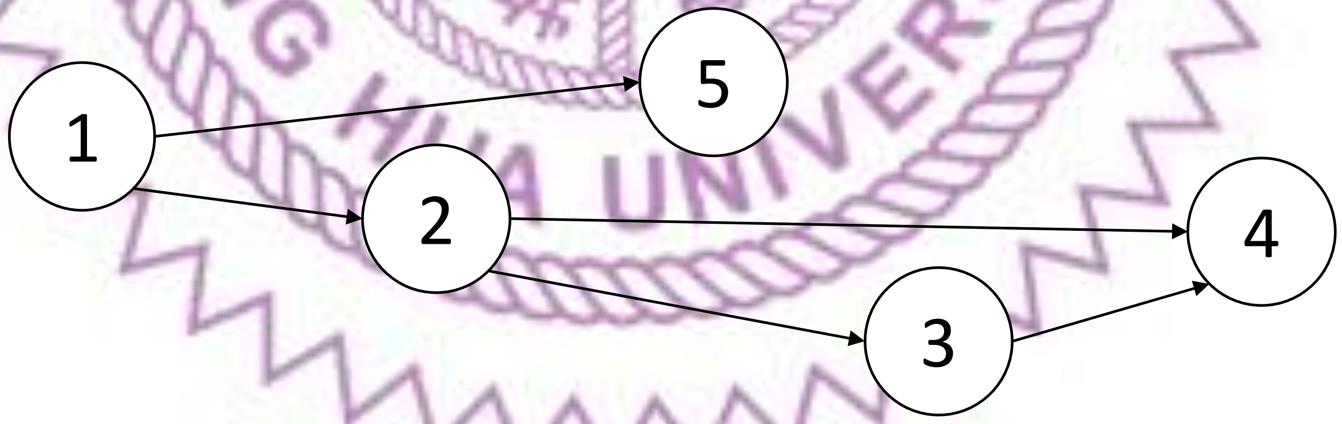
Kahn 演算法

Queue

1	2	5	3	4

1	2	3	4	5
0	0	0	0	0

- 不斷找出 in-degree 是 0 的點並刪掉
- 刪除的順序就是拓樸排序
- 可以用 queue、priority_queue 等容器維護刪除的順序



Kahn 演算法 + 判斷是否是 DAG

```cpp
vector<int> ans;
bool toposort(int n) {
  ans.clear();
  queue<int> Q;
  for (int u = 1; u <= n; ++u)
    if (in[u] == 0) Q.emplace(u);
  while (Q.size()) {
    int u = Q.front();
    Q.pop();
    ans.emplace_back(u);
    for (auto v : G[u])
      if (--in[v] == 0) Q.emplace(v);
  }
  return ans.size() == n;
}
```

$$O(|V| + |E|)$$

實際上 ans 就有 Q 的所有資訊

```
vector<int> ans;
bool toposort(int n) {
  ans.clear();
  for (int u = 1; u <= n; ++u)
    if (in[u] == 0) ans.emplace_back(u);
  for (size_t i = 0; i < ans.size(); ++i) {
    int u = ans[i];
    for (auto v : G[u])
      if (--in[v] == 0) ans.emplace_back(v);
  }
  return ans.size() == n;
}
```

另一種 DFS 的方法 (不用紀錄 in-degree)

```
vector<int> visit;
vector<int> ans;

bool dfs(int u) {
  visit[u] = -1;
  for (int v : G[u]) {
    if (visit[v] < 0) return false;
    else if (!visit[v])
      if (!dfs(v)) return false;
  }
  visit[u] = 1;
  ans.emplace_back(u);
  return true;
}
```

```
bool toposort(int n) {
  ans.clear();
  visit.assign(n + 1, 0);
  for (int u = 1; u <= n; ++u)
    if (!visit[u])
      if (!dfs(u)) return false;
  reverse(ans.begin(), ans.end());
  return true;
}
```