Ch11. 深度優先搜尋 Depth-First Search

本章首先，一樣會先簡介並且實作經典的深度優先搜尋演算法 DFS。

再來，會介紹一些深度優先搜尋的常見應用：

A. 環的確認

B. 二分圖的判別

C. 拓樸排序

D. 尋找強連通元件

E. 八皇后問題

最後，來做幾題實戰練習。

1. 深度優先搜尋簡介與實作

在進入深度優先搜尋的講解之前，先複習一下資料結構中的「堆疊」。

（1）堆疊 Stack

使用堆疊這種資料結構時，新增資料和刪除資料在「同側」。



像一個書堆一樣，增加一本書時，是把書放在「最上面」，拿掉一本書時，也是從「最上面」取走。

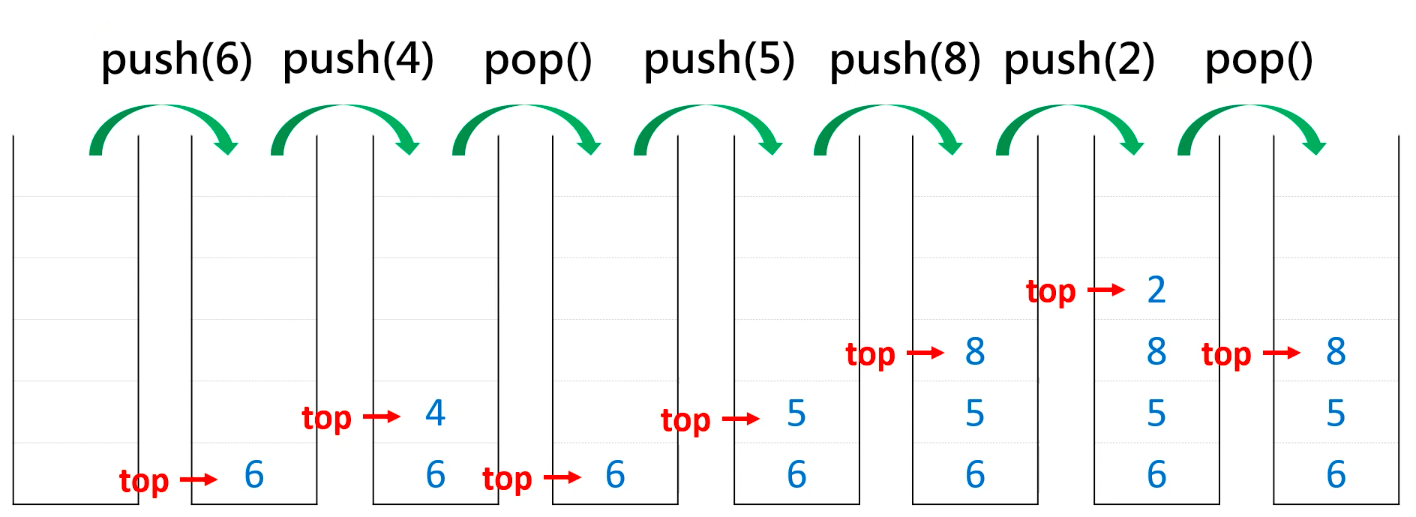
另外一個例子是吃洋芋片，每次都把洋芋片從圓筒的上方拿出來，吃不完要放回去時，也是從上面放。

洗衣服也類似，把衣服洗完、堆好後，每次拿一件衣服出來穿是從最上面拿，髒衣服洗完後，也是繼續往上堆，這種配置會導致總是在穿衣服堆最上面的那幾件，這也是堆疊的一種特色。

這種操作又叫做 last-in-first-out（LIFO），最後放進去的會最先出來，因為最後放進去的在「最上方」，要拿取資料時同樣也是從「最上方」。

|  |  |
| --- | --- |
| 常見的堆疊操作 | |
| push  pop  top  empty  size | 新增一筆資料  刪除一筆資料  回傳最末端（上面）的資料  確認 stack 裡是否有資料  回傳 stack 內的資料個數 |

實際來看一個例子，表示堆疊時，通常會畫成下圖中像品客洋芋片的圓筒形狀，上面是開口，下面則是封閉的，push 時會從上面放入資料，pop 時也是從上面取出。



假設一開始執行 push(6)，代表新增一筆資料 6 到這個堆疊裡面，堆疊中多出 6 這筆資料。再來，執行 push(4)，把 4 這筆資料放進堆疊裡，這樣 4 就在 6 的上面。pop() 是刪除一筆資料，要刪除時一樣從上面拿走，所以會把最上面的 4 拿掉，剩下 6。接下來是 push(5)，把 5 放到堆疊裡，變成 5 和 6，push(8) 和 push(2) 依序把 8 和 2 放到堆疊裡，這時 pop() 把最上面的 2 取出來，堆疊從上而下剩下 8 5 6 三筆資料。

（2）堆疊的用途

堆疊最常見的用途：依序紀錄先前的資訊 A. 常用來回復到先前的狀態 a. 瀏覽器回到上一頁 b. 編輯器復原：word 等文書軟體中的「復原」 c. 編譯器的解析 parse d. 函式呼叫（遞迴） B. 迷宮探索、河內塔、發牌：深度優先搜尋 DFS



在文書軟體中常會做一連串操作，如依序執行剪下、貼上、斷行、輸入、調整字體、貼上、刪除。完成這些操作後，若想要復原應如何進行呢？需要從最後執行的「刪除」開始，依序檢視做過哪些動作，並進行反向操作。

對於堆疊而言，最後放入的資料在最上面，與復原時「最後做的操作應最先復原」的特性相同，因此堆疊特別適合用來實現這類工作。

另外，每次瀏覽一個網頁，或者遞迴呼叫函式的時候，也都是把這些操作放到一個堆疊 Stack 裡面，方便後續處理。深度優先搜尋使用的也是堆疊 Stack；相對的，上一章介紹的廣度優先搜尋則是使用佇列 Queue。

Stack 和 Queue 有一個共通特性：無法直接存取位於中間的資料，只能用 pop 函式一個一個依序把資料取出來。

（3）深度優先搜尋 DFS

深度優先搜尋同樣是一種尋訪所有頂點的行為準則，在 DFS 中，先不斷尋訪頂點的「單一」相鄰頂點，移動到該相鄰頂點後，直接再繼續選擇該相鄰頂點的「一個」相鄰頂點繼續移動，直到無法尋訪時，返回之前頂點，並選擇「其他可供選擇的相鄰頂點」繼續進行。

用深度優先搜尋走迷宮時，每次遇到岔路，先選其中一條往下走，從這一條路往下又遇到岔路時，繼續往下走，一直到遇到死胡同為止，才退回之前的路口，選擇還沒走過的路線。

再以找工作為例，要找到自己真的想做的工作，就先從所有工作中選一個，一直做到再也受不了為止，才去試其他工作。不同於廣度優先搜尋是把老師、藥師、...所有的工作都先做個一兩天，全部試過一點之後再做下一輪，深度優先搜尋是把一個工作做到完全確定不可行，才去換下一個工作。通常 DFS 會搭配堆疊來完成，因為走迷宮時一條路走到底，確定不能走之後，要「退回先前的路口」，這種「回復之前的狀態」的工作正適合使用堆疊。

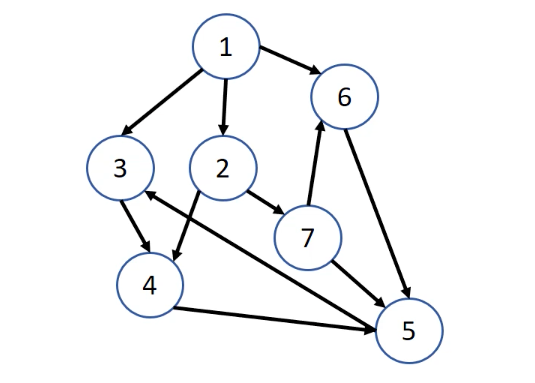
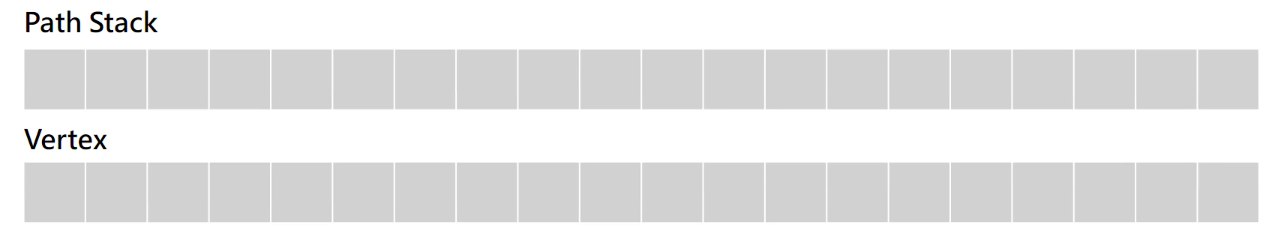
為了避免陷入無窮迴圈，同樣要把頂點區分為三個顏色：

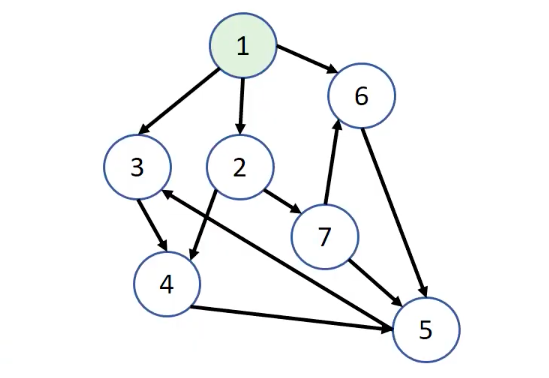
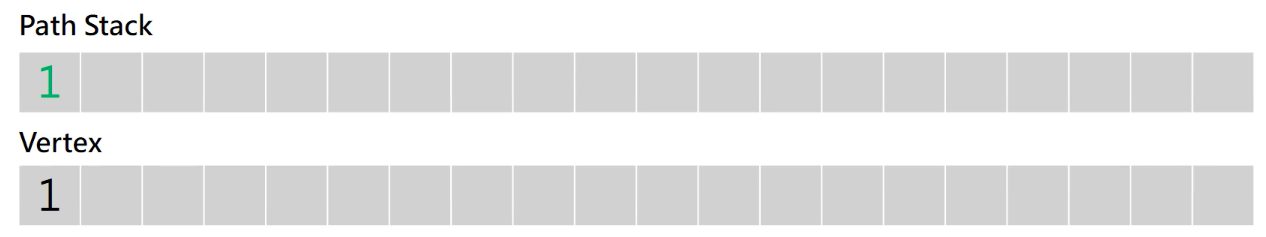
A. 白色 White：尚未尋訪過

B. 灰色 Gray：已尋訪過，尚未處理

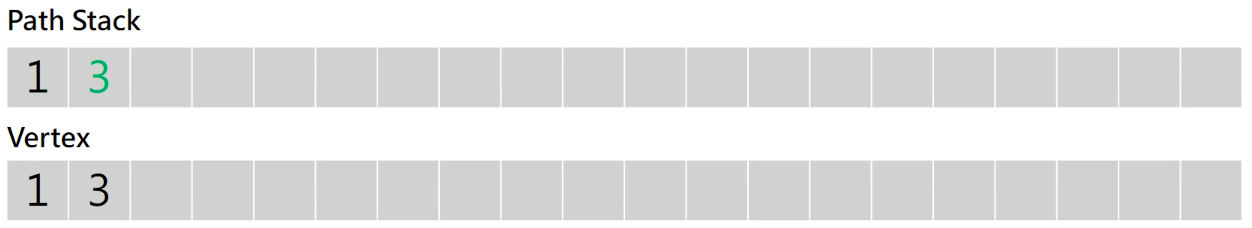
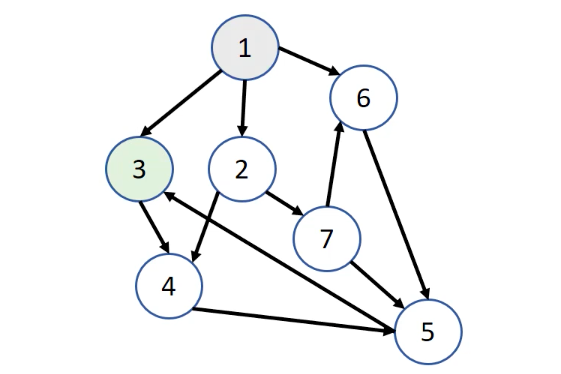
C. 黑色 Black：已尋訪過、已處理

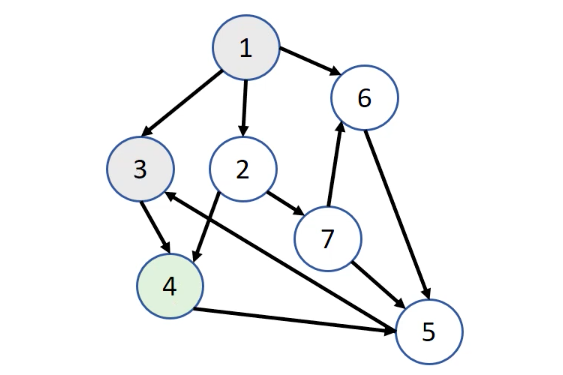
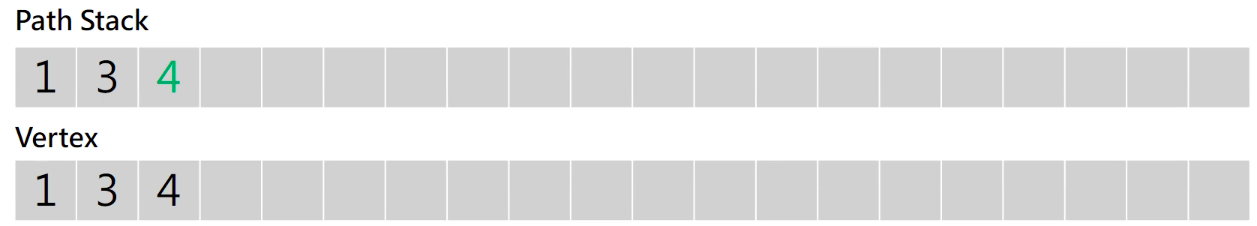
進行 DFS 時，若有終點，進行到終點時就可以停止，若沒有終點，則待尋訪過所有頂點後才停止。

（4）深度優先搜尋的進行過程

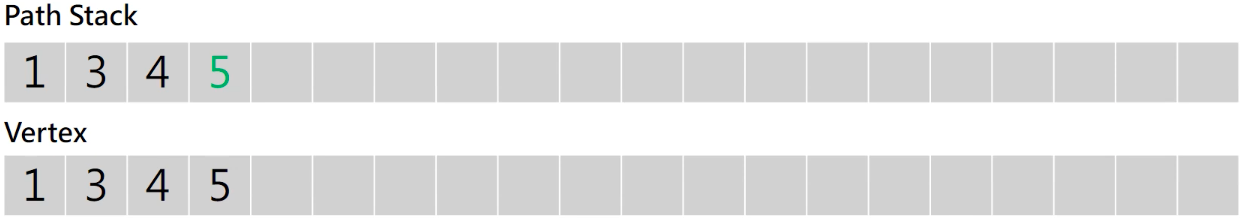
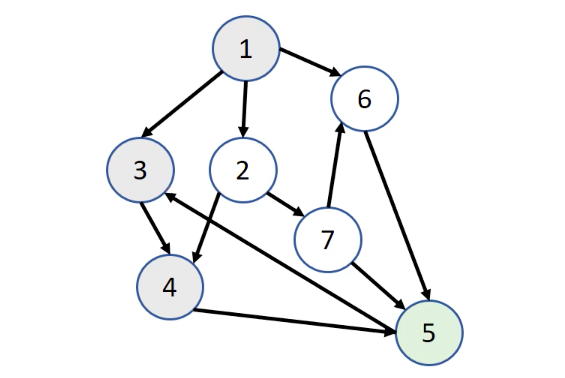
首先準備一個堆疊 Path 記錄路徑，一個向量 Vertex 用來記錄走過的頂點。

一開始任意選擇一個頂點 1 出發，先把頂點 1 放入 Path 與 Vertex 中，且把頂點 1 塗成灰色。

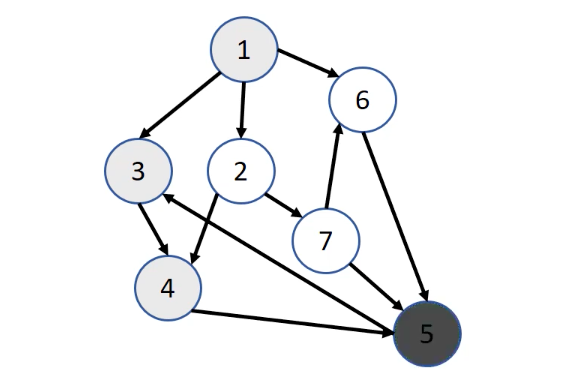
 

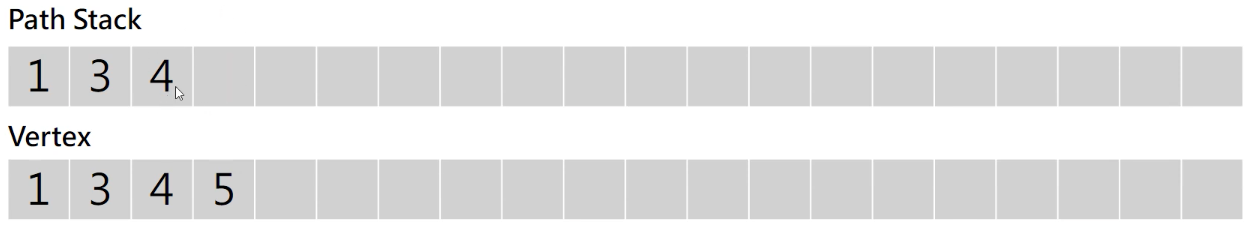
從頂點 1 有三條路可以向外走，像一個岔路一樣，分別連到頂點 3、2、6，假設選擇往頂點 3 移動，頂點 3 同樣被放到 Path 與 Vertex 中，且被塗成灰色。

從頂點 3 出發，只能往 4 移動，因此緊接著把 4 放入 Path 與 Vertex 中，並塗成灰色。

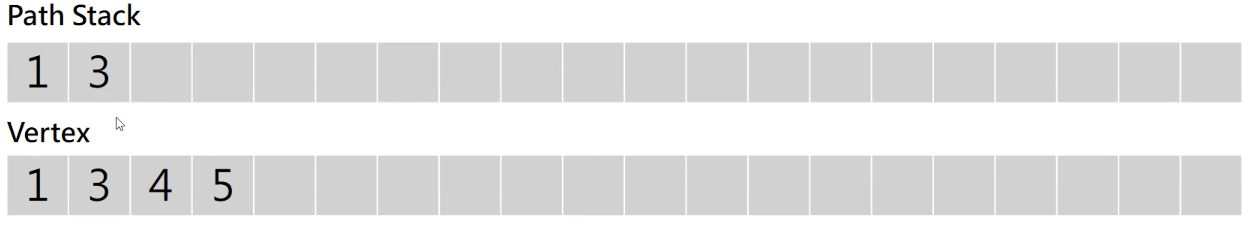
頂點 4 只有一條出邊指向 5，因此往 5 移動，把 5 放入 Path 與 Vertex 中，並塗成灰色。

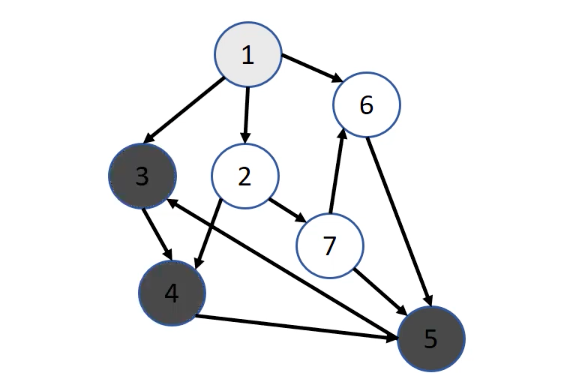
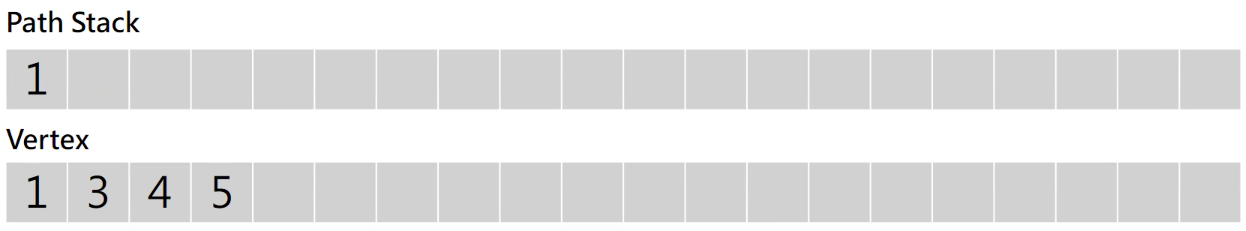
從 5 出發，有一條出邊指向頂點 3，但是頂點 3 此時是灰色，代表已經造訪過，所以代表已經沒有可以移動的方向，需要退回上一個路口。要往回退時，因為 Path 這個堆疊中記錄了剛才走過的路徑，所以按照裡面的順序就可以順利按原路退回。



從堆疊 Path 中取出一筆資料，會取到最後放入的 5，代表要退回頂點 5 之前，把頂點 5 塗成「黑色」，代表已經處理完成。

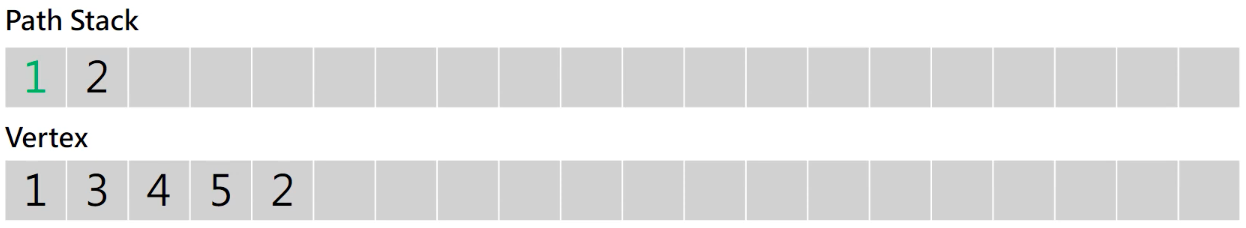
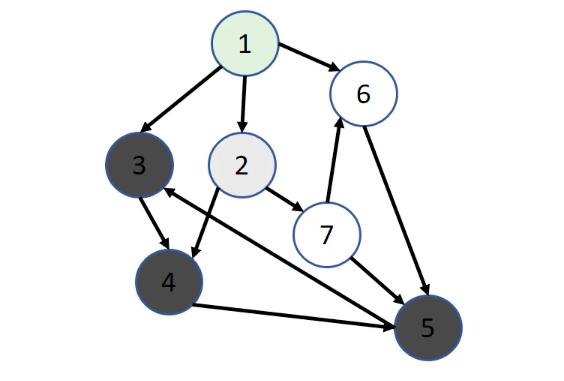


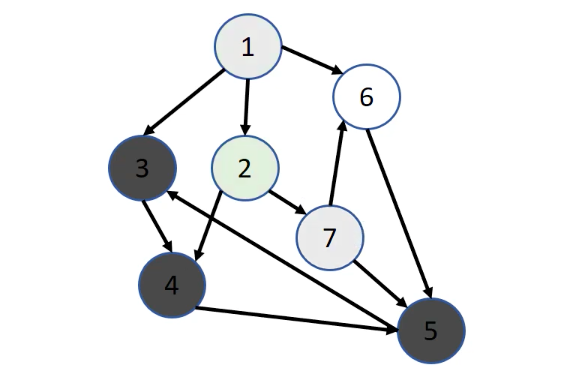
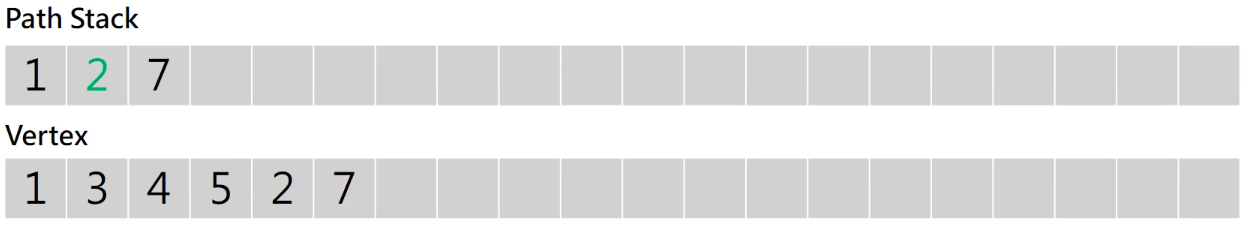


退回頂點 4 後，因為頂點 4 沒有指向還是白色的頂點，因此還要再往回退，並且把頂點 4 塗成黑色。

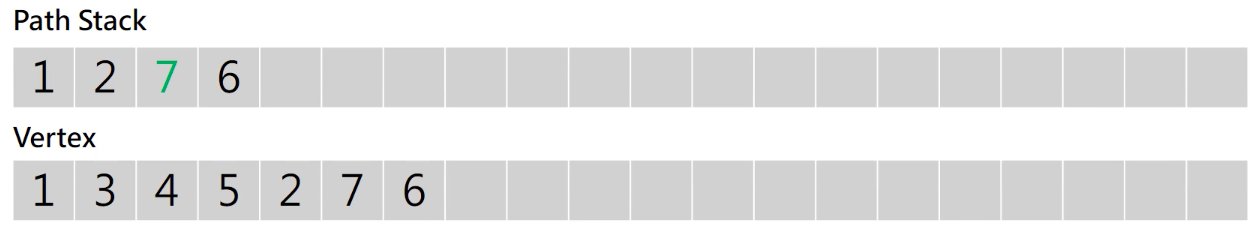
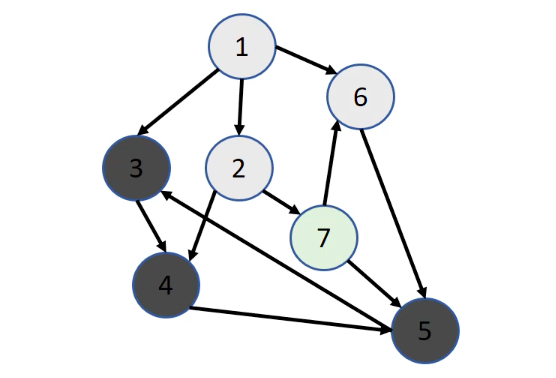
退回頂點 3 時，同樣沒有路可以向下走，再回退。

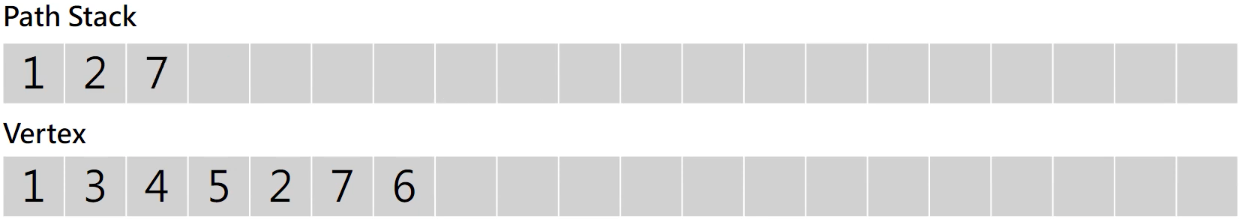
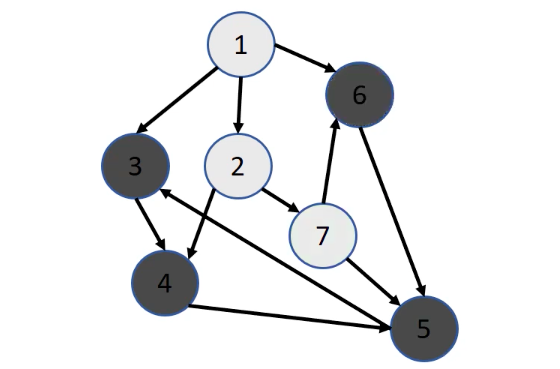
回退的過程中，就是從堆疊中取出之前經過的路徑，依照取出的頂點倒著走回去，一直到有可以選擇其他路徑的頂點時，才再往下進行。

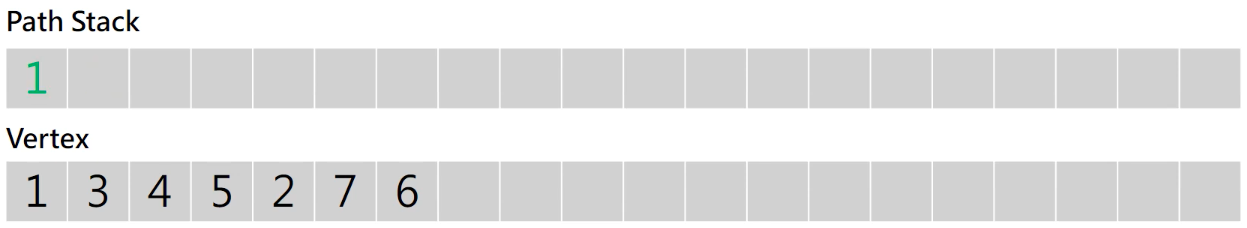
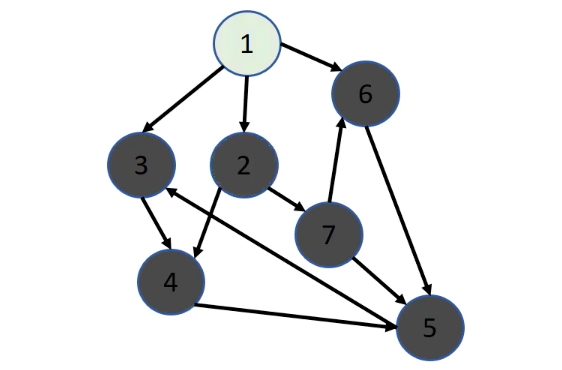
回退到頂點 1 時，有其他路徑可以走，因為頂點 1 另外還指到兩個白色頂點 2 和 6。任意選擇 2 往下走，把 2 放入 Path 和 Vertex 中，並把 2 塗成灰色。

到了頂點 2 後，因為頂點 4 已經被塗成灰色，此時唯一指到的白色頂點是頂點 7，所以往頂點 7 移動。把 7 放入 Path 和 Vertex 中，並把 7 塗成灰色。

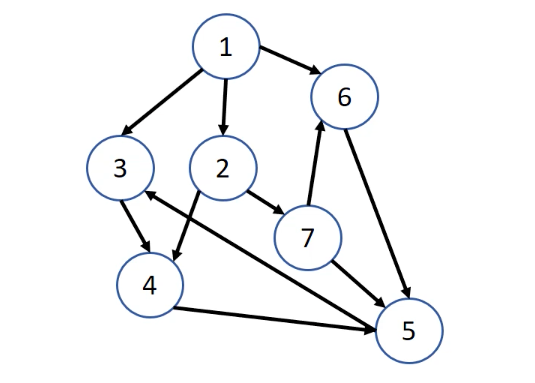
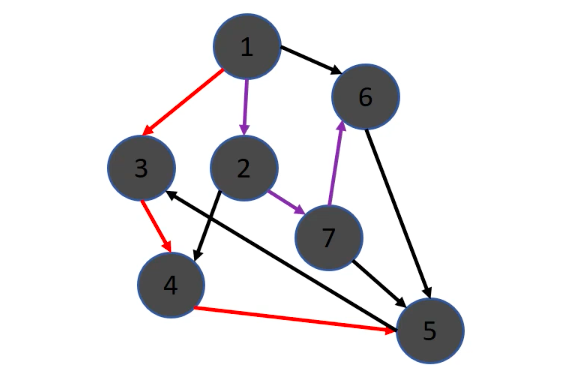
 

接著，頂點 7 可以再往頂點 6 移動，把 6 放入 Path 和 Vertex 中，並把 6 塗成灰色。

頂點 6 只指到頂點 5，但頂點 5 已經處理完成，被塗成黑色，因此要回退，把頂點 6 塗成黑色。

接著，頂點 7 也沒有其他可走的出邊，同樣塗成黑色。隨後頂點 2 也被塗成黑色。

最後，頂點 1 被塗成黑色，整個搜尋完成。

重新檢視一下剛才的流程：首先，從頂點 1 出發，走了 ，到了頂點 5 時，因為無法再往下走，所以一路回退，直到有其他選擇的 1。

回退到頂點 1 後，選擇往頂點 2 移動，經過路徑 ，到了頂點 6 時，又無法再往下走，因此回退到 1，發現所有頂點都已經被造訪過。

（5）深度優先搜尋的複雜度

|  |  |
| --- | --- |
| 深度優先搜尋的虛擬碼 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | DFS(G,s)  // O(|V|)  for each vertex(v) in G:  color[v] = white  path\_stack = {s}  DFS\_visit(s)  // O(|E|)  DFS\_visit(vertex)  color[vertex] = gray  for each vertex(v) in vertex.adjacent():  if color[v]==white  DFS\_visit(v)  color[vertex] = black |

效能分析

A. 初始化：

B. 處理所有邊（一個「相鄰」關係對應圖上的一條邊）：

C. 總和：

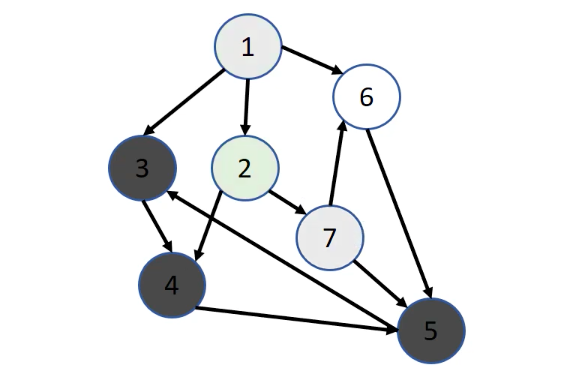
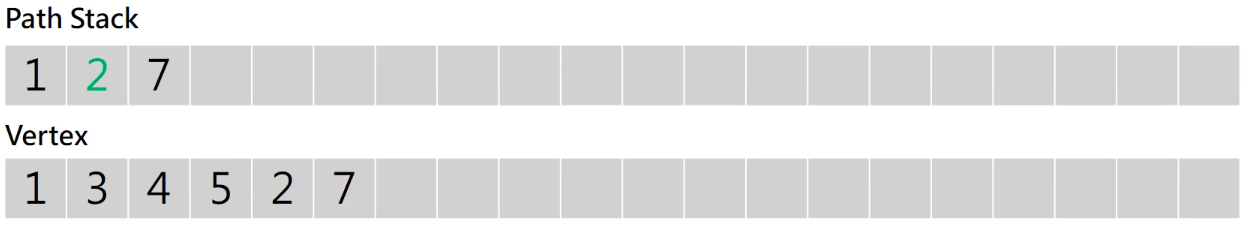
首先，把所有頂點塗成白色的複雜度是頂點數，為 。接著，用一個遞迴式來實現堆疊的效果，當中 for 迴圈試著往「相鄰頂點」移動的次數與邊數相同，為 。

因此，整個過程的複雜度總和為 ，與廣度優先搜尋相同。

（6）判斷環的存在

在 DFS 過程中，可以透過特定情形是否發生來判斷圖中是否有環：

將頂點放入 Stack 時，若發現該頂點已在 Stack 中，則有環；從頂點的顏色來看，檢查某一頂點的相鄰頂點時，若該相鄰頂點已經是灰色頂點，則有環。



DFS 的過程中，頂點的三種顏色也可以用 Stack 的角度來檢視：比如上圖中，白色代表還沒有被放到 Stack 中過，灰色代表目前正在 Stack 中，黑色代表先前被放入 Stack 中，但是已經被移除了。

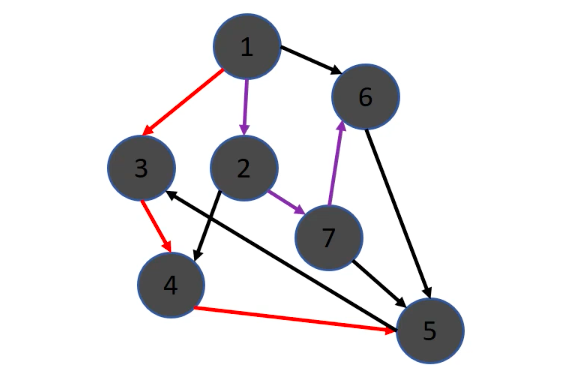
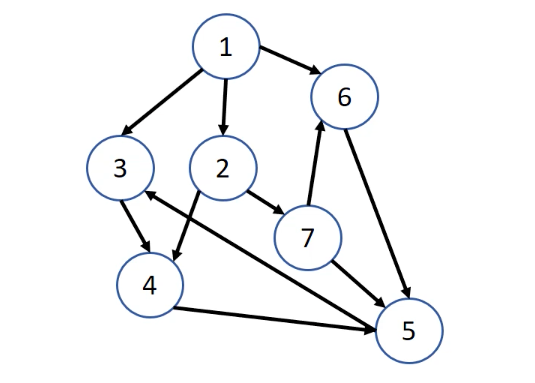
對於無向圖來說，如果往一個相鄰頂點移動，要把頂點放入 Stack 時，發現該頂點正在 Stack 中，代表沿著目前的路徑走一走，又走到了路徑中間的地方，繞成了一個圈。由該灰色頂點按照 Path 裡面的順序，又可以回到同一個頂點，因此圖中有環。有向圖也一樣，如果檢查相鄰頂點時，發現相鄰頂點已經是灰色，就代表圖中有環。

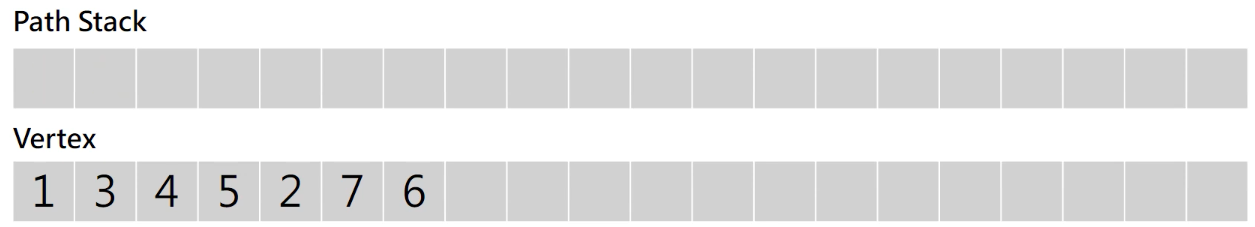
用深度優先搜尋檢查圖中有沒有環，比用廣度優先搜尋時簡單許多，因此通常要確認環的存在，會優先使用 DFS。

（7）實作深度優先搜尋

同樣擴寫鄰接列表的程式碼：

|  |  |
| --- | --- |
| 深度優先搜尋 DFS | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97 | #include <iostream>  #include <stdlib.h>  #include <vector>  #include <list>  using namespace std;  class Graph{  private:  int vertex;  vector<list<int>> edges;  // 記錄每個點的顏色，在建構式中初始化  vector<int> color;  // 尋訪的遞迴式，會在 DFS 中呼叫  void DFS\_visit(int);  public:  Graph(int);  void print\_edges();  bool add\_edge(int, int);  // 深度優先搜尋，指定起點 int  void DFS(int);  };  // 建構式  Graph::Graph(int v){  vertex = v;  edges.resize(vertex);  color.resize(vertex);  }  void Graph::print\_edge(){  for (int i=0 ; i<vertex ; i++){  cout << i+1 << "\t";  auto iter = edges[i].begin();  for ( ; iter!=edges[i].end() ; iter++){  cout << "->" << (\*iter)+1;  }  cout << endl;  }  }  bool Graph::add\_edge(int from, int to){  edges[from-1].push\_back(to-1);  }  void Graph::DFS(int start){  // 把所有頂點塗成白色  fill (color.begin(), color.end(), 0);  // 從起點開始尋訪，記得調整成索引值  DFS\_visit(start-1);  } // end of DFS  void Graph::DFS\_visit(int vertex){  // 印出過程  cout << vertex+1 << "->";  // 把傳入的頂點塗成灰色  color[vertex] = 1;  // 相鄰頂點在 edges[vertex] 中  for (auto iter=edges[vertex].begin() ; iter!=edges[vertex].end() ; iter++){  // 現在檢查的相鄰頂點  int current\_vertex = \*iter;  // 檢查是否為白色頂點  if (color[current\_vertex]==0){  // 可以從該相鄰頂點繼續往下移動  DFS\_visit(current\_vertex);  }  }  // 目前頂點的所有相鄰頂點都走過了  // 沒有可以往下走的路徑，因此要塗成黑色並回退  color[vertex] = 2;  } // end of DFS\_visit  int main(){  Graph g(7);  g.add\_edge(1,3);  g.add\_edge(1,2);  g.add\_edge(1,6);  g.add\_edge(2,4);  g.add\_edge(2,7);  g.add\_edge(3,4);  g.add\_edge(4,5);  g.add\_edge(6,5);  g.add\_edge(7,6);  g.add\_edge(7,5);  g.DFS(1);  return 0;  } |
| 執行結果 | |
| 1->3->4->5->2->7->6-> | |

執行結果與剛才講解的順序相同。



可以稍微修改上面的程式碼，若檢查相鄰頂點是否為白色時，發現該頂點顏色為灰色，即 ，就代表圖中有環。

若要確認該圖是否為二分圖（把所有頂點都放入兩個集合 、 之一中，使得所有邊 中 和 必不在同一集合，參考圖論章節），也可以透過深度優先搜尋把相鄰的點交互塗成不同顏色（黑白染色），如果把整張圖的頂點都塗完後，不會發生同樣顏色頂點相鄰的情況，就代表該圖為二分圖。

（8）LeetCode #207. 課程排序 Course Schedule

A. 題目

你必須修習總共 門課，編號分別為 到 。給定一個陣列 ，其中 代表必須要先修過「」這門課，才能往下修「」這門課（注意順序不是反過來）。

舉例來說，如果有陣列中有 ，就代表必須先修過課程 1，才可以修課程 0。如果有任一種方法能夠把所有課程修完，回傳 true，否則回傳 false。

B. 出處

https://leetcode.com/problems/course-schedule/

C. 範例輸入與輸出

輸入：numCourses = 2, prerequisites = [[1,0]]

輸出：true

先修完課程 0，就可以修課程 1。

輸入：numCourses = 2, prerequisites = [[1,0],[0,1]]

輸出：false

因為課程 0 和課程 1 互為先修課程，所以無法找到可以修完的順序。

D. 解題邏輯

本題先前已經做過，可以改以 DFS 檢查「環的存在」來解決本題。

|  |  |
| --- | --- |
| 課程排序 Course Schedule | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71 | class Solution{  // 儲存頂點顏色的向量  vector<int> color;  // 儲存邊的向量  vector<vector<int>> edges;  bool check\_cycle(int vertex){    // 把頂點 vertex 塗成灰色  color[vertex] = 1;    // 檢查頂點 vertex 的每個相鄰頂點  for (int i=0 ; i<edges[vertex].size() ; i++){  // 如果該相鄰頂點是白色  if (color[edges[vertex][i]]==0){    // 從該相鄰頂點出發，再往下檢查是否有環  bool cycled = check\_cycle(edges[vertex][i]);    // 回傳類型 A：  // 該頂點往下找會有環時，回傳 true，代表圖中有環  if (cycled){return true;}    }  // 如果該相鄰頂點是灰色  else if (color[edges[vertex][i]]==1){  // 回傳類型 B：  // 該頂點相鄰頂點是灰色，回傳 true，代表圖中有環  return true;  }    }  // 頂點 vertex 處理完成  color[vertex] = 2;    // 回傳類型 C：  // 從這個頂點向下都沒有遇到環時，回傳 false  return false;  }  public:  bool canFinish(int numCourses, vector<vector<int>>& prerequisites){  // 初始化 color 與 edges 向量  color.resize(numCourses);  edges.resize(numCourses);  // 取出先修資訊建立每條邊  for (auto edge:prerequisites){  // edge[1] -> edge[0]  edges[edge[1]].push\_back(edge[0]);  }  // 不一定是連通圖  // 如果跑完一次 DFS 還有白色頂點，要接續再做另一次 DFS  for (int i=0 ; i<numCourses ; i++){  // 確定從頂點 i 出發會不會有環  if (color[i]==0){  bool cycled = check\_cycle(i);  // 有環時回傳 false，代表無法把所有課修完  if (cycled)  return false;  }  }  return true;  } // end of canFinish  }; // end of Solution |

（9）LeetCode #785. 判別二分圖 Is Graph Bipartite?

A. 題目

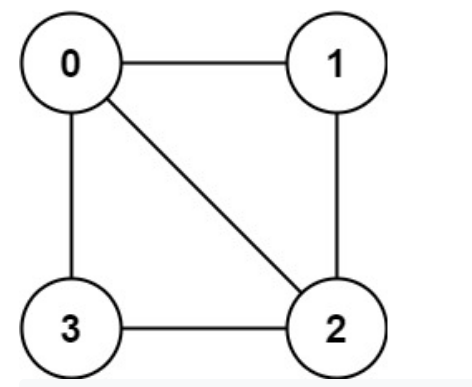
有一個共有 個節點的無向圖，每個節點分別被編號為 到 。給定一個二維陣列 ，其中 是含有節點 所有相鄰節點的陣列。也就是說，如果 中有節點 ，則節點 和節點 之間有一條無向邊。

如果一個圖是二分圖，該圖的所有節點可以被分到兩個獨立的集合 和 之一當中，使得該圖上的每一條邊都是連接一個「集合 中的點」與一個「集合 中的點」。

只有在該圖為二分圖時，回傳 true。

B. 出處

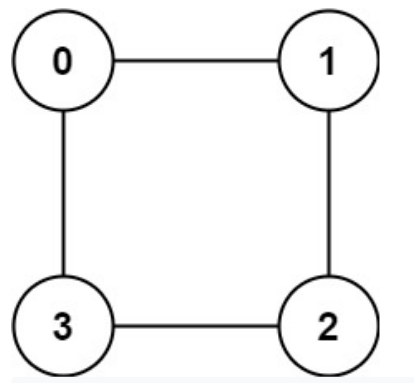
https://leetcode.com/problems/is-graph-bipartite/

C. 範例輸入與輸出

輸入：graph = [[1,2,3],[0,2],[0,1,3],[0,2]]

輸出：false

沒有辦法將頂點分成兩個符合要求的集合。



輸入：graph = [[1,3],[0,2],[1,3],[0,2]]

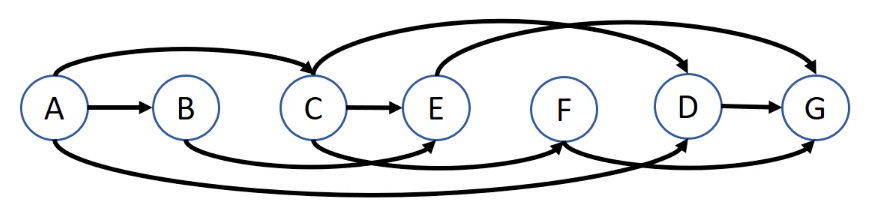
輸出：true

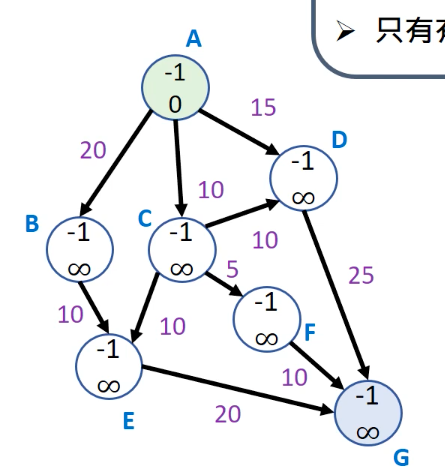
分為 {0,2}、{1,3} 兩個集合時，所有邊都跨越了兩個集合。

|  |  |
| --- | --- |
| 判別二分圖 Is Graph Bipartite? | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80 | class Solution{  // 記錄頂點是否造訪過（DFS 中的白、灰、黑）  vector<int> visited;  // 頂點的顏色（黑白染色的顏色）  vector<int> color;  bool DFS(int vertex, int color\_now, vector<vector<int>>& graph){  // 頂點塗成灰色  visited[vertex] = 1;  // 依序檢查相鄰節點  for (int i=0 ; i<graph[vertex].size() ; i++){    int vertex\_now = graph[vertex][i];  // 如果相鄰節點的染色與 vertex 的染色衝突  // 比如 vertex 的染色是「0」  // 但檢查到相鄰節點也已經被染色為「0」  if (color[vertex\_now]==color\_now){  // 並非二分圖  return false;  }  // 如果相鄰頂點是白色頂點（未造訪過）  if (visited[vertex\_now]==0){  // 如果相鄰頂點的染色是「-1」（未染色過）  if (color[vertex\_now] == -1){  // 把相鄰頂點染上「相反」顏色  // vertex 染色為 1 時，相鄰頂點染上 0  // vertex 染色為 0 時，相鄰頂點染上 1  color[vertex\_now] = !color\_now;  }  // 往相鄰頂點繼續進行  bool flag = DFS(vertex\_now, !color\_now, graph);  if(!flag){return false;}  }  } // end of for  // 頂點 vertex 處理完成  visited[vertex] = 2;  // 若從頂點 vertex 往下染色都不會發生衝突，回傳 true  return true;  }  public:  bool isBipartite(vector<vector<int>>& graph){  // 取出頂點個數  int vertex = graph.size();  // 初始化向量  visited.resize(vertex);  // -1 代表還未著色過（黑白染色）  color.resize(vertex,-1);  // 對所有白色頂點進行 DFS  // 過程中用 color 另外染色（與原本的白、灰、黑獨立）  // 染色順序為 0 1 0 1...  for (int i=0 ; i<vertex ; i++){  if (visited[i]==0){  color[i] = 0;  // 從起點開始，檢查能不能順利進行黑白染色  // 起點的染色為「0」  bool flag = DFS(i,0,graph);  // 如果染色過程中產生衝突，回傳 false  // 代表非二分圖  if(!flag){return false;}  }  } // end of for  // 沒有衝突時，代表為二分圖  return true;  } // end of isBipartite  }; // end of Solution |

2. 拓樸排序

接下來，來看如何用 DFS 列出拓樸排序。

之前在圖論的章節有介紹過拓樸排序的定義：若一個有向無環圖中有 ，拓樸排序中 必在 之前。

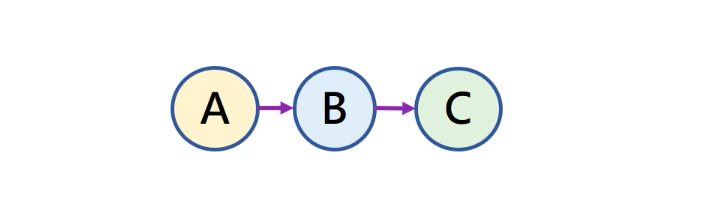


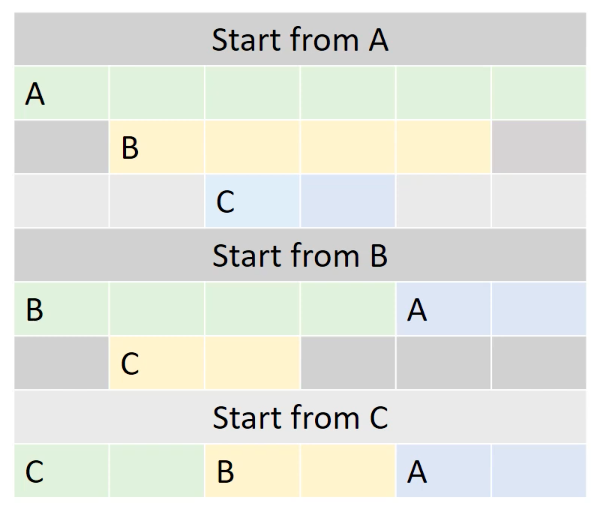
也就是說，把圖中的所有頂點加以排序，在排序結果中加上原先的邊，則所有的邊一定是從「左邊」指向「右邊」，不會有方向相反的邊。

經過拓樸排序後，可以把資料，如人口、車流等等的流動看作是由左往右。像「修課地圖」就是一個拓樸排序的應用。

類似的，安排編譯順序也會用到拓樸排序，比如把每個頂點看作一個 .cpp 檔，這些檔案之間可能相互引用，如果檔案 B 引用了檔案 A，那麼一定要先編譯完 A 才能編譯 B，因此，可以透過拓樸排序來決定哪些檔案應該先被編譯。工程各部分的「完工順序」也類似，可以用拓樸排序進行規劃。

在圖論的章節中，是利用「入度為 0 的頂點不可能形成環」的性質來依序將頂點放入排序當中，本節則探討如何利用 DFS 得到拓樸排序。

（1）DFS 與拓樸排序

上圖中，共有 A、B、C 三個頂點，進行 DFS 時可以任選三者中之一做為起點開始搜尋。

如果從頂點 A 開始搜尋，會先造訪 A，隨後造訪 B，再造訪 C，如上圖所示。到了 C 之後，因為沒有繼續往下的可行路徑，所以 C 會先從堆疊中被移除，也就是「離開」C，接下來，也會依序離開 B 和 A。

若改從頂點 B 開始，造訪 B 後會往下造訪 C，之後離開 C、離開 B，最後再從此時還是白色頂點的 A 再做一輪 DFS（這輪只會造訪 A 並離開 A）。

改從頂點 C 開始，造訪 C 後，從 C 離開。假設接續從 B 再做一次 DFS，同樣造訪 B 後再離開。最後，再從 A 開始做一次 DFS，最後由 A 離開。

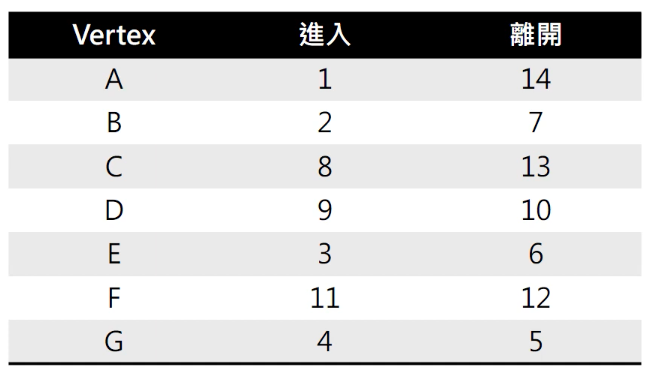
從上面三種情形可以發現，無論選擇從哪一點開始做 DFS 直到造訪圖中的所有頂點，「離開」頂點的順序都為拓樸排序的反向，即 C、B、A。

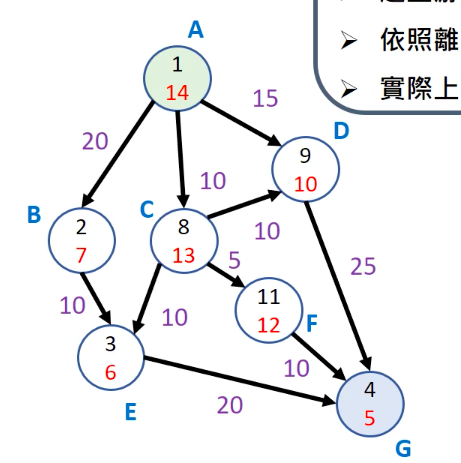
這是因為在 DFS 中，離開一個頂點的時間點一定在該頂點的「下游」完全處理完之後，因此把該頂點放在拓樸排序的「上游」時，可以保證所有出邊方向都為由左往右。

（2）利用 DFS 產生拓樸排序

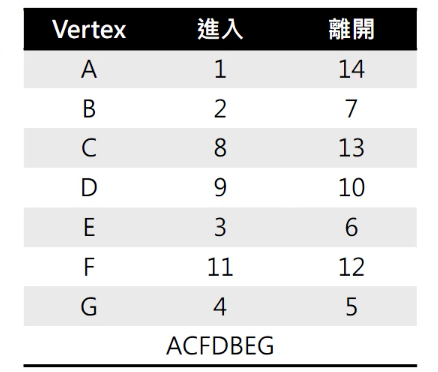
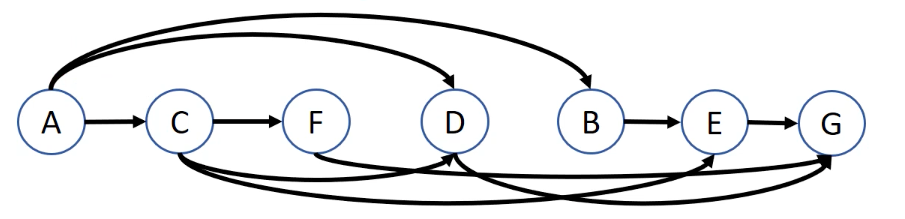
A. 進行 DFS 時，把路過頂點的次序記錄下來

B. 依照離開每個頂點的時間戳記，就可以得到拓樸排序

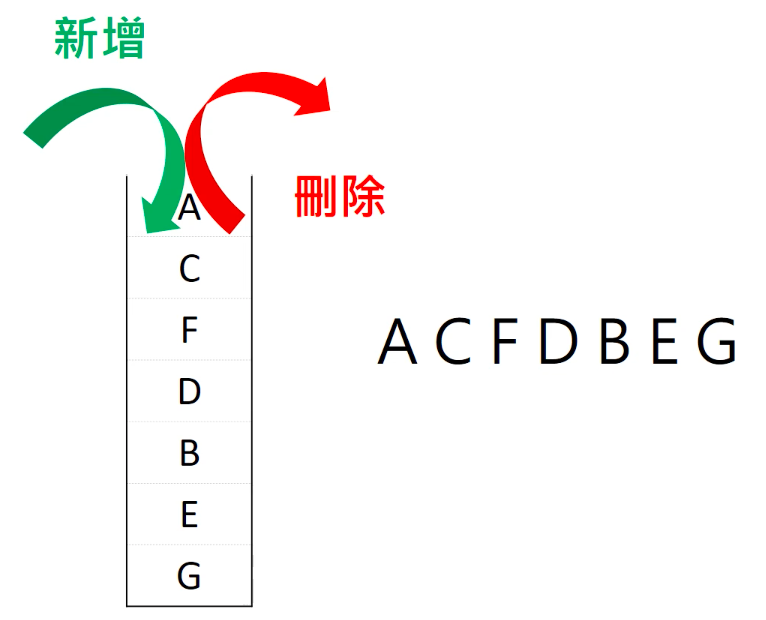
C. 「進入」頂點（進入 Stack）指的是將頂點塗成灰色的瞬間，「離開」頂點（離開 Stack）是將頂點塗成黑色的瞬間



進行 DFS 的過程中，先離開的頂點位在拓樸排序的「下游」（右邊）。上表中，先從 A 出發，經過 的路徑，記錄進入時間，接下來，一路回退到 A 的過程中，記錄經過頂點的離開時間。

回退到 A 後，往 C 的方向移動，並接著走 D，因為 D 沒有往下的可行路徑，退回 C 並記錄 D 的離開時間，接下來從 C 移動到 F，離開時記錄時間，最後記錄 C 的離開時間與 A 的離開時間。

由離開的時間戳記「由大到小」排序，可以得到一個拓樸排序： ，但過程中會發現有許多任意選擇往下路徑的時機，因此拓樸排序並不是唯一的。

實作上，可以不用真的產生時間戳記，只需在 DFS 的過程中每次離開一個頂點時，把頂點放進一個用於產生拓樸排序的 Stack 裡，待所有頂點都處理完並離開後，該 Stack「由上而下」正好是拓樸排序中的頂點順序。

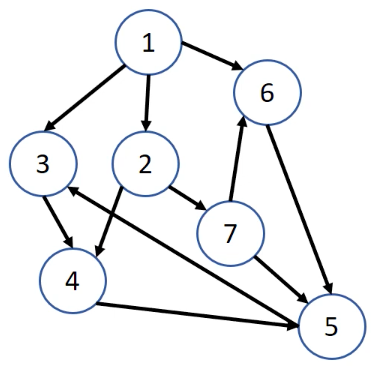
由於 Stack 後進先出的特性，正好是「由上而下取用」，只要一路用 pop() 將當中的資料取出，就會符合拓樸排序上游到下游的頂點順序。

（3）實作 DFS 產生拓樸排序

擴寫 DFS 的程式碼：

|  |  |
| --- | --- |
| 利用 DFS 產生拓樸排序 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 | #include <iostream>  #include <stdlib.h>  #include <vector>  #include <list>  using namespace std;  class Graph{  private:  // 紀錄拓樸排序結果的 Stack  stack<int> topological\_sort;  int vertex;  vector<list<int>> edges;  vector<int> color;  void DFS\_visit(int);  // 拓樸排序的造訪函式  void DFS\_visit\_topological();  public:  Graph(int);  void print\_edges();  bool add\_edge(int, int);  void DFS(int);  // 產生拓樸排序的主函式  // 傳入一個起點，可任意選定，不妨預設為 1  void get\_topological\_sort(int=1);  };  // 建構式  Graph::Graph(int v){  vertex = v;  edges.resize(vertex);  color.resize(vertex);  }  /\* 其他函式的程式碼 \*/  void Graph::get\_topological\_sort(int start){  // 起點調整成索引值  start--;  // 所有頂點塗成白色  fill (color.begin(), color.end(), 0);  // 先從 start 開始進行一次 DFS  DFS\_visit\_topological(start);  // 一次 DFS 不一定會造訪到所有頂點  // 拓樸排序要把所有頂點造訪過才能結束  // 因此要進行至所有頂點都被處理完成  for (int i=0 ; i<vertex ; i++){  if (color[i]==0){  // 用造訪的函式從白色頂點出發進行 DFS  DFS\_visit\_topological(i);  }  }  // 上面的迴圈執行完後  // 拓樸排序已經在 topological\_sort 堆疊中  cout << "Topological sort:" << endl;  while (!topological\_sort.empty()){  // 依序移除並印出堆疊中的頂點  int current = topological\_sort.top();  topological\_sort.pop();  // 印出時，從索引值調整成頂點編號  cout << current+1 << "->";  }  }  void Graph::DFS\_visit\_topological(int vertex){  // 把頂點塗成灰色  color[vertex] = 1;  for (auto iter = edges[vertex].begin() ; iter!=edges[vertex].end() ; iter++){  int current\_vertex = \*iter;  if (color[current\_vertex]==0){  DFS\_visit\_topological(current\_vertex);  }  }  // 把頂點 vertex 塗成黑色  color[vertex] = 2;  // 塗成黑色的瞬間，要加入拓樸排序堆疊中  topological\_sort.push(vertex);  }  int main(){  Graph g(7);  g.add\_edge(1,3);  g.add\_edge(1,2);  g.add\_edge(1,6);  g.add\_edge(2,4);  g.add\_edge(2,7);  g.add\_edge(3,4);  g.add\_edge(4,5);  g.add\_edge(6,5);  g.add\_edge(7,6);  g.add\_edge(7,5);  g.DFS(1);  g.get\_topological\_sort(1);  return 0;  } |
| 執行結果 | |
| 1->3->4->5->2->7->6->  Topological sort:  1->2->7->6->3->4->5-> | |

在拓樸排序中加入原圖的所有邊，方向皆為由左至右。



（4）重訪 LeetCode #210. 課程排序 II Course Schedule II

A. 題目

你必須修習總共 門課，編號分別為 到 。給定一個陣列 ，其中 代表必須要先修過「」這門課，才能往下修「」這門課（注意順序不是反過來）。

舉例來說，如果有陣列中有 ，就代表必須先修過課程 1，才可以修課程 0。回傳任一種能夠把所有課程修完的修課順序，如果沒有任何可行的順序，回傳一個空陣列。

B. 出處

https://leetcode.com/problems/course-schedule-ii/

C. 範例輸入與輸出

輸入：numCourses = 2, prerequisites = [[1,0]]

輸出：[0,1]

先修完課程 0，就可以修課程 1，回傳修課順序的陣列 [0,1]。

輸入：numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]

輸出：[0,2,1,3]

以 [0,1,2,3] 和 [0,2,1,3] 的順序都可以修完所有課程，任選其一回傳即可。

D. 解題邏輯

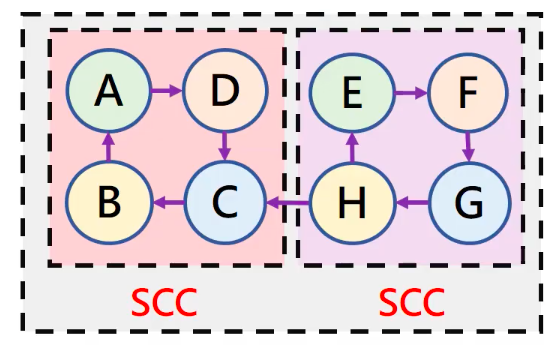
本題可以擴寫上一節中「LeetCode #207 課程排序」的程式碼。

|  |  |
| --- | --- |
| 課程排序 II Course Schedule II | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74 | class Solution{  vector<int> color;  vector<vector<int>> edges;  // 儲存拓樸排序的向量  vector<int> topological\_sort;  bool check\_cycle(int vertex){  color[vertex] = 1;  for (int i=0 ; i<edges[vertex].size() ; i++){  if (color[edges[vertex][i]]==0){    // 從該相鄰頂點出發，再往下檢查是否有環  bool cycled = check\_cycle(edges[vertex][i]);    // 回傳類型 A：從該頂點往下找會有環時  // 回傳 true，代表圖中有環  if (cycled){return true;}    }  // 如果該相鄰頂點是灰色  else if (color[edges[vertex][i]]==1){  // 回傳類型 B  // 該頂點的相鄰頂點是灰色，回傳 true，圖中有環  return true;  }    }  // 頂點 vertex 處理完成  color[vertex] = 2;  // 這裡用的是向量，每次把資料插到向量「前端」  topological\_sort.insert(topological\_sort.begin(),vertex);    // 回傳類型 C：從這個頂點向下沒有遇到環  return false;  }  public:  vector<int> findOrder(int numCourses, vector<vector<int>>&  prerequisites){  // 初始化 color 與 edges 向量  color.resize(numCourses);  edges.resize(numCourses);    // 無法修完所有課時，回傳空向量  vector<int> empty;  // 取出先修資訊建立每條邊  for (auto edge:prerequisites){  // edge[1] -> edge[0]  edges[edge[1]].push\_back(edge[0]);  }  // 從每個白色頂點出發作 DFS  for (int i=0 ; i<numCourses ; i++){  // 確定從頂點 i 出發會不會有環  if (color[i]==0){  bool cycled = check\_cycle(i);  // 有環時回傳 false，代表無法把所有課修完  if (cycled)  return empty;  }  }  // 回傳拓樸排序  return topological\_sort;  } // end of findOrder  }; // end of Solution |

3. 強連通元件

接下來，探討如何使用 DFS 來尋找強連通元件。

（1）強連通元件 SCC

複習一下強連通元件 SCC 的定義：在一個強連通元件中，任兩點 間一定存在 及 的路徑。

比方說，上圖中有兩個強連通元件：ABCD 是一個強連通元件，EFGH 也是一個強連通元件。

從 ABCD 中任一個頂點出發，都可以到其他任一個頂點，因此 ABCD 本身是一個強連通元件，但是從 ABCD 四個頂點出發，都沒辦法走到 EFGH 四個頂點，因此兩個元件間不能組成一個更大的強連通元件。

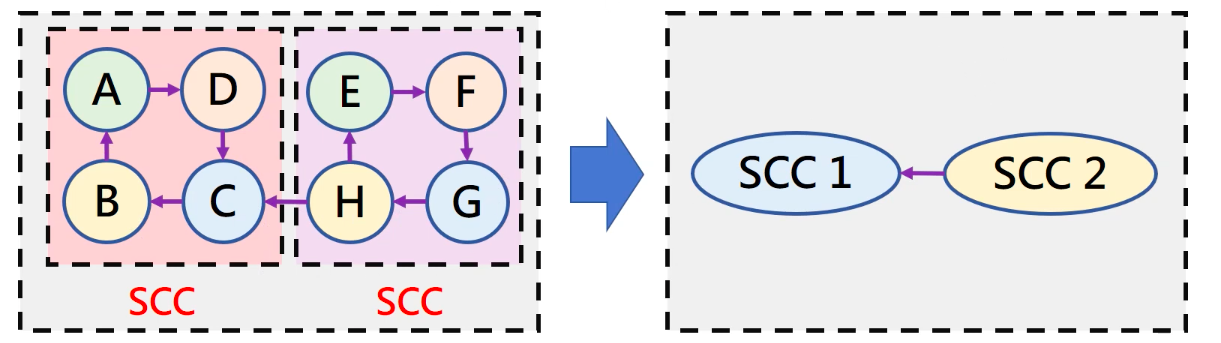
要怎麼找到圖中的強連通元件呢？從某個強連通元件的任意頂點出發作 DFS，必定會經過該元件中的「每個頂點」，因為既然有「強連通」的特性，任兩頂點間都有路徑，那麼在 DFS 中就一定會被尋訪到。比如上圖中從頂點 A 出發作 DFS，一定會走遍 ABCD 四個頂點。

（2）利用 DFS 找出強連通元件

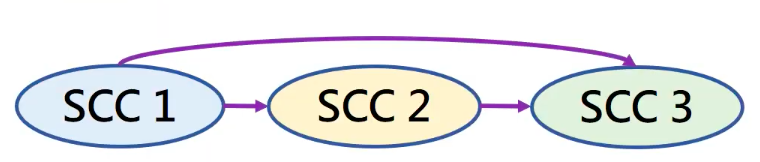
在「適當的順序」下，每次 DFS 走過的所有頂點都可以視為在同一個 SCC 內。

比如先從頂點 A 出發，經過 後 DFS 結束，再從頂點 E 出發，經過 之後 DFS 結束（因為 ABCD 被標記為已經處理完成了）。這樣一來，就找到了圖中的兩個強連通元件。

然而，如果第一次就從頂點 E 出發，那麼 DFS 就會把整張圖跑遍，但整張圖並不全部屬於一個強連通元件。因此，如何選定每次 DFS 的出發頂點，就顯得特別重要。



如右上圖所示，若把每個 SCC 看作「一個頂點」，則會形成「有向無環圖 DAG」。

DAG 內並不會有環，比如 SCC2 有出邊連到 SCC1，但是 SCC1 沒有出邊連到 SCC2，因為若兩個 SCC 可以互通，那一開始就會屬於同一個 SCC 了。

應用這個特性，如果先把上述由每個 SCC 組成的 DAG 列出，那麼從下游 SCC 的頂點出發做 DFS，並不會做一做跑到上游 SCC 的頂點當中（因為該 DAG 中沒有由右往左的邊，所以不可能發生從 SCC3 內出發，卻有路徑走到 SCC2 或 SCC1 的情形），這樣一來，就可以保證每次 DFS 都只跑過一個特定 SCC 中的所有點，而不會誤跑入其他 SCC 中。

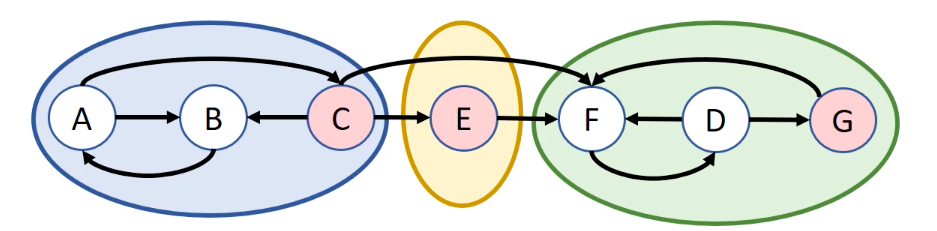
實作上，由 DAG 的下游開始，第一次 DFS 會找出 SCC3 包含的所有頂點，第二次 DFS 會找出 SCC2 的所有頂點，第三次 DFS 則會找出 SCC1 的所有頂點，待所有頂點都處理完成後，就找出了所有 SCC。

（3）找到位在「下游」的 SCC

根據上面的敘述，只要能夠分辨出哪些頂點會在「下游」的 SCC 中，哪些頂點會在「上游」的 SCC 中，就可以先跑完下游的 SCC，再跑完上游的 SCC，直到找到圖中所有強連通元件。

然而，如何知道哪些頂點位在下游，哪些頂點位在上游呢？

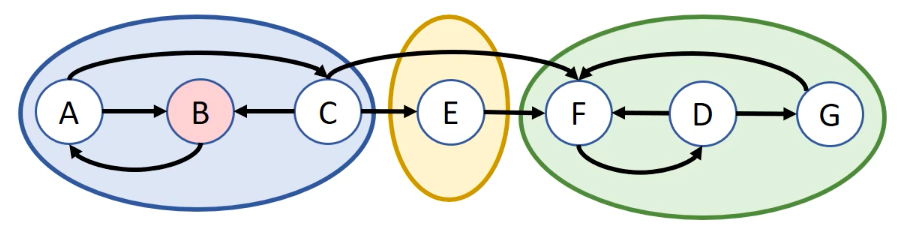
依據之前用 DFS 產生拓樸排序的經驗，可能會產生一種假設：「DFS 中越早離開的頂點，會位在越下游的 SCC 中」，這是不是正確的假設呢？（當然必須注意到，事實上只有「有向無環圖」才能做拓樸排序）。



先看一個成功的例子，上面的圖中，預先知道 ABC 是一個 SCC，E 是另一個 SCC，FDG 是第三個 SCC，而 是這三個 SCC 間的上下游關係。

任意選定頂點 B 出發做 DFS，進入每個頂點的順序是 BACEFDG，離開的順序則與此反向，為 GDFECAB。根據剛才的假設「越早離開的頂點位在越下游的 SCC 中」，首先會從最早離開的 G 做一次 DFS，跑遍頂點 GFD，再來，從還沒跑過的頂點中最早離開的 E 進行 DFS，只會跑過 E 本身，最後，由還沒跑過的頂點中最早離開的 C 出發，會跑過頂點 CBA。這樣一來，就順利找到了圖中的三個 SCC。

然而剛才是任意選定了頂點 B 出發做 DFS，如果選到其他頂點，假設還會成立嗎？



若任意選定到頂點 A，就會產生錯誤。由頂點 A 出發，進入順序可能是 ABCEFDG，那麼對應的離開順序是 BGDFECA。同樣根據假設「越早離開的頂點位在越下游的 SCC 中」，會從 B 開始做 DFS，試圖找到與 B 位於同一個 SCC 中的所有頂點，然而從 B 出發後，卻把 ABCDEFG 全部頂點都跑完了，而它們並非都與 B 在同一個 SCC 中。

這表示單靠「DFS 中，越早離開的頂點位在越下游的 SCC 中」這個假設並不穩固，因為任選 DFS 的起點時，有時會產生正確的離開順序，有時卻產生錯誤的離開順序。

（4）顛倒圖

觀察剛才的兩次嘗試：

從 B 出發做 SCC（成功）

進入：BACEFDG

離開：GDFECAB

從 A 出發做 SCC（失敗）

進入：ABCEFDG

離開：BGDFECA

實際上 SCC 在 DAG 中的順序（上游 -> 下游）

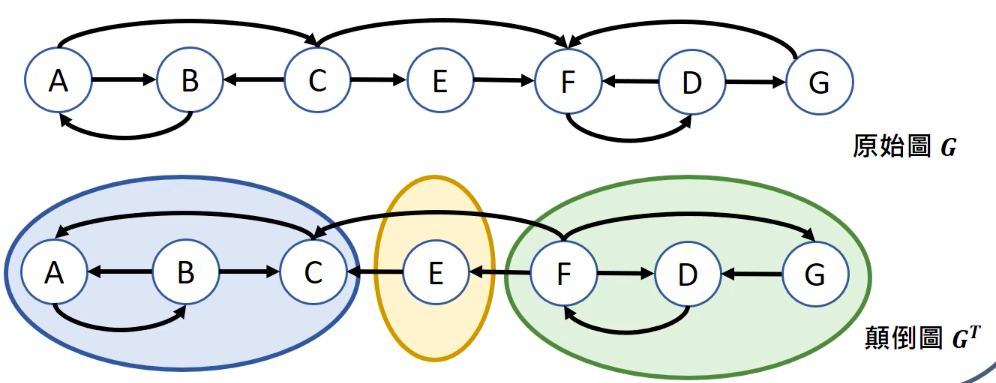
（）

雖然「最早離開的頂點」並不一定位在最下游的 SCC 當中，但是「最晚離開的頂點」卻會在最上游的 SCC 當中！

先看第一次嘗試，離開的順序是 GDFECAB，最晚離開的 CAB 三個頂點，在最上游的 SCC1 當中，剩下頂點中最晚離開的 E 位在 SCC2 中，其餘的 GDF 則在最下游的 SCC3 中。

第二次嘗試裡也一樣，離開的順序是 BGDFECA，若不看作為出發頂點的 B，最晚離開的頂點 AC 在最上游的 SCC1，較早離開的 E 在 SCC 2，其餘的 GDF 在最下游的 SCC2。

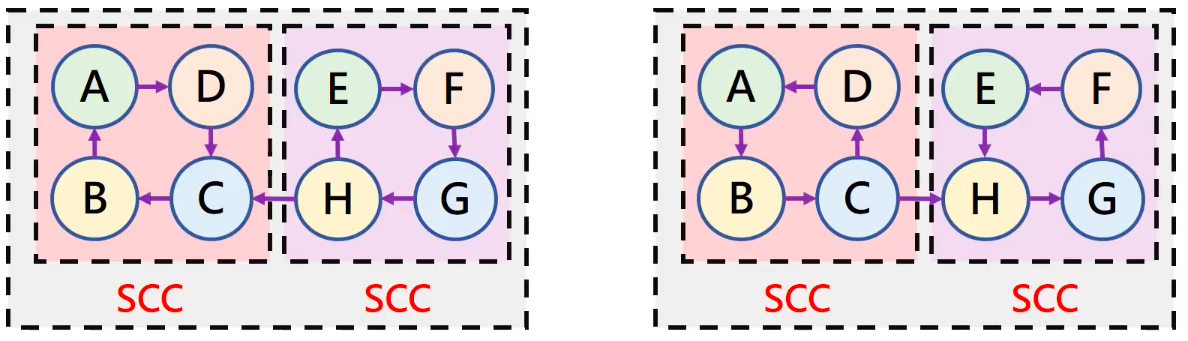
「最晚離開的頂點會在最上游的 SCC 當中」是成立的，但是根據前面的討論，要完成找到所有 SCC 的目標，一定要從 DAG 「最下游」的 SCC 開始進行，不能從「最上游」開始進行，那該怎麼辦呢？



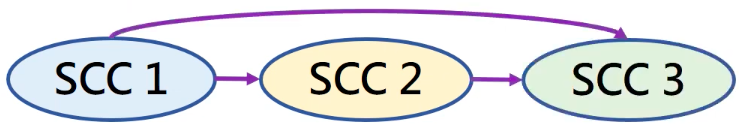
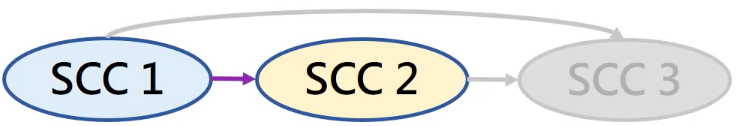
如果圖 是圖 的顛倒圖，那麼 中所有的頂點都與圖 相同，但是 中所有的邊是圖 中所有邊的反向。

可以想見，根據強連通元件的定義，顛倒圖的強連通元件會與原圖完全相同，因為兩個頂點間有路徑，那麼邊全部反向之後，仍然會有路徑，兩個頂點間如果沒有路徑，那麼邊全部反向之後，還是不會有路徑。

同時，把每個 SCC 看成一個頂點形成的 DAG 中，因為 SCC 間相互連接的邊方向反過來了，所以原圖中上游的 SCC 在顛倒圖中變成位在下游，原圖中下游的 SCC 在顛倒圖中則變成位在上游。



像上面的兩張圖中，把原圖左圖改為其顛倒圖如右圖，SCC 仍然相同，只有上下游關係變成相反。

這樣一來，就建立出了尋找所有 SCC 的可行方式：以原圖的 DFS 離開順序（最晚離開的先做）在 上尋找 SCC（先跑完下游的，然後依序往上游一個一個跑完）。

每輪 DFS 做完後，都不再理會已尋訪過的所有頂點，而剩下的頂點中，最晚離開的必在 剩下圖形中的最下游，可以接續找到剩下最下游的 SCC。

這樣一來，找到的 SCC 與原圖中所有的 SCC 相同，也就解決了「尋找所有強連通元件的問題」。

（5）實作「利用 DFS 找出強連通元件」

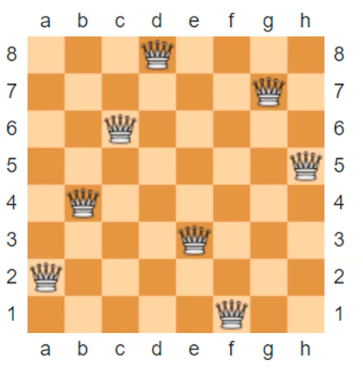
擴寫拓樸排序的程式碼：

|  |  |
| --- | --- |
| 利用 DFS 找出強連通元件 [其他函式] | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  39  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70 | #include <iostream>  #include <stdlib.h>  #include <vector>  #include <list>  using namespace std;  class Graph{  private:  stack<int> topological\_sort;  int vertex;  vector<list<int>> edges;  // 顛倒圖上的邊  vector<list<int>> edges\_inverse;  vector<int> color;  void DFS\_visit(int);  void DFS\_visit\_topological();  // 尋找 SCC 用的 DFS  void DFS\_SCC(int);  public:  Graph(int);  void print\_edges();  bool add\_edge(int, int);  void DFS(int);  void get\_topological\_sort(int=1);  // 找出所有 SCC  void print\_SCC();  };  // 建構式  Graph::Graph(int v){  vertex = v;  edges.resize(vertex);  // 初始化 edges\_inverse  edges\_inverse.resize(vertex);  color.resize(vertex);  }  // 要加上顛倒圖的邊 edges\_inverse  void Graph::print\_edge(){  edges[from-1].push\_back(to-1);  edges\_inverse[to-1].push\_back(from-1);  }  bool Graph::add\_edge(int from, int to){...}  void Graph::DFS(int start){...}  void Graph::DFS\_visit(int vertex){...}  // 程式碼見下述  void Graph::get\_topological\_sort(int start){…}  void Graph::DFS\_visit\_topological(int vertex){…}  void Graph::DFS\_SCC(int vertex){…}  void Graph::print\_SCC(){…}  int main(){  Graph g(7);  g.add\_edge(1,3);  g.add\_edge(1,2);  g.add\_edge(1,6);  g.add\_edge(2,4);  g.add\_edge(2,7);  g.add\_edge(3,4);  g.add\_edge(4,5);  g.add\_edge(6,5);  g.add\_edge(7,6);  g.add\_edge(7,5);  g.print\_SCC();  return 0;  } |

|  |  |
| --- | --- |
| 利用 DFS 找出強連通元件 [拓樸排序與尋找 SCC] | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81 | void Graph::get\_topological\_sort(int start){  start--;  fill (color.begin(), color.end(), 0);  DFS\_visit\_topological(start);  for (int i=0 ; i<vertex ; i++){  if (color[i]==0){  DFS\_visit\_topological(i);  }  }  /\*  只需用到 topological\_sort 中的頂點順序，不需印出  cout << "Topological sort:" << endl;  while(!topological\_sort.empty()){  int current = topological\_sort.top();  topological\_sort.pop();  cout << current+1 << "->";  }  \*/  }  // 要把輸出註解掉!  void Graph::DFS\_visit\_topological(int vertex){  ...  // cout << current\_vertex << endl;  ...  }  // 尋找 SCC 用的 DFS  // 注意是在顛倒圖 上做，因此邊的向量是 edges\_inverse  void Graph::DFS\_SCC(int vertex){  color[vertex] = 1;  // 輸出目前頂點  cout << char(vertex+65) << " ";    for (auto iter=edges\_inverse[vertex].begin() ;  iter!=edges\_inverse[vertex].end() ; iter++){  int current\_vertex = \*iter;  if (color[current\_vertex]==0){  DFS\_SCC(current\_vertex);  }  }  color[vertex] = 2;  }  void Graph::print\_SCC(){  // 呼叫建立拓樸排序的函式以取得離開順序  // 可以從任意頂點開始  get\_topological\_sort(0);  // topological\_sort 的順序是「最晚離開頂點」->「最早離開頂點」  // 按照 topological\_sort 的順序，在 上做 DFS  // 重新把所有頂點塗成白色（先前拓樸排序時會全部塗成黑色）  fill (color.begin(),color,end(),0);  // 計算圖中 SCC 數目  int SCC = 1;  while(!topological\_sort.empty()){  // 從 topological\_sort 中取出一個頂點  int current\_vertex = topological\_sort.top();  topological\_sort.pop();  if (color[current\_vertex]==0){  cout << "SCC #" << SCC << ":" << endl;  // 對 current\_vertex 做 DFS  DFS\_SCC(current\_vertex);  SCC++;  cout << endl;  }    } // end of while    } // end of print\_SCC |
| 執行結果 | |
| SCC #1:  A B C  SCC #2:  E  SCC #3:  G D F | |

4. N皇后問題

本節來看深度搜尋演算法中相當經典的應用：八皇后問題。

（1）八皇后問題

八皇后問題源自於西洋棋，在西洋棋中，「皇后」是最強的棋種，因為她不僅可以前後左右移動，也可以進行斜角移動。在一個 的棋盤上，若要安排複數個皇后所在的位置，使得她們無法在一步之內互吃，則最多只能放上 個皇后。

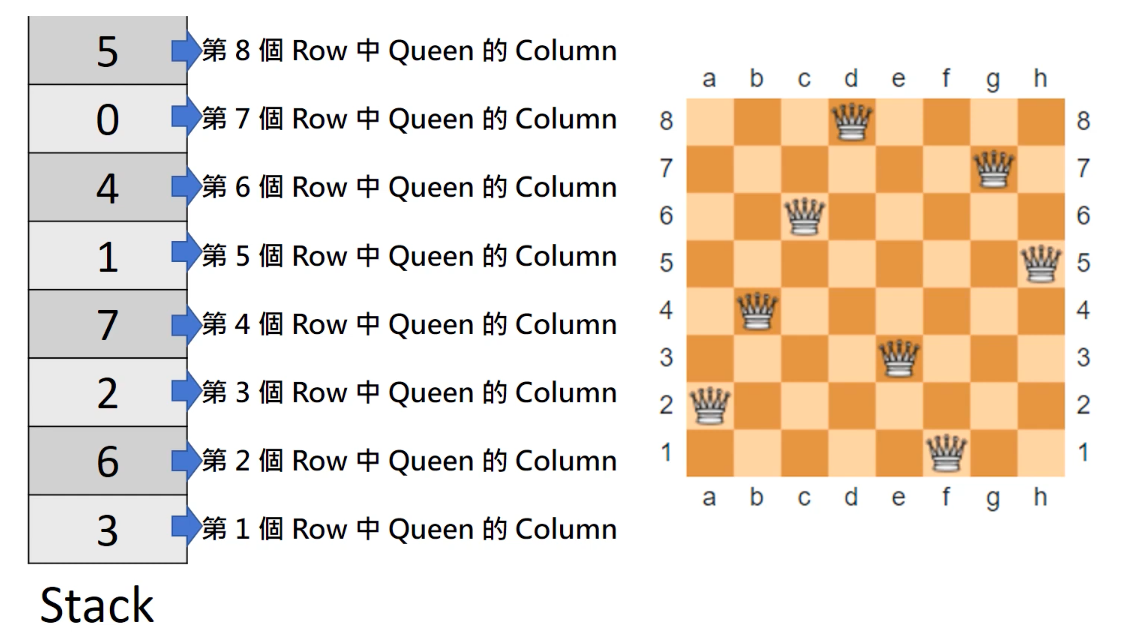
這是很直覺的結果，因為每個皇后都可以吃「直的」和「橫的」，所以每個直行和橫列上，必定只會有一個皇后， 的棋盤有 個直行、 個橫列，最多自然只能放上 個皇后。

至於為什麼關注 的「八皇后」呢？這是因為西洋棋的棋盤正是 的棋盤。傳統的「八皇后問題」，就是要在 的棋盤上放上 8 個皇后，使她們之間不會處於同一直行、同一橫列，或者同一對角線上。

稍後的實作中，會需要印出 stack 中的所有元素（並且印出後不改變 stack 內容），可以使用下列程式碼：

|  |  |
| --- | --- |
| 印出 stack 中的所有元素 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | void print\_stack(stack<int> s){  // 邊界條件：s 是空的  if (s.empty()){return;}  // 取出 s 中最上面的元素  int col = s.top();  s.pop();  // 遞迴取出下一個元素  print\_stack(s);  // 印出當前元素  cout << col+1 << " ";  // 將當前元素放回 stack 中  s.push(col);  } |

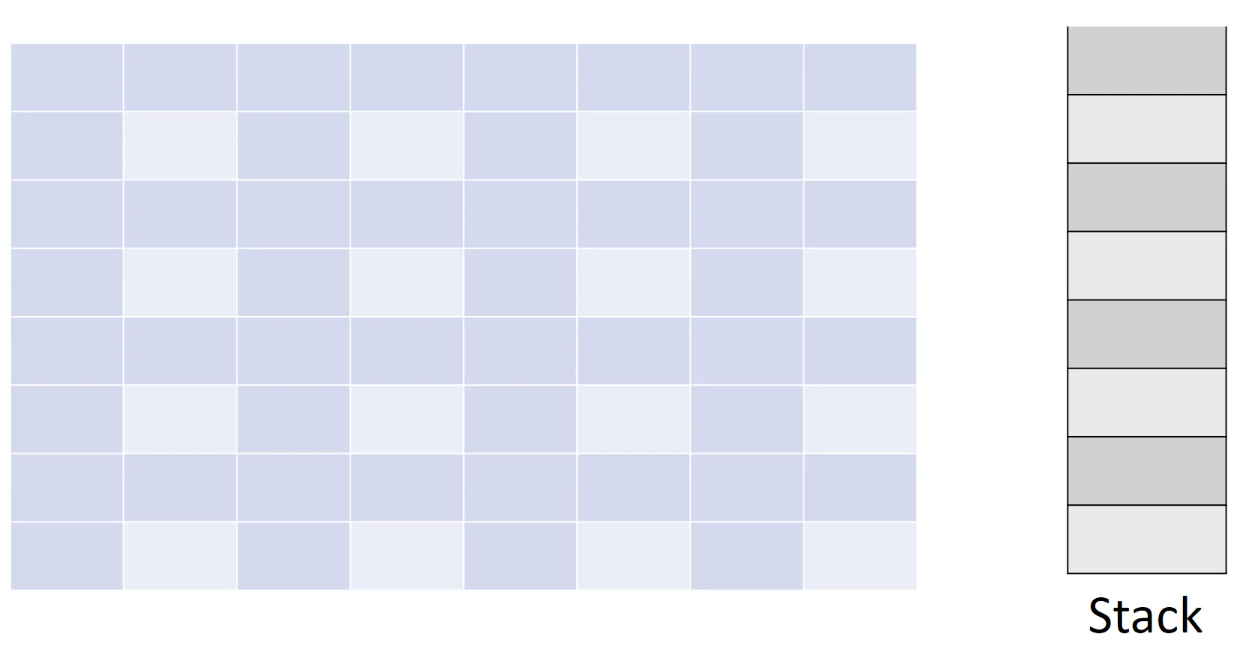
（2）DFS 解決八皇后問題的進行過程

要記錄棋盤上所有皇后的位置，可以只使用一個 Stack 來達成：

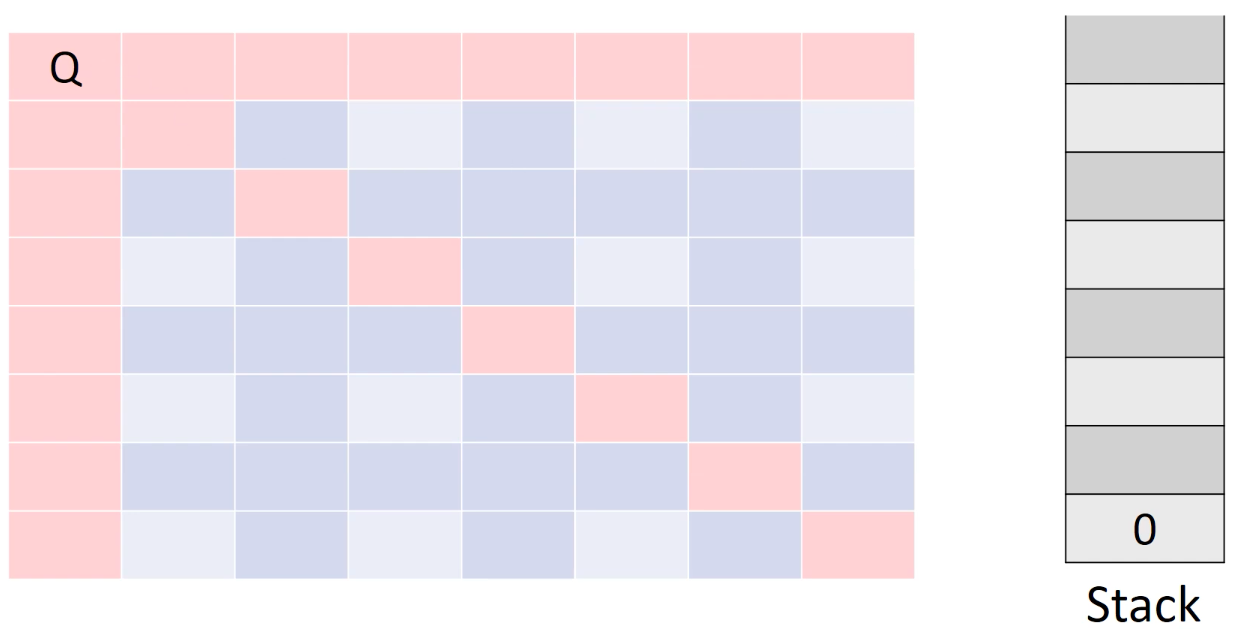
上面的 Stack 中，從下面數上來第 個元素代表「棋盤上第 個橫列中，皇后位在第幾個直行」。比如左邊的堆疊就對應了右邊的棋盤。

在找尋可行解的過程中，可以先在第一個橫列（最上面的橫列）中放一個皇后，接下來，依序在第二個、第三個、第四個、...、第八個橫列都放一個皇后，並且每次都要把新的皇后放到不會與先前皇后產生衝突的位置，如果一直放到第八個都還沒有產生衝突，就代表找到了一組可行解。

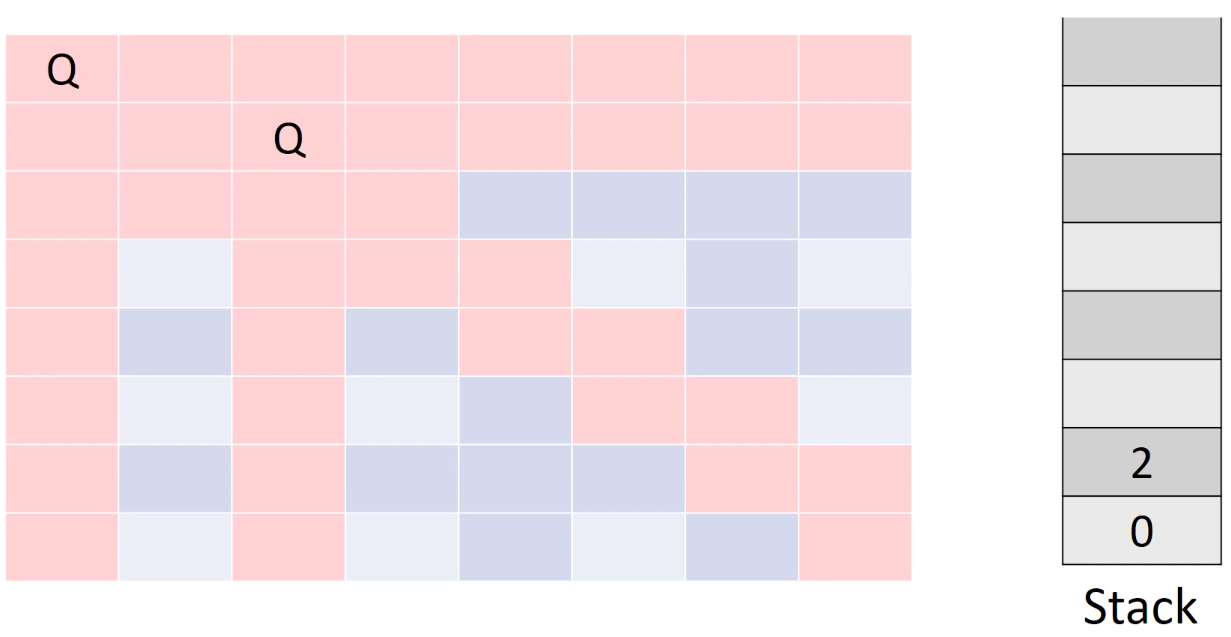
如果過程中某個皇后完全找不到位置放，代表先前的選擇不能產生可行解，就要回退到還未產生衝突時，選擇其他方法往下放，這正是深度優先搜尋的思考方式：無法往下走時，就退回路口選擇其他可行路徑。



上圖中，左邊是棋盤，右邊是記錄每個皇后位置的堆疊。

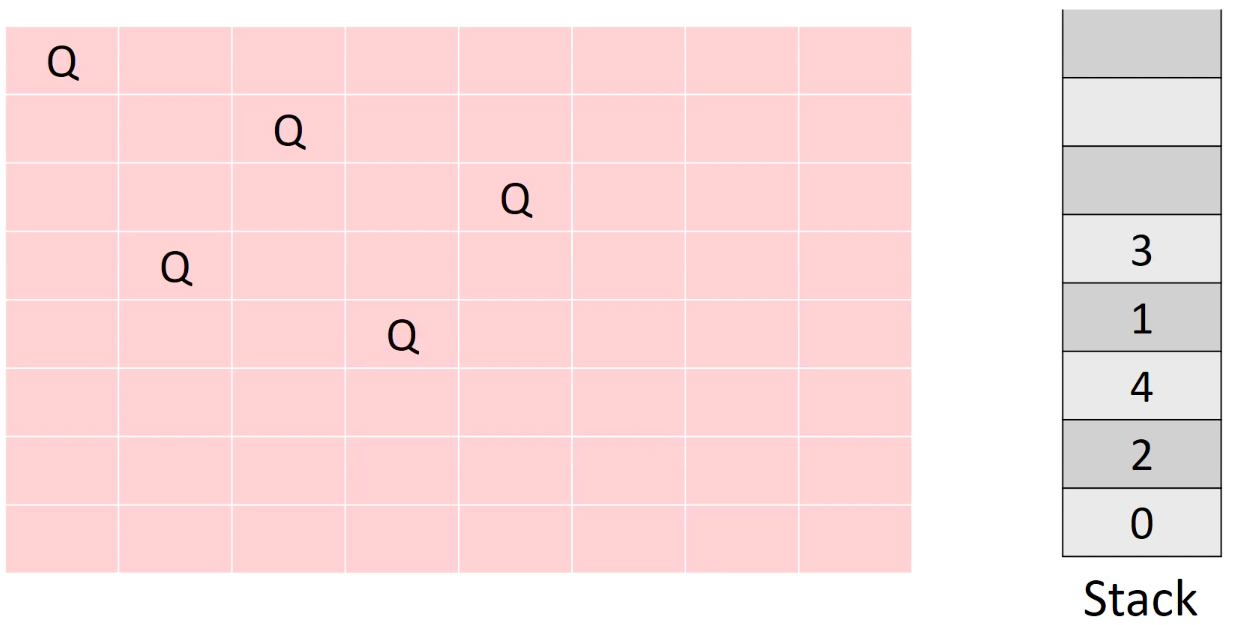


首先，在堆疊中放入 0，代表第一個橫列的皇后放到第一個直行上。因為皇后可以吃前後左右與斜角，因此棋盤上第一直行、第一橫列和左上到右下的對角就不能再放其他皇后了。

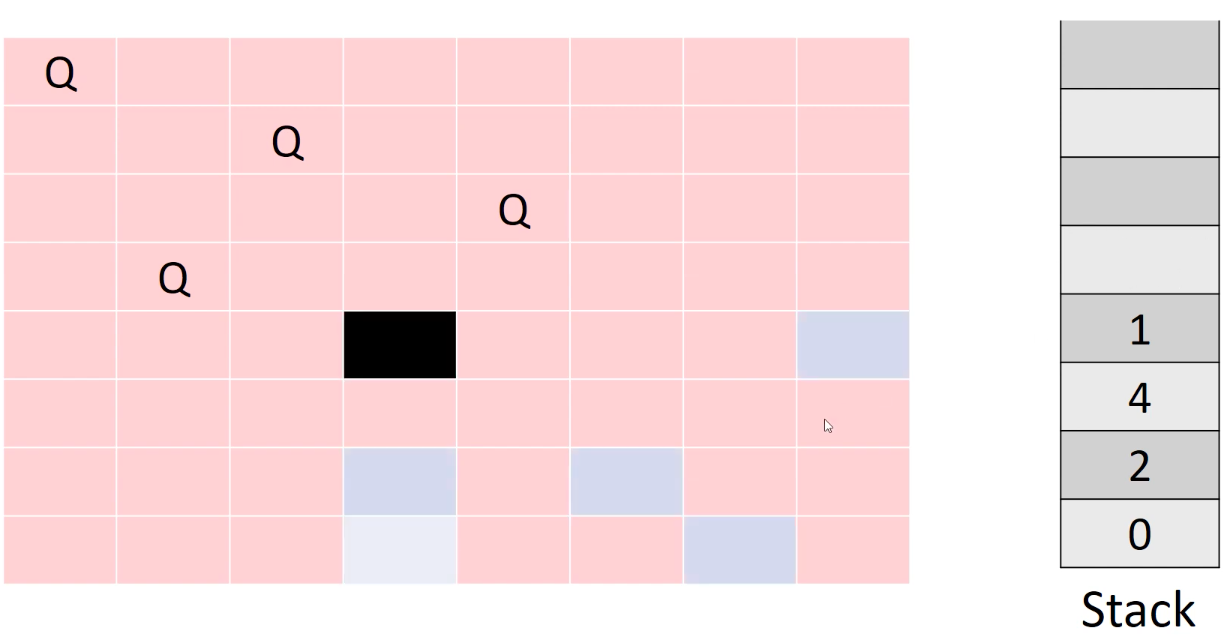


再來，如果選擇把第二個橫列的皇后放到第三個直行（注意索引值與棋盤的對應），棋盤上又有更多位置不能再放接下來的皇后。

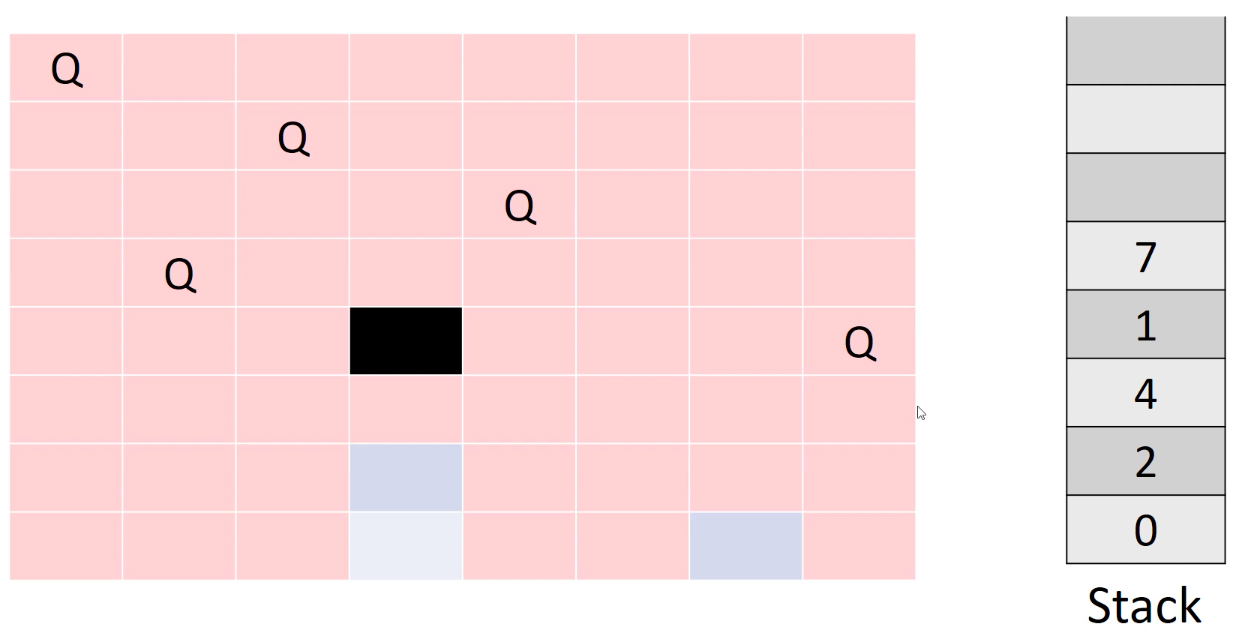
再來，選擇第三、第四個橫列的皇后放的位置，再把第五個皇后放到第四直行上。此時，棋盤上所有剩下的位置都不能放皇后了。



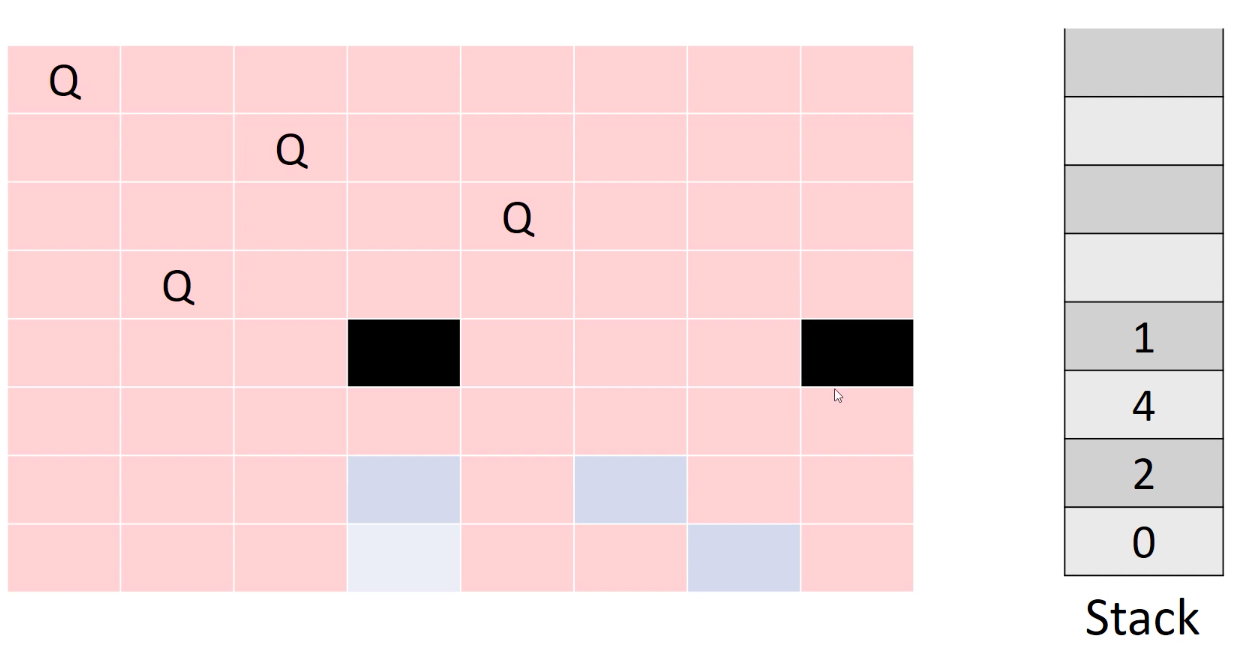
這代表先前五個皇后的位置安排是不能產生可行解的，必須回退選擇其他安排方式。



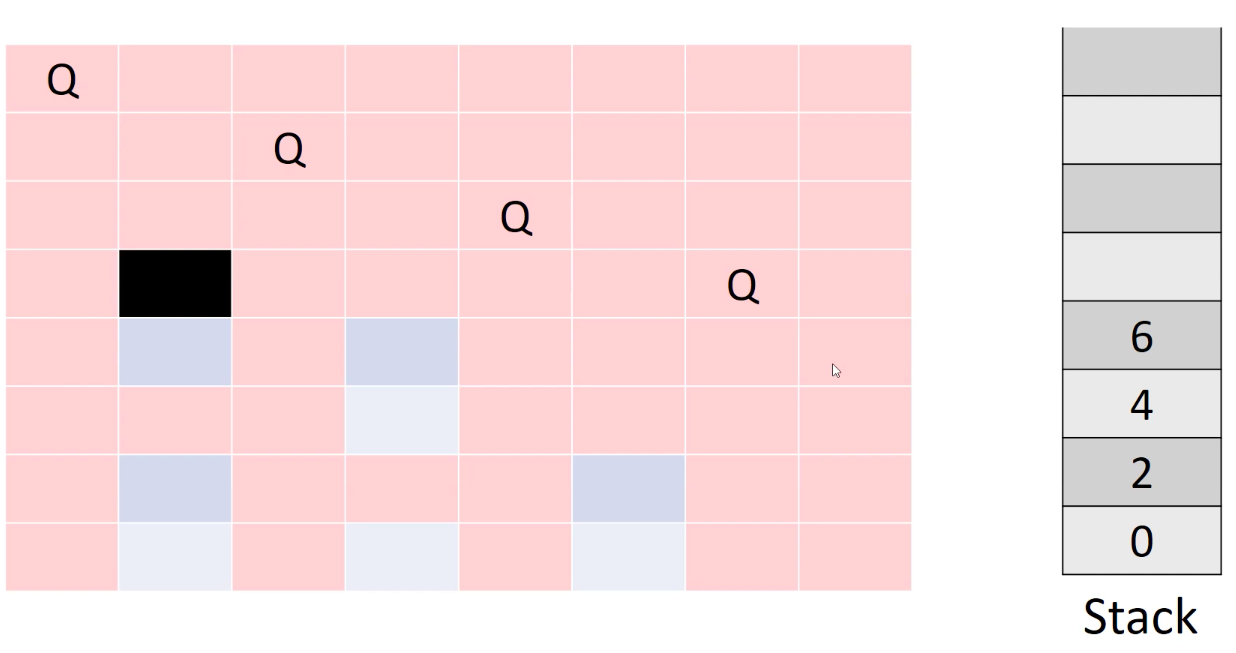
回退時，先把堆疊最上面的（最後放上棋盤的）皇后移除，並且剛才選擇放的位置不能再選，所以只能試著把第五橫列的皇后放到第八個直行上看看。



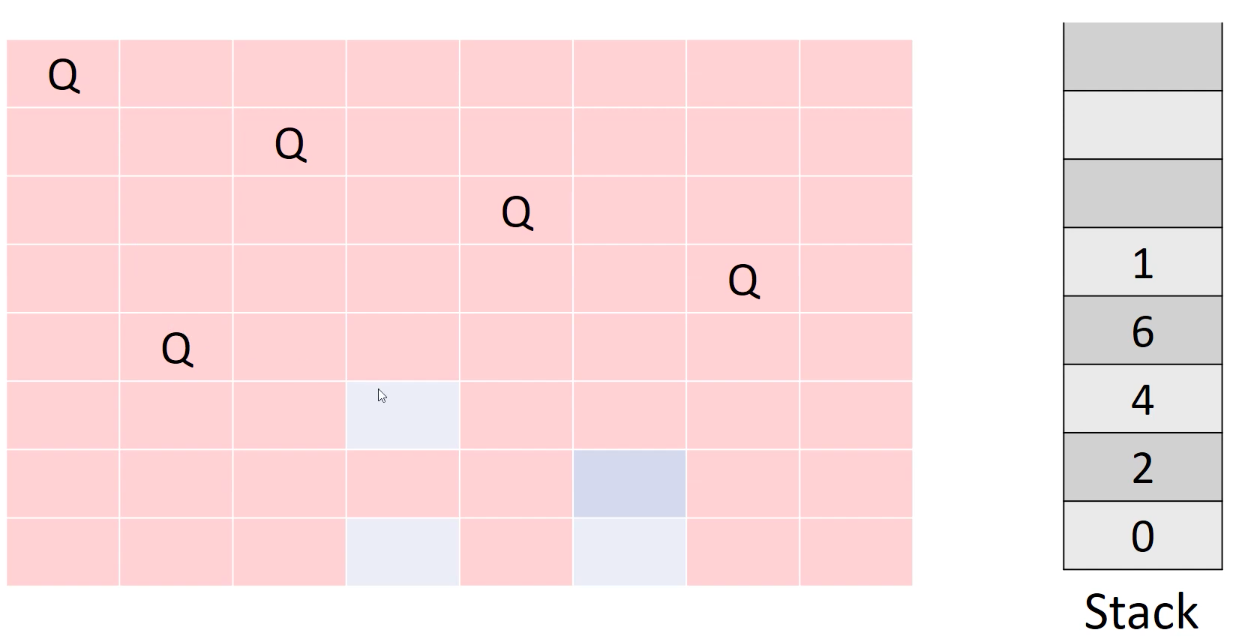
但是，把第五個皇后放到第八個直行上後，發現第六個橫列的皇后仍然沒有位置放，所以必須再次回退。



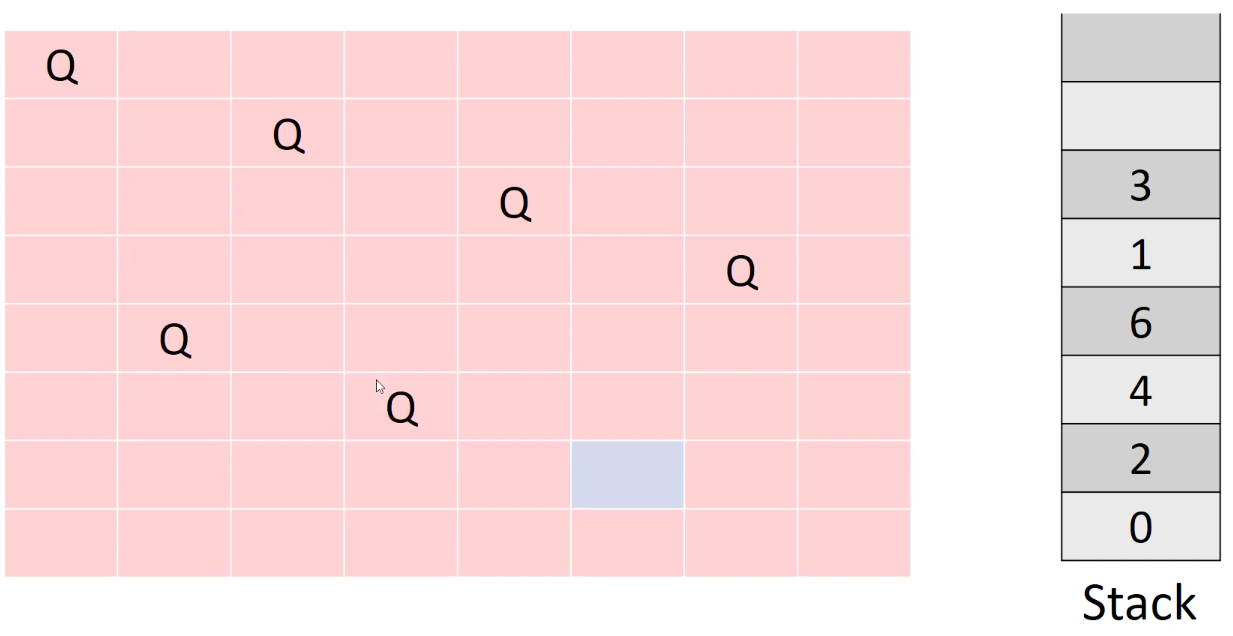
這時，發現如果前四個皇后在目前的位置上，不論第五個皇后放在哪裡，都不能產生可行解，因此要回退到第四個皇后還未被放到棋盤上的時候，選擇其他位置來擺放。



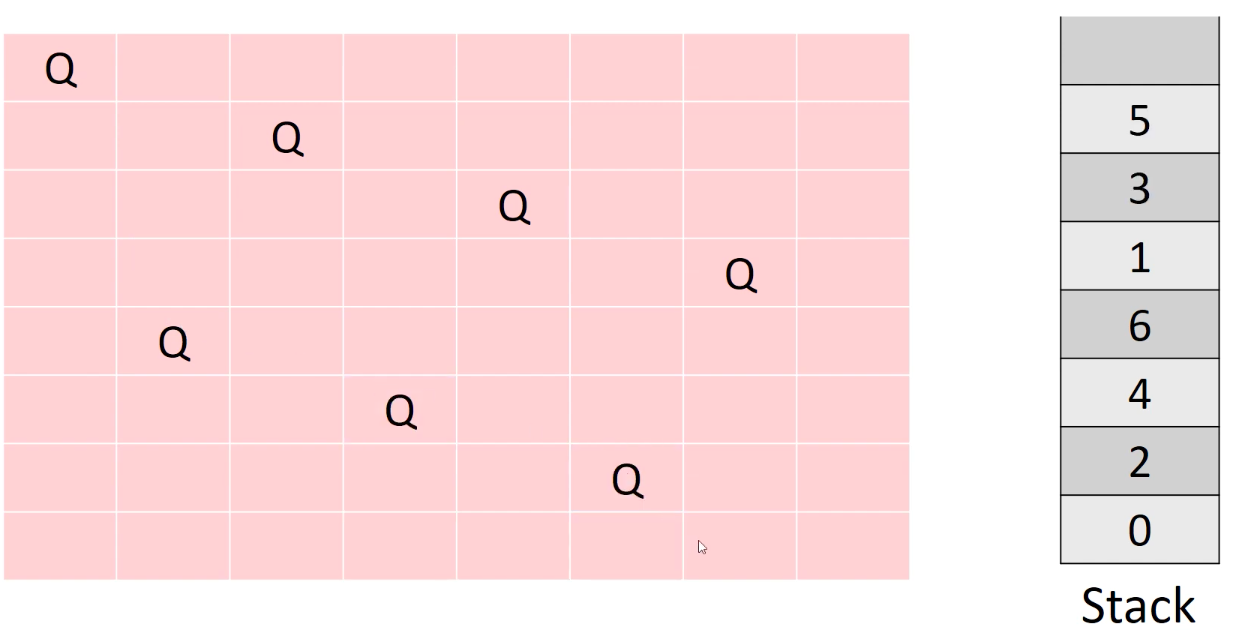
剛才已經發現第四個皇后擺在第二個直行無法往下產生可行解，所以要改將其放在其他直行，比如放在第七個直行上。



接下來，第五個皇后可以放在第二個直行。



第六個皇后可以放在第四個直行。



此時第七個皇后只能放在第六個直行上。但是一旦放上第七個皇后之後，第八個皇后又沒有可以擺放的位置了，因此又要進行回退，同樣從堆疊上方開始把皇后移除，直到某個有其他可以選擇的擺放位置時，採取該選擇往下做看看。

（3）用 DFS 解決 N 皇后問題

讓使用者輸入 N，輸出 N 皇后問題的所有解答。

|  |  |
| --- | --- |
| 用 DFS 解決 N 皇后問題 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128 | #include <iostream>  #include <stack>  using namespace std;  // 印出 stack 內容的函式  void print\_stack(stack<int>& s){  if (s.empty()){return ;}  int col = s.top();  s.pop();  print\_stack(s);  cout << col+1 << " ";  s.push(col);  }  // 檢查某種放法是否合法的函式  // 先看完 KQueens 函式內容再看本函式內容  // 因為不要連帶改動 KQueens 函式中 s 的內容，不使用參考符 &  bool available(stack<int> s, int row, int col){  // 例外處理  // s 為空的時候一定不會衝突  if (s.empty()){return true};  /\*  A. s 中是現在嘗試的橫列「之前」所有皇后的擺放位置的擺放位置  B. row 是嘗試擺放的新皇后在「第幾橫列」  C. col 是新皇后要放在「第幾直行」  \*/  /\*  確定最新橫列上皇后放的位置是否與先前皇后衝突  A. 假設要放新皇后到第五橫列上  B. 先檢查與第四橫列的皇后是否衝突  C. 沒有衝突的話，遞迴呼叫函式，檢查與第三橫列皇后是否衝突  D. 也沒有衝突的話，遞迴檢查與第二、第一橫列皇后是否衝突  E. 直到邊界條件「s 中沒有資料」，代表和先前的皇后都沒有衝突  \*/  // 取出先前的一個皇后的位置 (s.size(), col\_prev\_queen)  // 檢查與新皇后的位置 (row,col) 是否衝突  int col\_prev\_queen = s.top();  s.pop();  // 先假設不會衝突，可以放該位置  bool result = true;  // 新皇后與先前的皇后位置比較  // 相差 diff\_r 個橫列、diff\_c 個直行  int diff\_r = row - s.size();  int diff\_c = col - col\_prev\_queen;    // 直行相同時代表位置衝突，回傳 false（not available）  if (col==col\_prev\_queen)  result = false;  // 新皇后在先前皇后右下角對角線上時  // 回傳 false  if (diff\_c==diff\_r)  result = false;  // 新皇后在先前皇后左下角對角線上時  // 回傳 false  if (diff\_c==-1\*diff\_r)  result = false;    // 如果與現在比對的先前皇后沒有衝突  if (result==true){  // 繼續跟再上一個直行的皇后比對  // 注意上面執行過 s.pop()  // 因此遞迴傳入的 s 中，最上面是「再上一個直行」的皇后  result = available(s,row,col);  }  return result;  }  int KQueens(stack<int>& s, int N){  // 如果目前堆疊已經有 N 個元素  // 代表找到一組可行解  if (s.size()==N){  // 印出目前這組可行解  print\_stack(s);  cout << endl;  return 1;  }  // 堆疊還未放滿時，往下執行  // 計算在不更動目前已經擺放的皇后下  // 總共有幾組可行解  int counts = 0;    // 檢查下一個橫列的某種放法是否可行  for (int i=0 ; i<N ; i++){  // s 是已放的前幾個皇后的位置  // s.size() 和 i 是嘗試擺放下一個皇后的「橫列」和「直行」  if (available(s,s.size(),i)){  // 某個擺放位置 i 可行時，把新的皇后放到該位置上  s.push(i);  // 放了新的皇后之後，往下繼續放下一橫列的皇后  counts += KQueens(s,N);  // 下一個橫列可能有好幾個可以放  // 因此要 pop 掉，接續試下一個可行位置  s.pop();  }  }  return counts;  }  int main(){  int N;  cout << "Please enter N:" << endl;  cin >> N;  // 儲存皇后位置的 stack  stack<int> Positions;  cout << "Found a total of " << KQueens(Positions, N) << " solutions for "  << N << "-Queen problem." << endl;  return 0;  } |
| 執行結果 | |
| Please enter N:  >> 8  ...  8 3 1 6 2 5 7 4  8 4 1 3 6 2 7 5  Found a total of 92 solutions for 8-Queen problem. | |

可以得到八皇后問題共有 92 種解。

（4）LeetCode #51. N 皇后問題 N-Queens

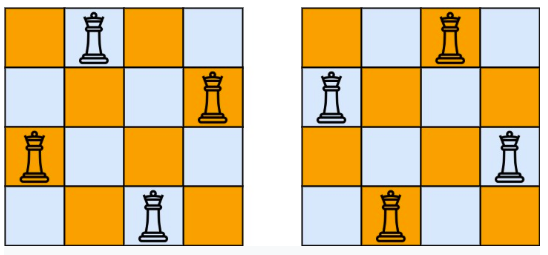
A. 題目

「N 皇后問題」指的是要在 的西洋棋盤上放置 N 個皇后，使得皇后間不會互相攻擊。給定一個整數 ，以任意順序回傳「N 皇后問題」所有可行的解。

回傳時，需以 'Q' 和 '.'（分別代表擺放皇后的位置和空位）來表示整個棋盤上的分佈。

B. 出處

https://leetcode.com/problems/n-queens/

C. 範例輸入與輸出

輸入：n = 4

輸出：

[[".Q..","...Q","Q...","..Q."],

["..Q.","Q...","...Q",".Q.."]]

四皇后問題只有兩種不同的解。

D. 解題邏輯

本題解法與剛才實作的「N 皇后問題」解法相同，只需調整輸出格式。

|  |  |
| --- | --- |
| 用 DFS 解決 N 皇后問題 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98 | class Solution{  // 檢查新皇后位置是否合法的函式  bool available(stack<int> s, int row, int col){    if (s.empty()){return true;}  int col\_prev\_queen = s.top();  s.pop();  bool result = true;  int diff\_c = col - col\_prev\_queen;  int diff\_r = row - s.size();  if (col==col\_prev\_queen)  result = false;  if (diff\_c==diff\_r)  result = false;  if (diff\_c==-1\*diff\_r)  result = false;  if (result){  result = available(s,row,col);  }  return result;  }  // 生成一個可行解用 'Q' 和 '.' 的表示方式  vector<string> add\_solution(stack<int>& s, int n){  if (s.empty()){return {};}    // 從可行解 s 中取出一個橫列的皇后位置  // 比如「1」  int col = s.top();  s.pop();  // 遞迴得到前面所有橫列產生的向量 this\_solution  // 比如得到 ["..Q.","Q...","...Q"]  vector<string> this\_solution = add\_solution(s,n);  // 產生目前橫列的表達方式  string result\_str;  // 除了要放的位置 col 外，用 '.' 補滿  // 比如得到 ".Q.."  for (int i=0 ; i<col ; i++){  result\_str += '.';  }  result\_str += 'Q';  for (int i=col+1 ; i<n ; i++){  result\_str += '.';  }  s.push(col);  // 在 this\_solution 中加上目前橫列的內容  // 比如 ["..Q.","Q...","...Q"].push(".Q..")  // = ["..Q.","Q...","...Q",".Q.."]  this\_solution.push\_back(result\_str);  return this\_solution;  }  int KQueens(stack<int>& s, int N){    if (s.size()==N){  // 找到一個可行解  // 透過 add\_solution 產生對應的向量後加到 result 中  vector<string> this\_solution = add\_solution(s,N);    result.push\_back(this\_solution);  }  int counts = 0;  for (int i=0 ; i<N ; i++){  if (available(s,s.size(),i)){  s.push(i);  counts += KQueens(s,N);  s.pop();  }  }  return counts;  }  // 儲存目前皇后位置的堆疊  stack<int> Positions;  // 回傳的結果  vector<vector<string>> result;  public:  vector<vector<string>> solveNQueens(int n){  KQueens(Positions, n);  return result;  } // end of solveNQueens  }; // end of Solution |

5. 實戰練習

（1）LeetCode #695. 最大島嶼的面積 Max Area of Island

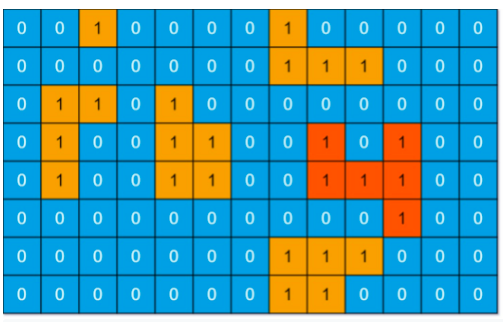
A. 題目

給定一個 的二元陣列 ，一個「島嶼」是一些上下左右連接在一起的「1」，假設 的邊界外全部都是代表水的「0」。

一個島嶼的面積是組成整個島嶼的「1」的個數，回傳整個 中最大島嶼的面積，如果沒有任何島嶼，回傳 0。

B. 出處

https://leetcode.com/problems/max-area-of-island/

C. 範例輸入與輸出

輸入：grid = [

[0,0,1,0,0,0,0,1,0,0,0,0,0],

[0,0,0,0,0,0,0,1,1,1,0,0,0],

[0,1,1,0,1,0,0,0,0,0,0,0,0],

[0,1,0,0,1,1,0,0,1,0,1,0,0],

[0,1,0,0,1,1,0,0,1,1,1,0,0],

[0,0,0,0,0,0,0,0,0,0,1,0,0],

[0,0,0,0,0,0,0,1,1,1,0,0,0],

[0,0,0,0,0,0,0,1,1,0,0,0,0]

]

輸出：6

D. 解題邏輯

本題同樣可以從左上角開始一列一列進行，每遇到一個 1 就進行 DFS，得到每個島嶼的面積，並把每個經過的 1 都改成 -1，避免重複計算。

|  |  |
| --- | --- |
| 最大島嶼的面積 Max Area of Island | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60 | class Solution{    // DFS  int area(vector<vector<int>>& grid, int i, int j, int row, int col){  // 過程中超出邊界時，回傳 0  if (i<0 || j<0){  return 0;  }  if (i>=row || j>=col){  return 0;  }  // 遇到海洋或已經走過的陸地時，回傳 0  if (grid[i][j]==0 || grid[i][j]==-1){  return 0;  }  // 把現在造訪的陸地值改為 -1  grid[i][j] = -1;  // 往四個方向移動  int up = area(grid, i-1, j, row, col);  int down = area(grid, i+1, j, row, col);  int left = area(grid, i, j-1, row, col);  int right = area(grid, i, j+1, row, col);  // 回傳的面積是目前方格的面積 1  // 加上往四個方向移動後繼續處理所得面積  return up + down + left + right + 1;  }  public:  int maxAreaOfIsland(vector<vector<int>>& grid){  // 取出 grid 的大小  int row = grid.size();  int col = grid[0].size();  // 最大島嶼面積  int max\_area = 0;  for (int i=0 ; i<row ; i++){  for (int j=0 ; j<col ; j++){    // 遇到陸地 1 時就進行 DFS  if (grid[i][j]==1){  int tmp = area(grid,i,j,row,col);  max\_area = max\_area>tmp?max\_area:tmp;  }    }  }  return max\_area;  } // end of maxAreaOfIsland  }; // end of Solution |

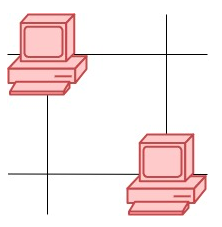
（2）LeetCode #1267. 溝通伺服器的總數 Count Servers that Communicate

A. 題目

給定一個 的整數陣列，代表了一個伺服器中心的地圖，陣列中每個 1 代表該處有伺服器，0 代表該處沒有伺服器。兩個伺服器只有位在相同直行或相同橫列時，被認為會互相「溝通」。回傳會和其他伺服器溝通的伺服器總數。

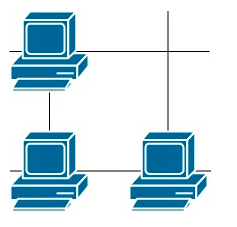
B. 出處

https://leetcode.com/problems/count-servers-that-communicate/

C. 範例輸入與輸出

輸入：grid = [[1,0],[0,1]]

輸出：0

因為兩個伺服器不在同一直行或橫列上，沒有互相溝通的伺服器。

輸入：grid = [[1,0],[1,1]] （grid 中三個 1 就代表三個伺服器的位置）

輸出：3

D. 解題邏輯

本題同樣可以在遇到每個伺服器時，進行 DFS，造訪過的伺服器位置改填 -1。

|  |  |
| --- | --- |
| 溝通伺服器的總數 Count Servers that Communicate | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61 | class Solution{  int connected(vector<vector<int>>& grid, int i, int j, int row, int col){  // 不處理的情形  if (grid[i][j]==-1){  return 0;  }  // 把目前伺服器標成 -1 代表已處理過  grid[i][j] = -1;  int row\_sum = 0, col\_sum = 0;  // 檢查同橫列的伺服器  for (int m=0 ; m<col ; m++){  if (grid[i][m]==1){  row\_sum += connected(grid,i,m,row,col);  }  }  // 檢查同直行的伺服器  for (int n=0 ; n<row ; n++){  if (grid[n][j]==1){  col\_sum += connected(grid,n,j,row,col);  }  }  return row\_sum + col\_sum + 1;  }  public:  int countServers(vector<vector<int>>& grid){  // 取得 grid 的大小  int row = grid.size();  int col = grid[0].size();  // 記錄會與其他伺服器溝通的伺服器數量  int counts = 0;  for (int i=0 ; i<row ; i++){  for (int j=0 ; j<col ; j++){  if (grid[i][j]==1){  // tmp 會從現在這台伺服器一路向外進行 DFS  // 得到過程中共有幾台有在溝通的伺服器  int tmp = connected(grid, i, j, row, col);  // tmp = 1 時，目前的伺服器沒有向外溝通  // 有溝通時，counts 加上 DFS 找到的伺服器數  if (tmp > 1){  counts += tmp;  }  }  } // end of inner for  } // end of outer for  return counts;  } // end of countServers  }; // end of Solution |