Ch10. 廣度優先搜尋 Breadth-First Search

本章介紹廣度優先搜尋 BFS。

首先介紹什麼叫「圖的搜尋」，以及圖的搜尋有什麼特別需要注意的地方。

再來，會開始介紹廣度優先搜尋的常見應用：

A. 計算連通元件的數目
B. 窮舉所有情形
C. 尋找最短路徑
D. 確認圖中是否有環

最後，會再做幾題實戰練習。

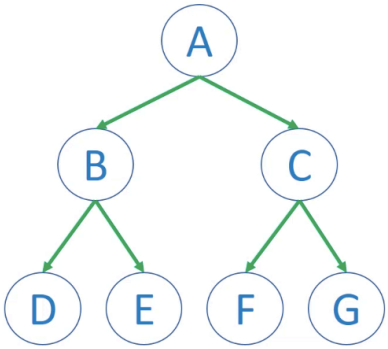
1. 圖的搜尋

圖的問題很多都由下列三種問題演變而來：

A. 兩頂點間是否存在路徑：從其中一個頂點出發是否能夠到達另一個頂點，比如從台北出發，可以「開車」到台中，但是不能「開車」到東京
B. 尋找兩頂點間的最短路徑
C. 盡可能把所有的頂點或路徑走過一次：比如要知道「哪個城市有賣漢堡」，必須把所有的城市（頂點）都造訪過一次才能得到答案

資料結構的課程中，會提到二元樹的走訪有「前序」、「中序」、「後序」三種方式，之所以可以用這些方式走訪，是因為二元樹中沒有「環」，否則就不能用這些方式進行。

- 前序 (Pre-order)
  - ✓ ABDECFG
  - ✓ 每經過一個新節點就先處理該節點
- 中序 (In-order)
  - ✓ DBEAFCG
  - ✓ 左子樹的資料都處理完再處理該節點
- 後序 (Post-order)
  - ✓ DEBFGCA
  - ✓ 左右子樹的資料都處理完再處理該節點



紅字為根節點

然而，在可能「有環」的圖中，仍然需要按照某種固定的行為準則來造訪所有頂點，每到一個頂點後，就按照此規則決定接下來要造訪哪個點。

一個理想的行為準則，可以避免因為圖中有環而陷入無窮迴圈，常見的兩種造訪方式就是「廣度優先搜尋」和「深度優先搜尋」。

（1）「廣度優先搜尋 BFS」和「深度優先搜尋 DFS」

這兩種搜尋方式決定了應該先求「廣」還是先求「深」。

A. 安排讀書日程

比如若有七個科目要考試，可以決定每個科目都先只念一天，隔天就念另外一科，直到七個科目國文、英文、數學、...都各念過一天之後，再從第一科開始念每科的第二天。第一輪時每一科只讀一點，下一輪又輪到時，再往下讀一點，這屬於「廣度優先讀書法」。

「深度優先讀書法」指的則是把一科從頭到尾準備完後，再開始準備下一科。

也就是說，廣度優先是對每個選擇 / 路徑都淺嘗輒止，先找到許多選擇、路徑後輪流向下執行；深度優先則是把一個路徑「走到底」，直到無法再往下走時，才考慮下一個路徑。

B. 找到喜歡的工作

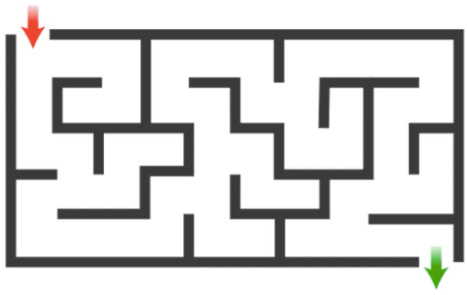
另一個例子是「如何找到喜歡的工作」：廣度優先的方法是每種工作先打工打個三天，三天一到就換下一個工作，直到選項中每個工作都做過三天之後，如果仍然無法決定，再從第一個選項開始新的一輪，繼續每個工作都再打工三天，直到能夠決定哪個工作自己最喜歡。

深度優先的方法則是把選項中的一個工作一直往下做，直到真的無法再做下去，確定這個工作一定不是自己喜歡的，才去嘗試下一個工作。

C. 空間複雜度的比較

因為「廣度優先搜尋」要記錄下每個選擇（路徑）各自進行到哪裡，所以會較耗費記憶體空間，「深度優先搜尋」則比較節省空間，不過這兩種搜尋方式各自有較適用的場合，之後會一一介紹。

（2）廣度優先與深度優先搜尋的走迷宮策略



上圖中的迷宮起點在左上角，終點則在右下角，目標是要找到一條可行的路從起點通到終點，此時廣度優先搜尋和深度優先搜尋進行的過程就會有所不同。

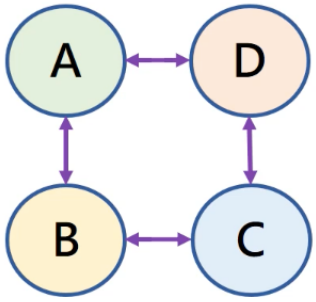
A. 廣度優先搜尋 Breadth-First Search（BFS）
   a. 先處理完所有相鄰節點後，再往下處理
   b. 每條路都淺嘗輒止
   c. 通常使用佇列 Queue 來達成

B. 深度優先搜尋 Depth-First Search（DFS）
   a. 先把相鄰節點之一一路往下處理完之後，再針對下一個相鄰節點處理

b. 每條路都走到底，不行再試下一條
c. 通常使用堆疊 Stack 來達成

為了避免因為圖中有環而導致無窮迴圈或記憶體的浪費，要找到能夠辨識環的方式。在童話故事中，一對兄妹在森林裡，為了避免迷路，就在往前走的過程中一路「撒麵包屑」，達到「凡走過必留下痕跡」的效果。這樣一來，之後只要看到麵包屑，就知道「這條路已經走過了」，避免陷入重複嘗試某一條路。

同樣的，每造訪圖中的一個頂點，就要對該頂點做標示，這樣一來之後執行時就可以得知該點是否已經走過。



在演算法的領域中，通常會把頂點分成三種

A. 白色：尚未尋訪、尚未處理
B. 灰色：已尋訪過，但尚未處理
C. 黑色：已尋訪過且已經處理

遇到不同顏色的節點時，會進行不同的動作

A. 走到白色節點：發現新大陸，可以進行處理
B. 走到灰色節點：忽略（BFS）、確定有環（DFS）
C. 走到黑色節點：可以結束該次尋訪

在走一個迷宮時，廣度優先搜尋會從出發點開始，找到所有可以移動的方向，接下來，會把這些方向記錄下來，例如（A1、B1、C1、...、Z1）。

沿著每個方向都往前走到一個路口（有超過一種往下走的可能性時），比如沿著 A1 路徑往前走到 T 字路口，有往左和往右的兩種可能走法（A2 和 B2），一旦到路口，就把繼續往下走的數個方向同樣記錄下來（清單中變為 A1、B1、C1、...、Z1、A2、B2），但是並不直接執行 A2、B2，而是按照原本清單中的順

序，從 B1 開始往下走到路口、新增路徑到清單後端、C1 開始往下走到路口、新增路徑，依此類推，直到完全無路可走，或者已經到達終點為止。

因為一路上需要同時記錄很多種可能，所以廣度優先搜尋比較耗費儲存空間。

2. 廣度優先搜尋的實作

在進入廣度優先搜尋的實作前，先看一下會用到的資料結構：佇列 Queue。

（1）佇列 Queue

佇列像是一個陣列 array，但是有固定的新增和刪除方向，插入資料和刪除資料的方向在「異側」，這又叫做 first-in-first-out（FIFO），最先進入結構的資料會最先出來。

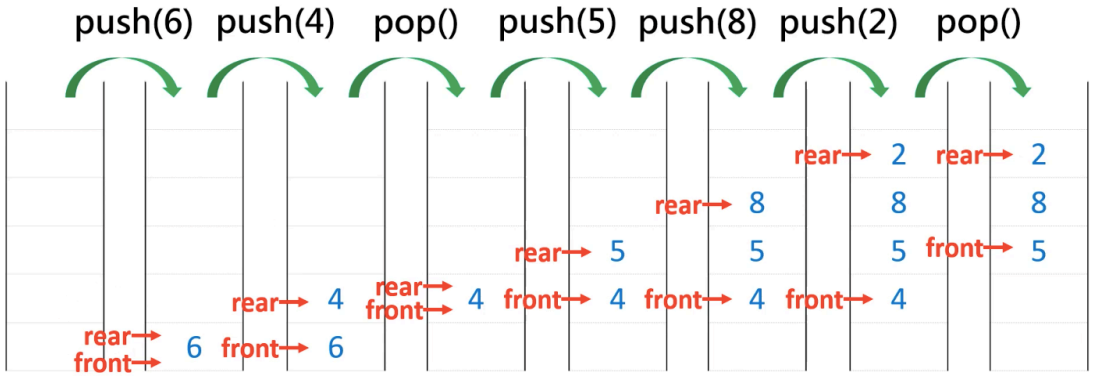
這樣的運作模式就像在電影院排隊入場時，新來的人要從後面（隊伍尾端）開始排，而入場是從前面（隊伍的開頭）一個個入場，假如新增資料在右邊，刪除資料則從左邊，兩種操作在「異側」進行的，就叫做佇列。

佇列的資料有「兩端」，所以可以回傳開頭的資料，也可以回傳結尾的資料：front 回傳「前端 / 刪除端」的資料，rear 則回傳「末端 / 新增端」的資料。

佇列 Queue 常用操作	
push	新增一筆資料
pop	刪除一筆資料
front	回傳前端（刪除端）的資料
rear	回傳末端（新增端）的資料
empty	確認 queue 中是否有資料
size	回傳 queue 中的資料個數

下面的例子中，push 是從「上面」往佇列中新增資料，pop 是從「下面」開始刪除資料。因為新增和刪除在異側，有兩個指標 front 和 rear 分別指向兩端的資料。

push(6) 的時候，把 6 從上面丟進去，push(4) 的時候，同樣是從上面丟，因此 4 會在 6 的上面。pop() 是從下面刪除一筆資料，因此 4 和 6 兩者中較下方的 6 被刪除。



操作	front	rear	說明
push(6)	6	6	front 和 rear 都指向唯一的一筆資料 6
push(4)	6	4	rear 指向新加入的資料 4 front 指向接下來若進行刪除會刪掉的資料 6
pop()	4	4	從下方把 6 刪掉，rear 和 front 都指向 4
push(5)	4	5	從上方新增 5
push(8)	4	8	從上方新增 8
push(2)	4	2	從上方新增 2
pop()	5	2	從下方把 4 刪掉

（2）佇列 Queue 的用途

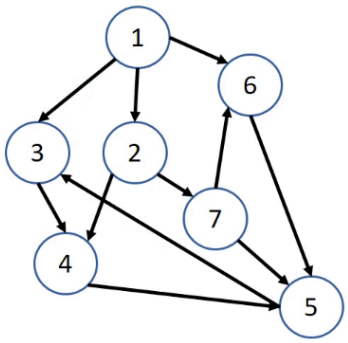
A. 依序處理先前的資訊
   a. 常用來做資料的緩衝區
   b. 記憶體（標準輸出、檔案寫入）
   c. 印表機輸出
   d. CPU 的工作排程
B. 迷宮探索、搜尋：廣度優先搜尋（BFS）
C. 無法得知 Queue 裡有哪些資料：與堆疊相同，只能以 pop 依序拿出資料

通常佇列 Queue 會被用來依序處理資訊和做工作排程。

假設今天教室中的二十台電腦共用一台印表機，如果這些電腦在很短的時間內都要求列印，印表機應該如何決定先印誰的文件呢？比較公平的做法應該是哪台電腦的文件先傳送到印表機，就先把這台電腦的文件印出，所以印表機當中就有類似佇列的資料結構，它每接收到一個要列印的文件，就將文件放到佇列中，可以不停接收，而真正進行列印時，則從另一端取出一個一個文件依序印出，先到者就先印出來。

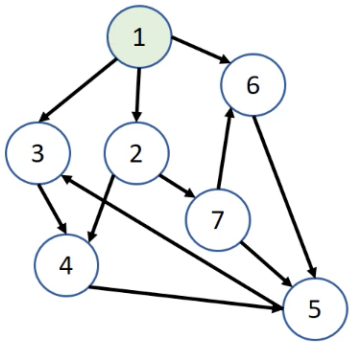
想得知位在 Queue 中間的某筆資料內容為何，不能跳過前面的資料，必須使用 pop 依序將前面的資料全部取出，直到目標資料在最前端時才能得知。

（3）廣度優先搜尋的進行過程



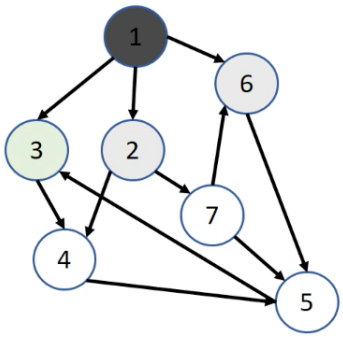
廣度優先搜尋的規則是先把所有的「鄰居」都造訪過之後，再繼續處理其他資料，也就是說，每個可能的路徑都只往下走一小步，下一輪時才會再往下又走一小步。

上圖當中，一開始所有節點都是未經處理也未經尋訪的「白色」。先任選一個起點 1，把 1 放到 Queue 中。



1																			

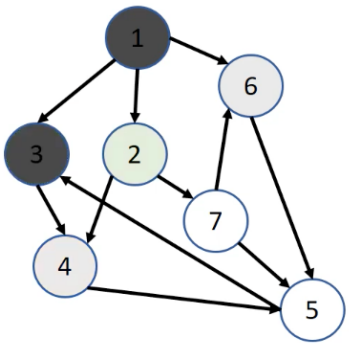
接下來，節點 1 有 3 個相鄰節點 3、2、6，這三個相鄰節點都被放到 Queue 中，此時 1 已經處理完，因此從 Queue 中將 1 移除（Queue 的圖中保留 1 只是為了閱讀方便，實際上已經移除），並且圖中節點 1 被塗上已處理完的「黑色」，3、2、6 則是「灰色」。



1	3	2	6																

此時 Queue 中最前面的資料是 3，因此針對 3 進行處理，3 的相鄰節點是 4，4 被放到 Queue 中。

此時，3 已經被處理完，因此從 Queue 中移除，並把節點 3 塗上「黑色」、節點 4 塗上「灰色」。



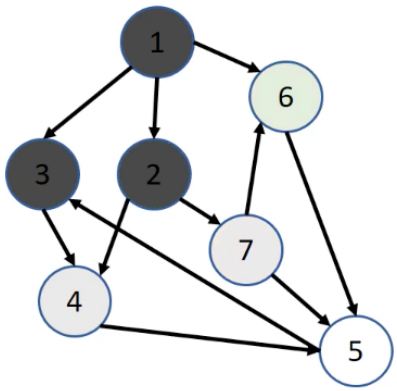
1	3	2	6	4															

就像這樣，每一輪的步驟是

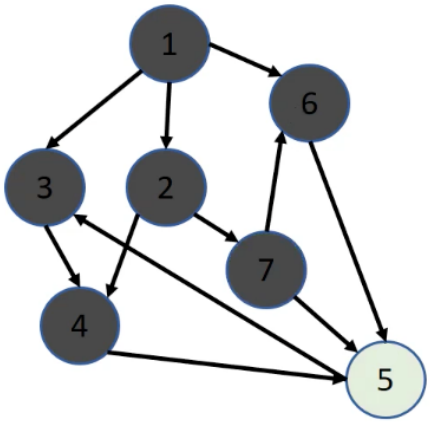
A. 處理 Queue 中最前面的資料
B. 把該節點的相鄰節點（當中白色者）放入 Queue
C. 把 Queue 中最前面的節點從 Queue 中移除、圖中該節點塗成黑色
D. 把剛才放入 Queue 的節點塗成灰色

接下來 Queue 中最前面的資料是 2，它的相鄰節點是 4 和 7，但是 4 已經被塗成灰色，因此只需把白色的 7 放入 Queue 中即可。

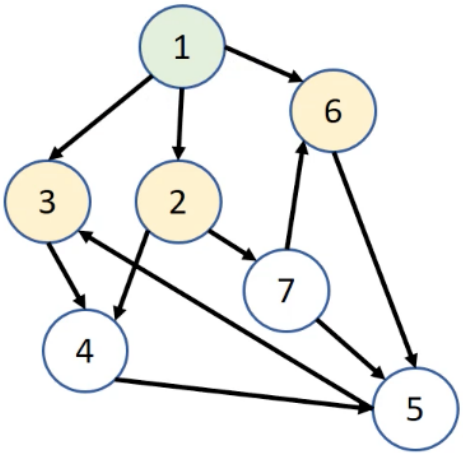
接下來，把 2 從 Queue 中移除，節點 2 塗成黑色、節點 7 塗成灰色。



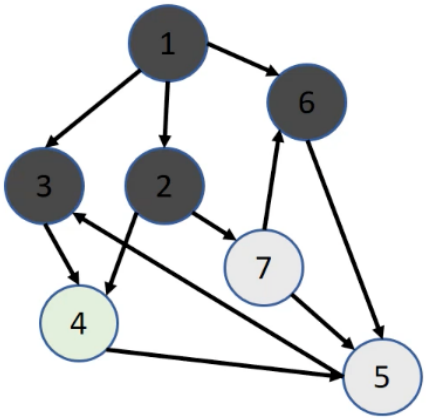
依此類推，直到 Queue 只剩下節點 5，因為節點 5 沒有相鄰節點，所以結束處理。



廣度優先搜尋的進行過程像是從起點開始「一層層擴散出去」，一開始從起點 1 擴散出第一層：



接下來，從第一層的 3、2、6 又向外擴散出一層：



一層一層擴散出去就是廣度優先搜尋的特性。

從 Queue 的角度來看，廣度優先搜尋的結束條件是「Queue 中的資料全部處理完畢」，而圖上節點的顏色意義則是：

A. 白色：沒有被放入 Queue 中過
B. 灰色：已經被放入 Queue 中，還未處理到
C. 黑色：已經被放入 Queue 中過，並且已經處理而被移除

（4）廣度優先搜尋的複雜度

廣度優先搜尋的虛擬碼	
1	BFS(G,s)
2	
3	// 把所有節點塗成白色
4	for each vertex(v) in G:
5	color[v] = white
6	
7	// 一開始 queue 中只有起點
8	path_queue = {s}
9	color[s] = gray
10	
11	// queue 中還有資料時要繼續執行
12	while path_queue.size() != 0:
13	
14	// 從 queue 中取出最前面的節點
15	vertex_now = path_queue.pop()
16	
17	// 處理每個相鄰節點，一個「相鄰」關係對應一條邊
18	for each vertex in vertex_now.adjacent():
19	// 只針對其中白色的節點處理
20	if color[vertex]==white:
21	// 相鄰節點塗成灰色
22	color[vertex] = gray
23	// 相鄰節點被加入 queue 中
24	path_queue.push(vertex)
25	color[vertex_now] = black

初始化的過程把所有頂點塗成白色，需要 $O(|V|)$，接下來，每條邊都會在 for 迴圈檢查相鄰節點時被處理，需要 $O(|E|)$，因此整個過程的複雜度為 $O(|V| + |E|)$。

（5）廣度優先搜尋的實作

這裡示範用前面建立的「無權重的鄰接列表」來實作廣度優先搜尋，修改鄰接列表如下：

修改鄰接列表

```
1  #include <iostream>
2  #include <stdlib.h>
3  #include <vector>
4  #include <list>
5  using namespace std;
6
7  class Graph{
8  private:
9        int vertex;
10       // 無權重時只需記錄邊的終點
11       vector<list<int>> edges;
12 public:
13       Graph(int);
14       void print_edges();
15       bool add_edge(int, int);
16 };
17
18 // 建構式
19 Graph::Graph(int v){
20       vertex = v;
21       edges.resize(vertex);
22 }
23
24 void Graph::print_edge(){
25       for (int i=0 ; i<vertex ; i++){
26             cout << i+1 << "\t";
27             auto iter = edges[i].begin();
28             for( ; iter!=edges[i].end() ; iter++){
```

```
29                      cout << "->" << (*iter)+1;
30              }
31              cout << endl;
32      }
33  }
34
35  bool Graph::add_edge(int from, int to){
36      edges[from-1].push_back(to-1);
37  }
38
39  int main(){
40
41      Graph g(7);
42      g.add_edge(1,3);
43      g.add_edge(1,2);
44      g.add_edge(1,6);
45      g.add_edge(2,4);
46      g.add_edge(2,7);
47      g.add_edge(3,4);
48      g.add_edge(4,5);
49      g.add_edge(6,5);
50      g.add_edge(7,6);
51      g.add_edge(7,5);
52
53      return 0;
54  }
```

執行結果

```
1   ->3->2->6
2   ->4->7
3   ->4
4   ->5
5
6   ->5
7   ->6->5
```

擴寫上面的程式碼，實作廣度優先搜尋函式：

廣度優先搜尋

```
1  #include <iostream>
2  #include <stdlib.h>
3  #include <vector>
4  #include <list>
5  using namespace std;
6
7  class Graph{
8  private:
9        int vertex;
10       // 無權重時只需記錄邊的終點
11       vector<list<int>> edges;
12 public:
13       Graph(int);
14       void print_edges();
15       bool add_edge(int, int);
16       // 廣度優先搜尋，指定起點 int
17       void BFS(int);
18 };
19
20 // 建構式
21 Graph::Graph(int v){…}
22
23 void Graph::print_edge(){…}
24
25 bool Graph::add_edge(int from, int to){…}
26
27 void Graph::BFS(int start){
28
29       // 起點 start 對應到索引值 start-1
30       start--;
```

```
31
32      // 自己指定顏色與 int 的對應
33      // 0：白色 white
34      // 1：灰色 gray
35      // 2：黑色 black
36
37      // 把所有頂點塗成白色
38      vector<int> color(vertex,0);
39      // BFS 使用的佇列
40      queue<int> BFS_Q;
41
42      // 把起點放進 Queue 中並塗成灰色
43      BFS_Q.push(start);
44      color[start] = 1;
45
46      // 印出頂點
47      cout << start+1 << "->";
48
49      // 當 Queue 中還有資料
50      while(!BFS_Q.empty()){
51          // 取出 Queue 中最前面的資料
52          int current = BFS_Q.front();
53          BFS_Q.pop();
54
55          // 針對 current 的相鄰節點處理
56          for (auto iter=edges[current].begin() ; iter!=edges[current].end() ;
57          iter++){
58
59              // 只處理相鄰頂點中白色者
60              if (color[*iter]==0){
61
62                  // 印出目前處理的節點
63                  cout << iter+1 << "->";
64
```

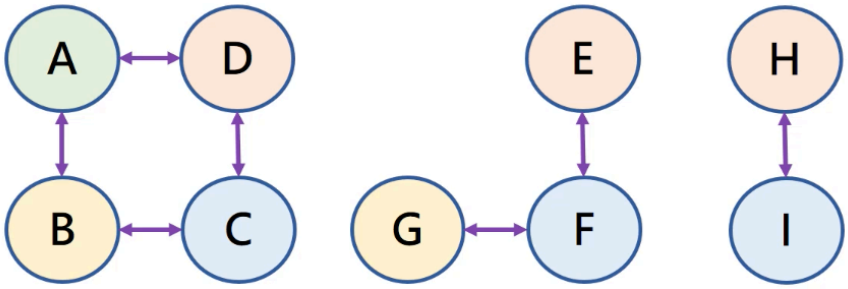
```
65                          // 放入 Queue 中
66                          BFS_Q.push(*iter);
67
68                          // 塗成灰色
69                          color[*iter] = 1;
70
71                      }
72
73              } // end of for
74
75              // 把 current 塗成黑色
76              color[current] = 2;
77
78          } // end of while
79
80  } // end of BFS
81
82  int main(){
83          Graph g(7);
84          g.add_edge(1,3);
85          g.add_edge(1,2);
86          g.add_edge(1,6);
87          g.add_edge(2,4);
88          g.add_edge(2,7);
89          g.add_edge(3,4);
90          g.add_edge(4,5);
91          g.add_edge(6,5);
92          g.add_edge(7,6);
93          g.add_edge(7,5);
94          g.BFS(1);
95          return 0;
96  }
```

執行結果

```
1->3->2->6->4->7->5->
```

執行結果與先前講解中得到的順序相同。

（6）使用廣度優先搜尋計算連通元件的個數（無向圖）



上圖中有三個連通元件，若以 A 點做為起點進行廣度優先搜尋，A、B、C、D 四個點會被塗成黑色，但其他點仍未處理，因此要再進行第二次、第三次廣度優先搜尋，直到所有頂點都被處理過。

每執行一次廣度搜尋，就會把「一個」連通元件中的所有頂點塗成黑色，因此要跑幾次廣度優先搜尋才能把整張圖全部塗黑，就代表圖中有幾個連通元件。

更改剛才的程式碼：

計算連通元件的個數

```
1  #include <iostream>
2  #include <stdlib.h>
3  #include <vector>
4  #include <list>
5  using namespace std;
6
7  class Graph{
8  private:
9      int vertex;
10     // 無權重時只需記錄邊的終點
11     vector<list<int>> edges;
12 public:
13     Graph(int);
```

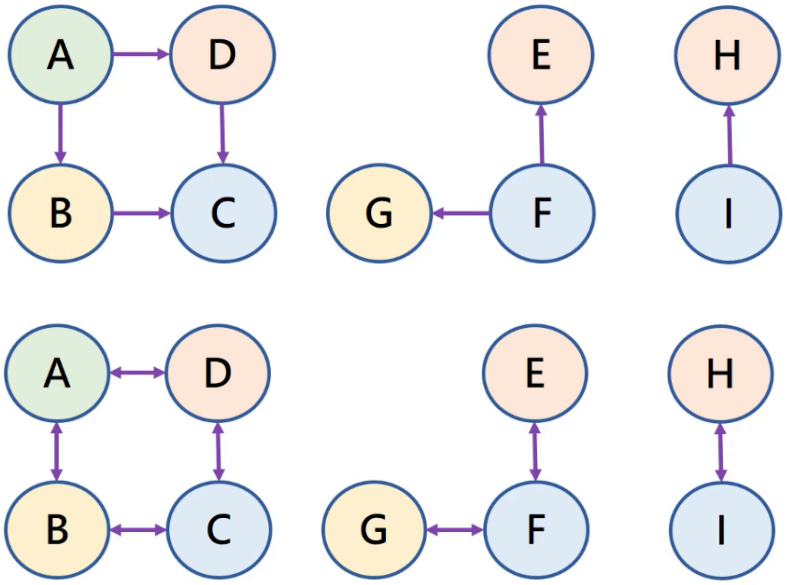
```
14        void print_edges();
15        bool add_edge(int, int);
16        // 廣度優先搜尋，指定起點 int
17        void BFS(int);
18        int connected_component();
19  };
20
21  // 建構式
22  Graph::Graph(int v){…}
23
24  void Graph::print_edge(){…}
25
26  // 改為無向圖，加入相反方向的邊
27  bool Graph::add_edge(int from, int to){
28        edges[from-1].push_back(to-1);
29        edges[to-1].push_back(from-1);
30  }
31
32  void Graph::BFS(int start){…}
33
34  int Graph::connected_component(){
35
36        // 把所有頂點塗成白色
37        vector<int> color(vertex,0);
38        // 計算連通元件的數目
39        int component = 0;
40
41        // 任選一個起點
42        for (int i=0 ; i<vertex ; i++){
43              // 如果頂點 i 經過前面的處理還是白色，就再進行一次 BFS
44              if (color[i]==0){
45
46                    // 進行一次 BFS 就代表算到一個連通元件
47                    component++;
```

```
48                  queue<int> BFS_Q;
49
50                  // 起點為 i
51                  int start = i;
52                  BFS_Q.push(start);
53                  color[start] = 1;
54
55                  while(!BFS_Q.empty()){
56                      int current = BFS_Q.front();
57                      BFS_Q.pop();
58
59                      for (auto iter=edges[current].begin() ;
60                      iter!=edges[current].end() ; iter++){
61                          if (color[*iter]==0){
62                              BFS_Q.push(*iter);
63                              color[*iter] = 1;
64                          }
65                      } // end of for
66
67                  color[current] = 2;
68              }
69      }
70      // 回傳連通元件數目
71      return component;
72  }
73
74  int main(){
75
76      Graph g(9);
77      g.add_edge(1,2);
78      g.add_edge(1,4);
79      g.add_edge(2,3);
80      g.add_edge(3,4);
81      g.add_edge(3,4);
```

```
82          g.add_edge(5,6);
83          g.add_edge(6,7);
84          g.add_edge(8,9);
85
86          // 輸出連通元件數目
87          cout << g.connected_component();
88
89          return 0;
90  }
```

執行結果

```
3
```



計算有向圖的「弱連通」元件個數時，先將其轉換為無向圖後，再用一樣的方法計算即可。

（7）LeetCode #733. 洪水填充 Flood Fill

A. 題目

一個 $m \times n$ 的整數陣列 $image$ 代表了一張圖片，其中 $image[i][j]$ 是圖片中一個像素的像素值。

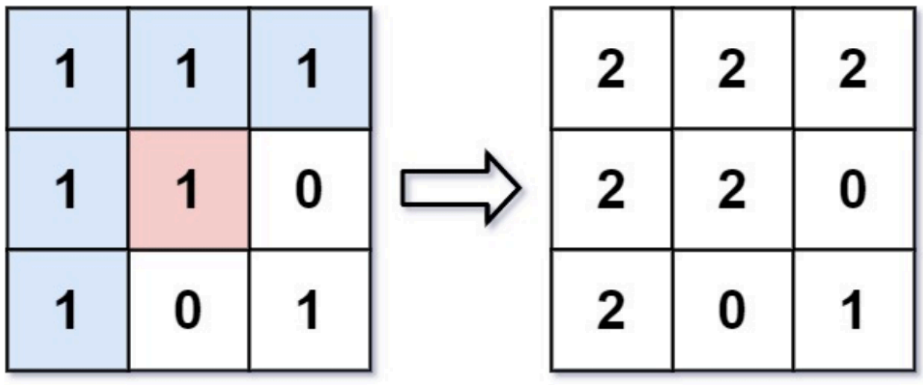
給定三個整數 sr, sc 和 $newColor$，請用「洪水填充」的方式從 $image[sr][sc]$ 開始把整張圖片塗成新的顏色。

所謂「洪水填充」，是從起點開始，把該像素的上下左右四個像素中與起點顏色相同者都一起塗成新的顏色 $newColor$，接下來，把剛才的四個像素的上下左右四個像素中顏色相同者也都塗成 $newColor$，依此類推，直到用 $newColor$ 填充完所有應被填充的像素。

完成洪水填充後，回傳修改過的 $image$。

B. 出處
https://leetcode.com/problems/flood-fill/

C. 範例輸入與輸出



輸入：image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, newColor = 2
輸出：[[2,2,2],[2,2,0],[2,0,1]]

從起點 [1][1] 開始，把上下左右四個像素中「顏色和起點同樣為 1 的上和

左」塗成 newColor = 1，接下來，因為左上、右上、左下三個像素和上一輪中被塗色的像素上下左右相連，因此也被塗成 2。

注意右下角的 1 沒有被塗成 2，因為從起點出發一路往上下左右四個方向（不含斜角）值為 1 的點擴張時，不會到達右下角。

D. 解題邏輯

本題可以用廣度優先搜尋解決。另外，雖然題目中給定的是二維陣列，但是進行 BFS 時，仍然可以使用一維的 Queue：

比如 $image[2][1]$ 是第 3 個 row，第 2 個 column。如果 cols 是一個 row 的長度（一個橫列共有幾筆資料），那麼 $image[2][1]$ 就會是從左上角數來第 $2 \times cols + 1$ 筆資料。

一般來說，$image[x][y] = x \times cols + y = P$。從位置 P（第幾筆資料）要轉換回以 x、y 表示時，$(x, y) = (P/cols, P\%cols)$。

洪水填充 Flood Fill

```
1  class Solution{
2  public:
3      vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc,
4      int newColor){
5
6          // 取得二維陣列 image 的大小
7          int rows = image.size();
8          int cols = image[0].size();
9
10         // BFS 會用到的 Queue
11         // 使用一維陣列來表示二維的資料
12         queue<int> Pixels;
13
14         // image[x][y] = x*cols + y = P
15         // (x,y) = (P/cols, P%cols)
```

```
16
17          // 在 Queue 中加入起點 (sr,sc)
18          Pixels.push(sr*cols+sc);
19          // 起點原本的顏色
20          int color_replaced = image[sr][sc];
21
22          // 例外處理：newColor 和起點原本的顏色相同
23          // 不需做任何動作
24          // 若不做此例外處理，可能產生無窮迴圈
25          if (color_replaced==newColor){
26              return image;
27          }
28
29          // 先把起點換成 new_color
30          image[sr][sc] = newColor;
31
32          // Queue 中還有資料時繼續進行
33          while(!Pixels.empty()){
34              // 取出 Queue 中最前面的資料
35              int current = Pixels.front();
36              Pixels.pop();
37
38              // 把 Queue 中資料從位置 P 的形式回復為 (x,y) 形式
39              int x = current/cols;
40              int y = current%cols;
41
42              // 檢查下方像素
43              // 未超出邊界，且與原點原本的顏色相同時才進行處理
44              if (x+1<rows && image[x+1][y]==color_replaced){
45                  // 顏色塗成 newColor
46                  image[x+1][y]=newColor;
47                  // 從這個像素的座標 (x,y) 換算出這個點的位置 P
48                  int P = (x+1)*cols + y;
49                  // 把位置 P 加到 Queue 中
```

```
50                          Pixels.push(P);
51                  }
52
53                  // 檢查上方像素
54                  if (x-1>=0 && image[x-1][y]==color_replaced){
55                          image[x-1][y]=newColor;
56                          int P = (x-1)*cols + y;
57                          Pixels.push(P);
58                  }
59
60                  // 檢查左方像素
61                  if (y-1>=0 && image[x][y-1]==color_replaced){
63                          image[x][y-1]=newColor;
63                          int P = x*cols + (y-1);
64                          Pixels.push(P);
65                  }
66
67                  // 檢查右方像素
68                  if (y+1>=0 && image[x][y+1]==color_replaced){
69                          image[x][y+1]=newColor;
70                          int P = x*cols + (y+1);
71                          Pixels.push(P);
72                  }
73
74              } // end of while
75
76              // 回傳 image
77              return image
78
79          } // end of floodFill
80  }; end of Solution
```

（8）LeetCode #200. 島嶼數目 Number of islands

A. 題目

給定一個 $n \times n$ 的二維陣列 $grid$，代表一張地圖，其中 1 代表「陸地」，0 代表「水」，回傳「島嶼的數目」。

一個島嶼被水包圍，陸地間上下左右連接起來就成為一個島嶼，假設陣列的外面四周都是水（島嶼不會延伸超出邊界）。

B. 出處

http://leetcode.com/problems/number-of-islands/

C. 輸入與範例輸出

輸入：grid = {
["1","1","1","1","0"],
["1","1","0","1","0"],
["1","1","0","0","0"],
["0","0","0","0","0"]
}
輸出：1

輸入：grid = {
["1","1","0","0","0"],
["1","1","0","0","0"],
["0","0","1","0","0"],
["0","0","0","1","1"]
}
輸出：3

島嶼數目 Number of islands	
1	class Solution{
2	public:
3	int numIslands(vector<vector<char>>& grid){

```
4           // 取得 grid 的大小
5           int rows = grid.size();
6           int cols = grid[0].size();
7
8           // BFS 使用的 Queue
9           queue<int> Pixel;
10
11          // 計算島嶼數目
12          int islands = 0;
13
14          // 對每筆資料進行處理
15          // 一旦發現一筆資料是未造訪過的陸地，就進行 BFS
16          for (int i=0 ; i<rows ; i++){
17              for (int j=0 ; j<cols ; j++){
18                  if (grid[i][j]=='1'){
19
20                      // 可以自行把 '2' 定義為「已造訪過」
21
22                      // 把 grid[i] 當作起點進行 BFS
23                      // 島嶼數量加一
24                      islands++;
25                      // 先把頂點標記為已造訪過
26                      grid[i][j] = '2';
27                      // 把起點加入 Queue 中
28                      Pixel.push(P);
29
30                      // BFS
31                      while(!Pixels.empty()){
32                          int current = Pixel.front();
33                          Pixel.pop();
34                          int x = current/cols;
35                          int y = current%cols;
36
37                          // 處理下方
```

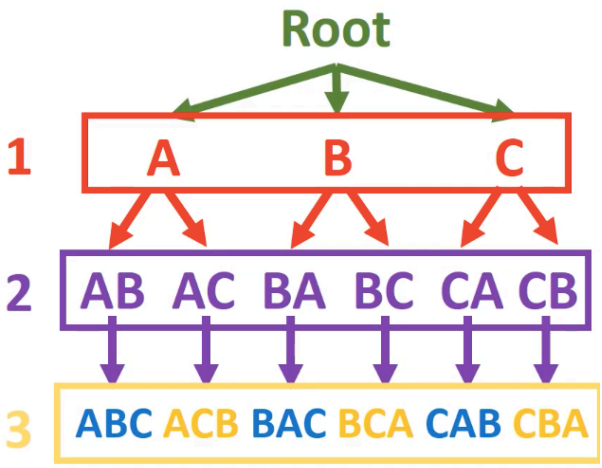
```
38                                  // 如果下方也是 1（陸地）
39                                  // 設定為 2，代表已經來過
40                                  // 並且把下方那格也加入 Queue
41                                  if (x+1<rows && grid[x+1][y]=='1'){
42                                      grid[x+1][y] = '2';
43                                      int P_now = (x+1)*cols + y;
44                                      Pixel.push(P_now);
45                                  }
46
47                                  // 處理上方
48                                  if (x-1>=0 && grid[x-1][y]=='1'){
49                                      grid[x-1][y] = '2';
50                                      int P_now = (x-1)*cols + y;
51                                      Pixel.push(P_now);
52                                  }
53
54                                  // 處理右方
55                                  if (y+1<cols && grid[x][y+1]=='1'){
56                                      grid[x][y+1] = '2';
57                                      int P_now = x*cols + (y+1);
58                                      Pixel.push(P_now);
59                                  }
60
61                                  // 處理左方
62                                  if (y-1>=0 && grid[x][y-1]=='1'){
63                                      grid[x][y-1] = '2';
64                                      int P_now = x*cols + (y-1);
65                                      Pixel.push(P_now);
66                                  }
67
68                              } // end of while
69                          } // end of if
70
71                      } // end of inner for
```

```
72              } // end of outer for
73
74              // 回傳島嶼數量
75              return islands;
76
77          } // end of numIslands
78  }; // end of Solution
```

3. 窮舉所有情形

另一個廣度優先常見的應用是「窮舉所有情形」。

（1）用 BFS 窮舉所有情形



所有的情形和可能發展可以被畫成「樹狀圖」。一層層依序把樹建立出來的過程中，每一層都在建立後被放到 Queue 裡，隨後該層的資料被取出來，進行下一步的窮舉，據以建立出下一層，直到 Queue 為空時，就得到了窮舉出的所有情形。

舉例來說，如果要得到「ABC 三個字母各一個的所有排列可能」，藉由數學運算可以知道共有 3! = 6 種可能。使用廣度優先搜尋可以把這 6 種可能都自動列出。

首先，選擇「開頭字母」時，只有 A、B、C 三個選項，因此第一層有三種可能。如果第一個字母已經選定 A，那麼第二個字母剩下兩個選項 B 和 C；如果第一個字母選定 B，第二個字母可以是 A 或 C，第一個字母是 C 時，第二個字母是 A 或 B。

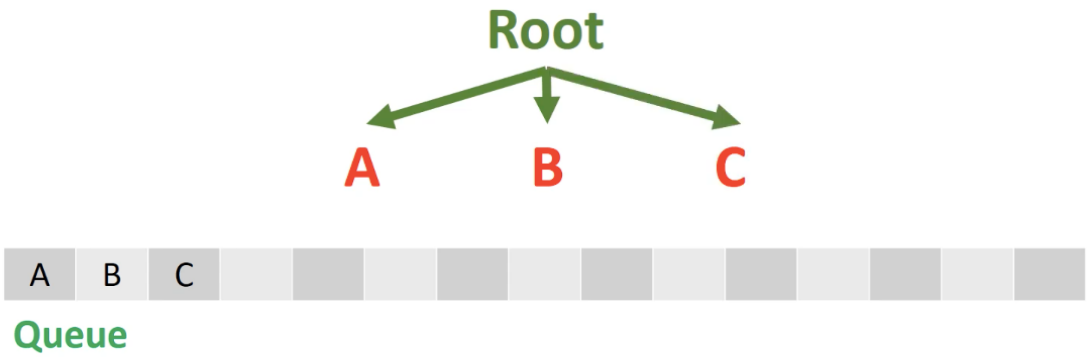
接下來，給定前兩個字母，第三個字母就已經被決定好了，比如前兩個字母是 AB 時，只有一種可能 ABC，前兩個字母是 BC 時，只有 BCA 一種可能。

就像這樣，窮舉所有情形的過程可以對應到一個樹狀圖，而建立樹狀圖的過程

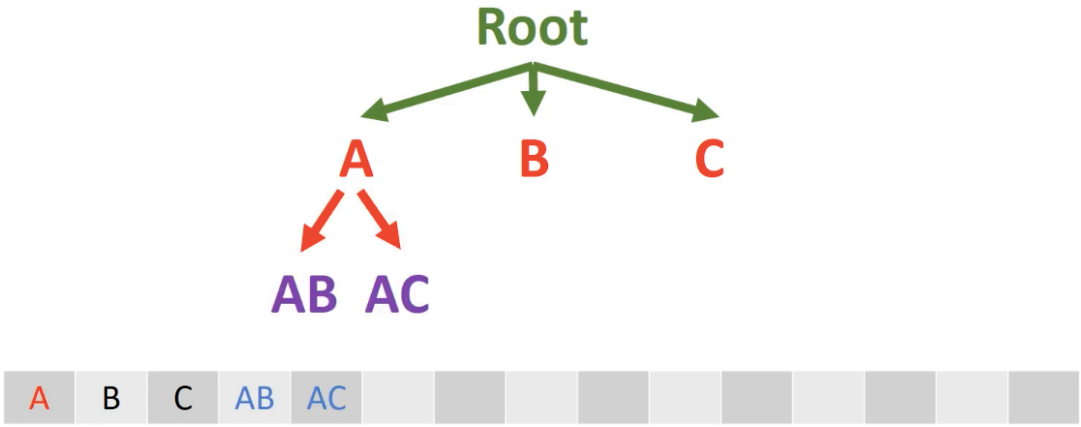
是從根節點開始「一層一層」向下進行，所以適合使用廣度優先搜尋。

（2）窮舉所有情形的過程

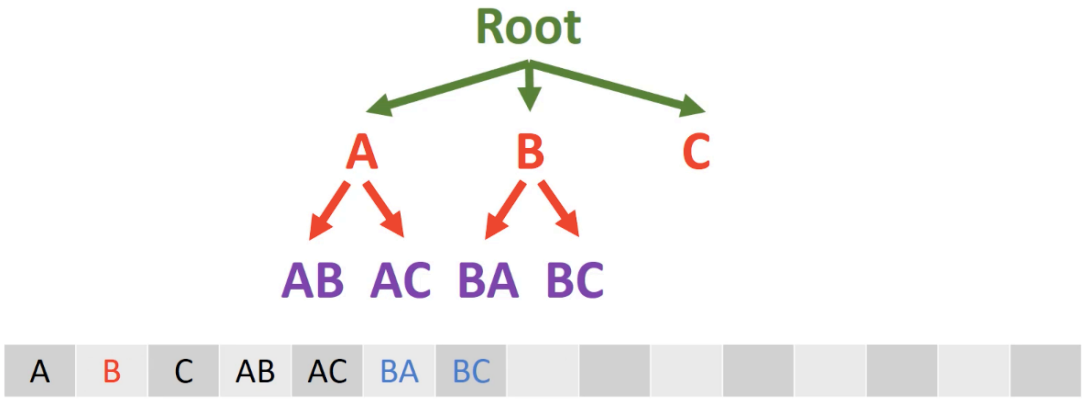
首先，第一個字母有三種可能：A、B、C，這三種可能都被放入 Queue 中。



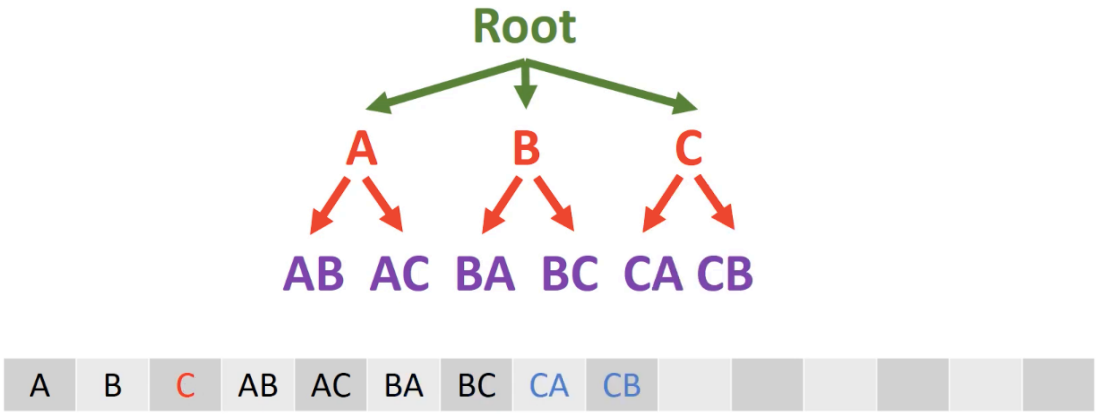
接著，同樣從 Queue 的最前面開始處理，此時開頭為 A，代表第一個字母選定 A，因此第二個字母只能選擇 B 或 C。AB 和 AC 兩種可能同樣被放到 Queue 中。



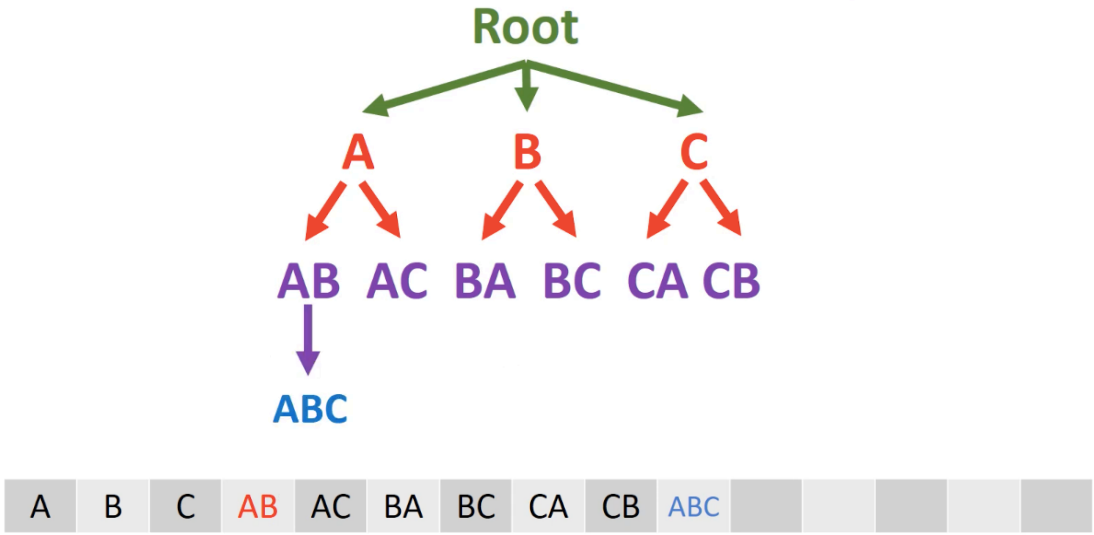
再來，第一個字母選定 B 時，第二個字母只能選擇 A 或 C，BA 和 BC 被放到 Queue 中。



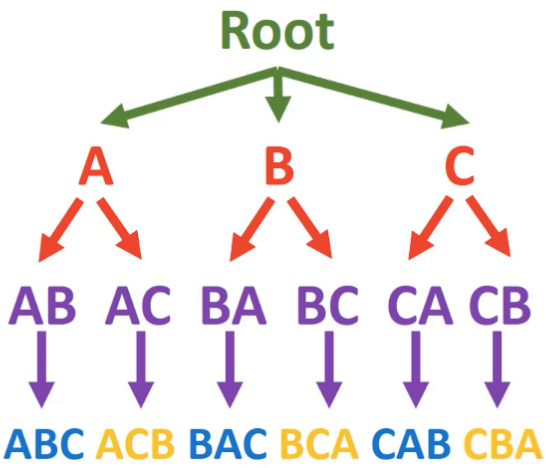
同樣的，CA 和 CB 也被放到 Queue 中，完成了這一層的建立。



往下繼續執行，此時 Queue 的開頭是 AB（注意 A、B、C 處理完都已經被從陣列中移除了），只有一種選定第三個字母的可能，即 ABC，ABC 同樣被放到 Queue 中。



接下來，ACB、BAC、BCA、CAB、CBA 都被放到 Queue 中。



A	B	C	AB	AC	BA	BC	CA	CB	ABC	ACB	BAC	BCA	CAB	CBA

在 Queue 的最後方，就是窮舉出的所有情形。

A	B	C	AB	AC	BA	BC	CA	CB	ABC	ACB	BAC	BCA	CAB	CBA

「窮舉所有情形」的問題乍看之下似乎與圖論沒有什麼關係，也不一定要轉換成圖來解決，但是許多問題一旦找到方法轉換為圖後，就可以運用圖論中已經建立起來的許多演算法（比如 BFS、DFS 等）解決。

（3）LeetCode #46. 排列 Permutations

A. 題目

給定一個陣列 $nums$，當中有一些互不相同的整數。回傳這些整數所有可能的排列，回傳的順序不限。

B. 出處

https://leetcode.com/problems/permutations/

C. 範例輸入與輸出

輸入：nums = [1,2,3]

輸出：[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

排列 Permutations

```
1  class Solution{
2  public:
3      vector<vector<int>> permute(vector<int>& nums){
4
5          // 取得 nums 的長度
6          int len = nums.size();
7
8          // BFS 使用的 queue
9          queue<vector<int>> permutation;
10
11         // 要回傳的結果
12         vector<vector<int>> final_result;
13
14         // 第一個數字（第一層）可以是 nums 中的每個元素
15         for (int i=0 ; i<len ; i++){
16             int number = nums[i];
17             // 建立一個長度為 1 的向量 [number]
18             vector<int> tmp(1, number);
19             // 放到 queue 中
20             permutation.push(tmp);
21         }
22
23         while(!permutation.empty()){
24
25             // 取出 Queue 中第一筆元素
26             vector<int> current = permutation.front();
27
28             // 從 Queue 中取得的資料長度跟 nums 元素數目相同時
29             // 把該資料加入結果中
30             // 比如 nums = [1,2,3] 時
31             // Queue 中已經處理到樹中最下面一層的元素 [1,2,3]
```

```
32                  if (current.size()==len){
33                      final_result.push_back(current);
34                      // 不用往下進行「加上其他數字」的步驟
35                      continue;
36                  }
37
38                  permutation.pop();
39
40                  // 要再加入其他數字時
41                  for (int i=0 ; i<len ; i++){
42
43                      // 繼續從 nums 中取出一個數字接在 current 後面
44                      int number = nums[i];
45
46                      // 要檢查 current 裡面是否已經包含 number 了
47                      // 比如 [1,2] 不可以再接一個 1 在後面
48                      auto iter = find(current.begin(),current.end(),number);
49                      if (iter!=current.end()){
50                          // 已經包含了，不能加進該排列裡
51                          continue;
52                      }
53                      else {
54
55                          // 用另一個向量來複製 current 後
56                          // 最後面加上新的數字
57                          vector<int> current_expand
58                          (current.begin(),current.end());
59                          current_expand.push_back(number);
60                          permutation.push(current_expand);
61                      }
62
63                  } // end of for
64
65              } // end of while
```

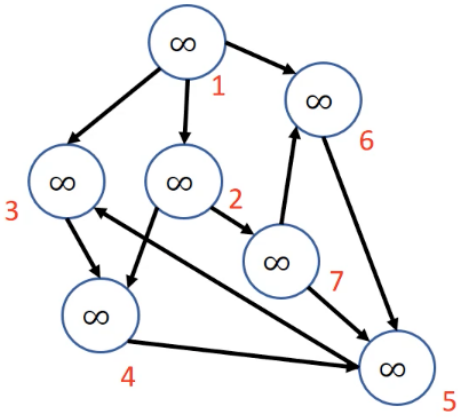
```
66
67              // 回傳得到的所有排列
68              return final_result;
69
70          } // end of permute
71  }; // end of Solution
```

4. 最短路徑

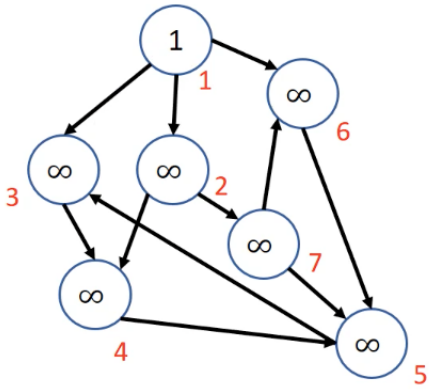
廣度優先搜尋另一個很常見的應用是「尋找最短路徑」。

所謂尋找最短路徑指的是給定兩個頂點，一個做為起點，一個做為終點，找出從起點到終點最少要花多少時間（或經過多少距離、花費多少成本等）。

（1）尋找最短路徑的進行流程



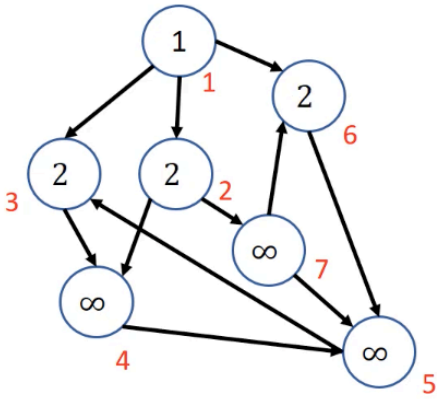
舉例來說，以上圖的頂點 1 做為起點、頂點 5 做為終點，找出起點到終點至少會花費多少時間或經過多少距離。只有在無權重圖中，也就是每個邊等價的情形下，才能使用廣度優先搜尋來求解最短距離，如果邊上有權重，則要使用其他專門解決該問題的演算法，待後續章節介紹。



1																			

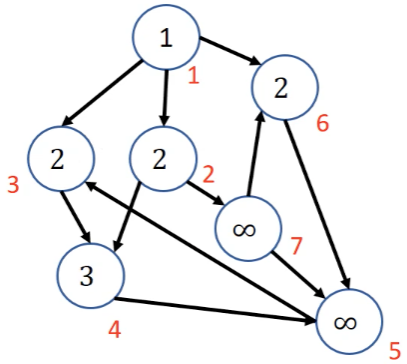
一開始，從起點 1 出發，進行廣度優先搜尋。首先，假設出發時的時間是「時間 1」，因此在頂點 1 寫上「1」，並把頂點放到 Queue 中。

因為頂點 1 與頂點 3、2、6 相鄰，所以這三個頂點被寫上「2」，代表時間 2 就可以到達這三個頂點任一者（假設走每一條邊都需要一單位時間），且這三個頂點被放 Queue 中。



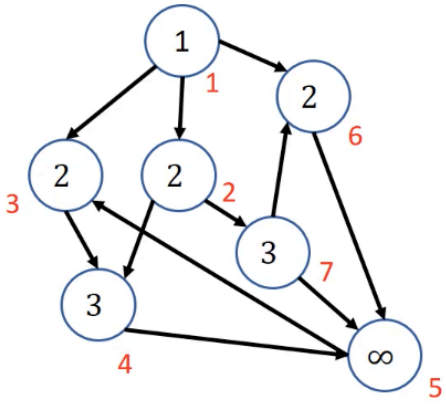
1	3	2	6															

接下來，Queue 中的頂點 1 已經處理完成，接著處理頂點 3，因為它與頂點 4 相鄰，所以頂點 4 被寫上 3（代表至少要到時間 3 才能從起點走到頂點 4），且把頂點 4 加到 Queue 中。



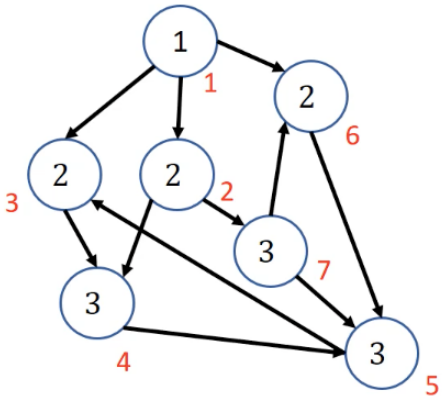
1	3	2	6	4														

接著，處理頂點 2，把相鄰的頂點 7 寫上 3 並放到 Queue 中，忽略另一個雖然相鄰，但已經放到 Queue 中過的頂點 4。



1	3	2	6	4	7													

處理頂點 6 時，在相鄰的頂點 5 寫上 3，並且同樣將其放到 Queue 中。



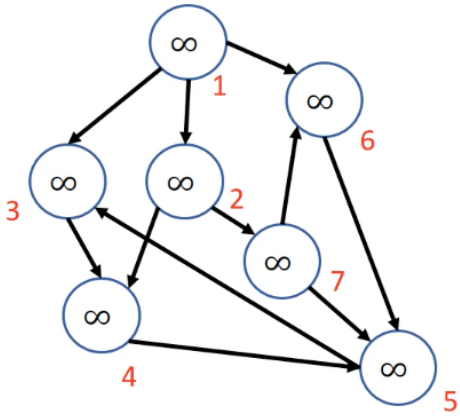
1	3	2	6	4	7	5												

透過這樣的「廣度優先搜尋」過程，把每個尚未造訪過的點都填上「上一個點的時間或成本加一」，就可以得到每個點至少需要的時間或成本。

過程中，一樣可以看成從起點「一輪輪向外擴散」。首先，第一輪是從起點擴散到所有與起點相鄰的頂點（3、2、6），接下來，第二輪擴散到所有「與起點相鄰的頂點」的相鄰頂點（4、7、5），以此類推，整個過程結束後，就可以得到從起點到圖上每個頂點的最短路徑（至少要經過幾個邊）。

因為過程基本上就是執行 BFS，因此時間複雜度同樣是 $O(|V| + |E|)$。

（2）實作「尋找最短路徑」



給定如上圖的數個城市與連接其間的道路，假設經過每一條道路需要花費的交通時間相等（即此為無權重圖），試找出從城市 1 到其他城市間的最短路徑。

使用先前實作的無向圖程式碼進行擴寫：

BFS 尋找最短路徑

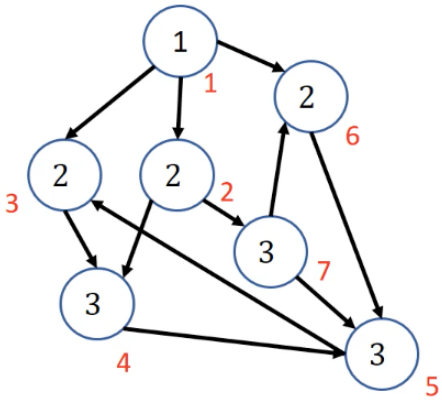
```
1   class Graph{
2   private:
3       ...
4   public:
5       ...
6       void Shortest_Path(int);
7   };
8
9   // 注意要採用無向圖的版本
10  bool Graph::add_edge(int from, int to){
11      edges[from-1].push_back(to-1);
12      edges[to-1].push_back(from-1);
13  }
14
15  /*
16  其他函式的程式碼
17  */
18
```

```
19 // 尋找最短路徑
20 void Graph::Shortest_Path(int start){
21
22     // 調整成索引值
23     start--;
24
25     vector<int> color(vertex,0);
26     // 記錄每個頂點距離的向量
27     vector<int> distance(vertex,-1);
28
29     queue<int> BFS_Q;
30     BFS_Q.push(start);
31     color[start] = 1;
32
33     // 把起點的距離設定為 1
34     // 視習慣，也可以設定為 0
35     distance[start] = 1;
36
37     // 印出起點
38     // 索引值與頂點編號相差 1
39     cout << "Distance of " << start+1 << ": " << distance[start] << endl;
40
41     while(!BFS_Q.empty()){
42
43         int current = BFS_Q.front();
44         BFS_Q.pop();
45
46         for (auto iter=edges[current].begin() ; iter!=edges[current].end() ;
47         iter++){
48             // 相鄰頂點還沒造訪過時
49             if (color[*iter]==0){
50                 BFS_Q.push(*iter);
51                 color[*iter] = 1;
52                 // 設定該相鄰頂點 *iter 的距離
```

```
53                          // 因為從 current 還要經過一條邊才能到 *iter
54                          // 所以 *iter 的距離比 current 的距離多 1
55                          distance[*iter] = distance[current]+1;
56
57                          // 輸出 *iter 的距離
58                          // 索引值與頂點編號相差 1
59                          cout << "Distance of " << *iter+1 << ": "
60                          << distance[*iter] << endl;
61                      }
62                  }
63
64                  color[current] = 2;
65
66          } // end of while
67
68  } // end of Shortest_Path
69
70  int main(){
71          Graph g(7);
72          g.add_edge(1,3);
73          g.add_edge(1,2);
74          g.add_edge(1,6);
75          g.add_edge(2,4);
76          g.add_edge(2,7);
77          g.add_edge(3,4);
78          g.add_edge(4,5);
79          g.add_edge(6,5);
80          g.add_edge(7,6);
81          g.add_edge(7,5);
82          g.Shortest_Path(1);
83
84          return 0;
85  }
```

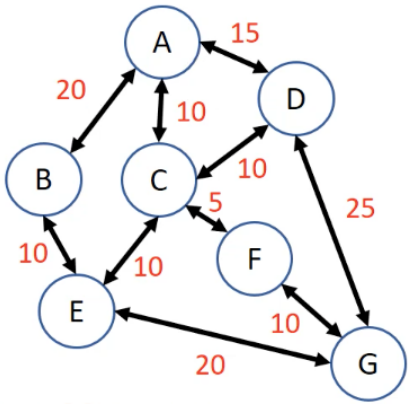
執行結果

```
Distance of 1: 1
Distance of 3: 2
Distance of 2: 2
Distance of 6: 2
Distance of 4: 3
Distance of 7: 3
Distance of 5: 3
```



1	3	2	6	4	7	5													

與剛才講解的圖對應，可以發現執行結果是正確的。



廣度優先搜尋找的其實是「邊數最少」的路徑，如果邊上有權重時（如上面的有權重圖），有時邊數比較少的路徑反而權重加起來比邊數多的路徑更大（就像有些路雖然距離短但容易塞車，要花更多時間），尋找有權重圖中最短路徑的方法請參考後續章節。

（3）LeetCode #1091. 二元陣列中的最短路徑 Shortest Path in Binary Matrix

A. 題目

給定一個二維、大小為 $n \times n$ 的二元陣列 $grid$，回傳最短「通關路徑」的長度，如果該陣列沒有通關路徑，回傳 -1。

「通關路徑」定義為從左上角（座標 $(0,0)$ 的方格）走到右下角（座標 $(n-1, n-1)$ 的方格）的路徑，且：

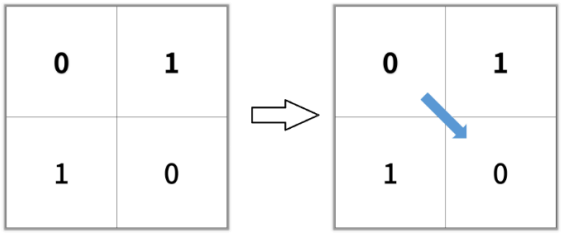
- 該路徑經過的所有方格值都是 0
- 該路徑經過的一個方格與下一個方格有邊相鄰，或者有角相鄰（往上下左右、左上、右上、左下或右下走）

通關路徑的長度指的是該路徑經過的方格數。

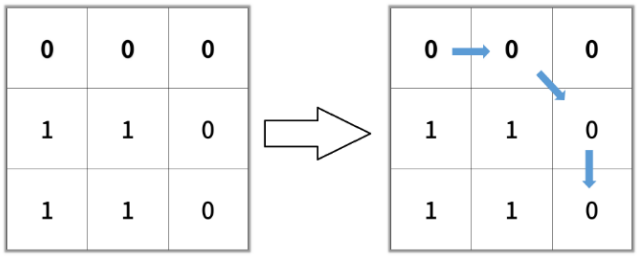
B. 出處

https://leetcode.com/problems/shortest-path-in-binary-matrix/

C. 範例輸入與輸出



輸入：grid = [[0,1],[1,0]]
輸出：2



輸入：grid = [[0,0,0],[1,1,0],[1,1,0]]
輸出：4

輸入：grid = [[1,0,0],[1,1,0],[1,1,0]]
輸出：-1

D. 解題邏輯
本題問的是最短路徑「造訪幾個方格」，因此每個方格與相鄰方格間的距離都被看作是相同的，可以適用廣度優先搜尋。

每次要移動到下一個方格時，可往八個方向之一行進，而行進的過程中，把每個造訪的方格的值都從 0 改成「該格與起點的距離」，這樣一來，右下角方格被填上的值就是回傳值。

為了避免陣列中原本的 1 值和造訪後填上的距離「1」搞混，在開始進行 BFS 前，可以先把陣列中所有的 1 都改成 -1，另外因為只有在算出的距離比可以走的方格目前的值小時，才更新距離，所以把 0 都先改成無限大，方便後續更新（整數的最大值 $2^{31} - 1 = 2147483647$）。

二元陣列中的最短路徑 Shortest Path in Binary Matrix	
1	class Solution{
2	public:
3	int shortestPathBinaryMatrix(vector<vector<int>>& grid){
4	
5	// 取出 grid 的大小
6	int rows = grid.size();
7	int cols = grid[0].size();
8	
9	queue<int> Position;
10	
11	// 例外處理：如果起點就是 1，沒有路徑，直接回傳 -1
12	if (grid[0][0]){
13	return -1;

```
14          }
15
16          // 一般情況
17          Position.push(0);
18
19          // 把原本陣列中 1 都改成 -1，避免與距離 1 混淆
20          for (int i=0 ; i<rows ; i++){
21              for (int j=0 ; j<cols ; j++){
22                  if (grid[i][j]){
23                      grid[i][j] = -1;
24                  } else {
25                      // 把陣列中的 0 都改成 int_max
26                      grid[i][j] = 2147483647;
27                  }
28              }
29          }
30
31          // 起點的距離是 1，因為此時路徑經過了一個方格
32          grid[0][0] = 1;
33
34          // 要造訪的八個方向的資料
35          // 每個索引值依序為
36          // 下、上、左、右、右下、左上、左下、右上
37          // 如索引值 7 的 (x,y) = (−1,1) 代表往右上移動
38          int direction_x[8] = {1,-1,0,0,1,-1,1,-1};
39          int direction_y[8] = {0,0,-1,1,1,-1,-1,1};
40
41          while(!Position.empty()){
42              // 取出 Queue 中最前面的資料
43              int current = Position.front();
44              Position.pop();
45
46              // 把位置 P 轉換為 (x,y) 形式
47              // P = x*cols + y;
```

```
48                  // (x,y) = (P / cols, P%cols)
49                  int x = current/cols;
50                  int y = current%cols;
51
52                  // 要造訪的方向有 8 個
53                  // 用設定好的八個方向的 x、y 偏移量來移動
54                  for (int i=0 ; i<8 ; i++){
55
56                      // 計算出要造訪的下個方格的位置
57                      int new_x = x + direction_x[i];
58                      int new_y = y+ direction_y[i];
59
60                      // 移動前的距離
61                      int distance = grid[x][y];
62
63                      // 如果移動之後超出邊界或者為牆（-1），不處理
64                      // 直接嘗試其他方向
65                      if (new_x<0 || new_y < 0){
66                          continue;
67                      }
68                      if (new_x>=rows || new_y >=cols){
69                          continue;
70                      }
71                      if (grid[new_x][new_y] == -1){
72                          continue;
73                      }
74
75                      // 還在邊界內時
76                      // 移動後方格要填上的距離
77                      int distance_now = distance + 1;
78
79                      if (distance_now < grid[new_x][new_y]){
80                          // 填上移動後方格的距離
81                          grid[new_x][new_y] = distance_now;
```

```
82                              // 把移動後的方格放入 Queue 中
83                              int P = new_x*cols+new_y;
84                              Position.push(P);
85                          }
86
87                  }
88
89                  // 如果右下角方格不是無限大，代表可以走到
90                  if (grid[rows-1][cols-1]!=2147483647){
91                      return grid[rows-1][cols-1];
92                  } else {
93                      return -1;
94                  }
95
96              } // end of while
97          } // end of shortestPathBinaryMatrix
98  }; end of Solution
```

5. 環的判別

廣度優先搜尋也可以用來確認「圖中是否有環的存在」。

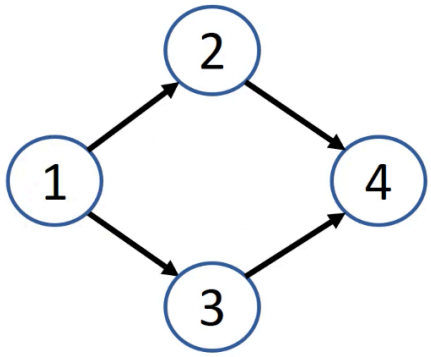
（1）如何判斷環的存在

A. 無向圖：將頂點放入 Queue 時，若發現該頂點存在 Queue 中，代表有環
B. 有向圖：計算 BFS 過程中頂點的 in-degree，較麻煩

無向圖可以很容易的被轉換成有向圖，但有向圖則沒有簡單的方法轉換成無向圖，因此可以專注在找到適用有向圖的解決方法上。

「無向圖」中，如果要把頂點放入 Queue 時，發現先前就已經造訪過該頂點了，就像在森林裡撒麵包屑，結果走一走發現又走到已經有麵包屑的地方，就代表先前走的路形成了一個環。

相對的，同樣的判別方法不能拿來確認「有向圖」中是否有環。

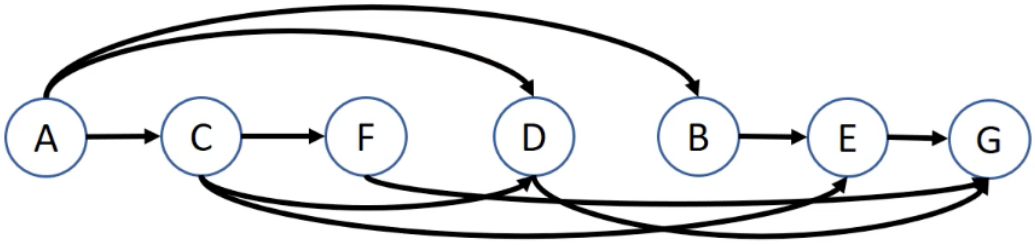


比如上面的有向圖中，一開始陣列中會放入 1，接下來，會放入 1 的相鄰頂點 2、3，變為 [1,2,3]（再移除 1），隨後在處理 2 時，會在 Queue 中加入 4，但是處理 3 時同樣會加入 4（這時頂點 4 已經是灰色，代表造訪過），就形成了剛才「遇到麵包屑」的情形，代表這張圖「如果是無向圖，一定有環」，但是有向圖中，這些指到頂點 4 的邊可能都以 4 為終點，並不形成環。

也就是說，針對有向圖必須另外尋找方法來確認環的存在。

（2）判斷有向圖中是否有環

回憶先前提過的有向無環圖的拓樸排序，若排序後的圖中有邊 $e(u,v)$，u 必排在 v 之前，使得所有的邊在圖中都是從左邊指到右邊。



一個入度為 0 的頂點在找尋環時被視為是「安全的」，因為沒有任何邊指向這個頂點，所以從這個頂點出發後，不可能再回到原點，代表這個頂點一定不在任何環中。

相對的，若圖中一開始沒有入度為 0 的節點的話，必定有環存在（可以透過反面的論述來理解，一個「沒有環存在的圖」一定會有至少一個入度為 0 的節點）。

這樣一來，就可以擬定出一個策略：從入度為 0 的頂點出發後，把該節點 u 的所有相鄰節點入度都 -1，因為從 u 指向這個相鄰節點的邊不會是任何環的一部份。

等到其他頂點的入度被降為 0 時，就確定了該頂點也是「安全的」，因此可以再從這個新的頂點出發，把所有相鄰節點的入度 -1。

不斷重複上述步驟後，如果所有節點的入度都被降為 0，就代表圖中沒有環。無法全部降為 0 時，得知圖中有環。

（3）Kahn's algorithm

上述的演算法稱為 Kahn's algorithm for Topological Sorting。

步驟如下：

A. 計算每個節點的入度 in-degree

B. 「已拜訪的節點數目」初始化為 0

C. 把入度為 0 的節點加入 Queue

D. 從 Queue 中 pop 出一個節點，「已拜訪的節點數目」+1

E. 把該節點的所有相鄰節點入度 -1

F. 如果某個節點入度被降為 0，將該點加入 Queue 中

G. 重複步驟 D 到 F，直到 Queue 為空

H. 如果「已拜訪的節點數目」=「總節點數」，沒有環，否則有環

「已拜訪的節點數目」代表的就是「安全的」頂點個數，如果所有頂點都是安全的，則圖中沒有環，反之有環。

不過，要確認環的存在，使用深度優先搜尋會更有效率，所以這個算法可以有概念即可。

Kahn's algorithm

```
1   BFS_Cycle(G,s)
2
3   // O(|E|)
4   for each e(u,v) in G:
5       in_degree[v]+=1
6
7   // O(|V|)
8   for each(vertex,value) in in_degree:
9       if in_degree[vertex] == 0
10          queue.push(vertex)
11
12  visited_node = 0
13
14  // O(|E|+|V|)
15  // while 最多會跑 |V| 次
16  // 這 |V| 次當中 for 迴圈會進行的總次數為 |E| 次
17  // 因為一個「相鄰頂點」一定是透過一條邊指到的
18  while(queue is not empty):
19      current_node = queue.pop()
```

```
20        for each vertex in current_node.adjacent():
21            in_degree[vertex] -= 1
22            if in_degree[vertex] == 0
23                queue.push(vertex)
24        visited_node += 1
25
26  if visited_node == |V|: return false
27  else: return true
```

效能分析

A. 初始化：O(|E|+|V|)
B. 處理所有頂點：O(|V|)
B. 處理所有邊：O(|E|)
C. 總和：O(|E|+|V|)

讀者可以運用此演算法重新寫一次「LeetCode #207. 課程安排 Course Schedule」，確定某些可能有先修要求的課之間有沒有辦法全部修完。

如果圖中存在環，比如向 A 課老師加簽時，老師要求要先修過 B 課，向 B 課老師加簽時，老師又要求修過 A 課，則一定沒有辦法全部修完。本題可以使用上方的演算法來解決。

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

6. 實戰練習

來看兩題實戰練習。

（1）LeetCode #404. 左葉節點的和 Sum of Left Leaves

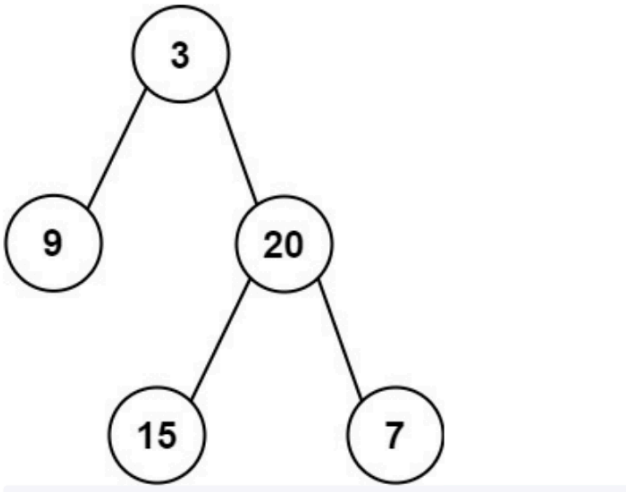
A. 題目

給定一個二元樹的根節點，回傳該樹中所有的左葉節點。

B. 出處

https://leetcode.com/problems/sum-of-left-leaves/

C. 範例輸入與輸出



輸入：root = [3,9,20,null,null,15,7]
輸出：24

兩個左葉節點（葉節點中，是父節點的左節點者）9 和 15 的和為 24。

D. 解題邏輯

本題可以把每個節點的子節點一個個放到 Queue 中，接下來依序取出 Queue 中的元素。

針對每個節點，檢查有沒有左節點，如果有，而且該左節點本身屬於葉節點（沒有子節點），就把值加到總和中。

左葉節點的和 Sum of Left Leaves

```
1   class Solution{
2   public:
3       int sumOfLeftLeaves(TreeNode* root){
4
5           // 例外處理：樹是空的
6           if (root==0)
7               return 0;
8
9           // 一般情形
10          int sum = 0;
11          queue<TreeNode> BFS;
12
13          // 從根節點開始做 BFS
14          BFS.push(root);
15
16          while(!BFS.empty()){
17              // 從 Queue 中取出最前面的元素
18              TreeNode* current = BFS.front();
19              BFS.pop()
20
21              // 檢查目前的節點 current（比如範例中的 3）的左節點
22              // 是否為葉節點（如範例中 9 的左右節點都是空的）
23              // current->left!=0：current 有左節點
24              // current->left->left == 0：左節點的左節點是空的
25              // current->left->right == 0：左節點的右節點是空的
26
27              if (current->left!=0){
28                  if (current->left->left == 0 && current->left->right == 0){
29                      // 把該左節點的值加入 sum 中
30                      sum += current->left->val;
```

```
31                          }
32                  }
33
34                  // 把 current 的子節點加入 Queue 中
35                  // 使樹中每一階層的節點都被依序加入 Queue
36                  if (current->left!=0){
37                          BFS.push(current->left);
38                  }
39                  if (current->right!=0){
40                          BFS.push(current->right);
41                  }
42
43              } // end of while
44              return sum;
45
46      } // end of sumOfLeftLeaves
47  }; // end of Solution
```

2. LeetCode #1020. 飛地數目 Number of Enclaves

A. 題目

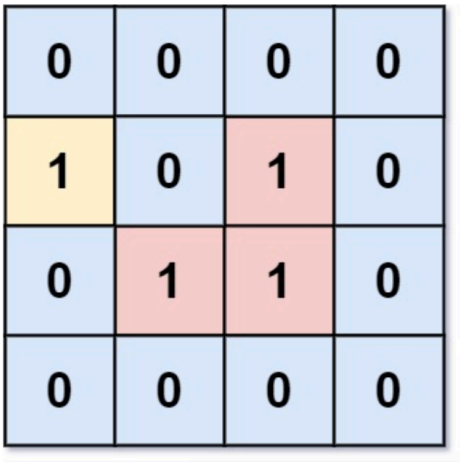
給定一個二維、大小為 $m \times n$ 的二元陣列 $grid$，每個值為 0 的方格代表海，值為 1 的方格代表陸地。

一次「移動」是從一個陸地方格走到相鄰（上下左右四個方向）的陸地方格，或者走到邊界之外。回傳所有「從這裡出發時，不論走多少步，都無法走到邊界外」的陸地方格數量。

B. 出處

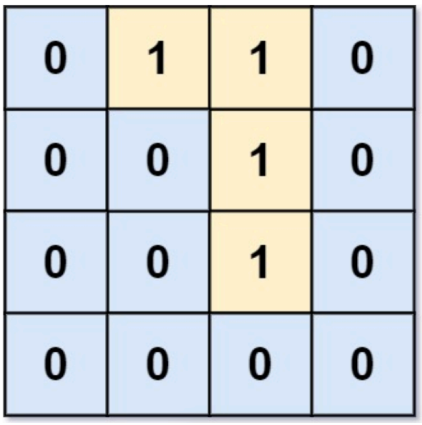
https://leetcode.com/problems/number-of-enclaves/

C. 範例輸入與輸出



輸入：grid = [[0,0,0,0],[1,0,1,0],[0,1,1,0],[0,0,0,0]]
輸出：3

中間的三個 1 連起來的陸塊上下左右都被海包圍，且沒有和邊界連接。左邊的 1 則因為與邊界相鄰，不算進飛地的數目中。



輸入：grid = [[0,1,1,0],[0,0,1,0],[0,0,1,0],[0,0,0,0]]
輸出：0

中間的陸塊因為往上與上方邊緣相接，所以不算飛地。

D. 解題邏輯

一個飛地就是一個完全沒有和任何邊界透過陸地連接的陸地方格。

本題因為與邊緣相接的陸塊不算作飛地，因此可以先從邊緣一圈所有的 1 出發做 BFS，並且把一路上經過的 1 都塗成 0。

做完之後，圖上還剩下的 1 都屬於飛地，計算其數量即可。

飛地數目 Number of Enclaves	
1	class Solution{
2	
3	// BFS 的函式
4	void BFS(vector<vector<int>>& grid, int rows, int cols, int start){
5	
6	queue<int> Position;
7	
8	// 傳入的 start 是開始 BFS 的起點
9	// 轉換成 (x,y) 形式

```
10      int x = start / cols;
11      int y = start%cols;
12
13      // 透過兩個陣列設定上下左右四個方向的 x、y 偏移量
14      int direction_x[4] = {-1,1,0,0};
15      int direction_y[4] = {0,0,-1,1};
16
17      Position.push(start);
18      while(!Position.empty()){
19          // 取出 Queue 中最前面的元素，並轉換為 (x,y) 形式
20          int P = Position.front();
21          Position.pop();
22          int x = P/cols;
23          int y = P%cols;
24
25          // 把取出的點改為 0
26          grid[x][y] = 0;
27
28          // 透過 x、y 偏移量，往上下左右四個方向移動
29          for (int i=0 ; i<4 ; i++){
30              int new_x = x + direction_x[i];
31              int new_y = y + direction_y[i];
32              // 邊界處理
33              if (new_x < 0||new_y<0){continue;}
34              if (new_x>=rows||new_y>=cols){continue;}
35
36              // 如果移動後是 0，不需加到 Queue 中
37              if (grid[new_x][new_y] == 0){
38                  continue;
39              }
40              // 只有移動後方格是 1 時需要加到 Queue 中處理
41              Position.push(new_x*cols+new_y);
42          }
43      } // end of while
```

```
44  } // end of BFS
45
46  public:
47      int numEnclaves(vector<vector<int>>& grid){
48
49          // 取得 grid 的大小
50          int rows = grid.size();
51          int cols = grid[0].size();
52          // 記錄飛地數目
53          int sum = 0;
54
55          for (int i=0 ; i<rows ; i++){
56
57              // Position = x*cols+y
58
59              // 檢查最左邊的直行
60              if (grid[i][0]==1){
61                  // 對應的 Position 是 i*cols
62                  BFS(grid,rows,cols,i*cols);
63              }
64              // 檢查最右邊的直行
65              if (grid[i][cols-1]==1){
66                  // 對應的 Position 是 i*cols+(cols-1)
67                  BFS(grid,rows,cols,i*cols+(cols-1));
68              }
69          }
70
71          for (int j=0 ; j<cols ; j++){
72              // 檢查最上面的橫列
73              if (grid[0][j]==1){
74                  // 對應的 Position 是 j
75                  BFS(grid,rows,cols,j);
76              }
77              // 檢查最下面的橫列
```

```
78                  if (grid[rows-1][j]==1){
79                      // 對應的 Position 是 (rows-1)*cols+j
80                      BFS(grid,rows,cols,(rows-1)*cols+j);
81                  }
82              }
83
84              // 用雙重迴圈檢查剩下幾個 1
85              for (int i=0 ; i<rows ; i++){
86                  for (int j=0 ; j<cols ; j++){
87                      if (grid[i][j]==1){
88                          sum++;
89                      }
90                  }
91              }
92              return sum;
93
94          } // end of numEnclaves
95  }; // end of Solution
```