建議對演算法與數據結構已經有足夠熟悉的學員跳過此章節

主要原因是我們課程的設計核心為實際去模擬當你真實遇到該問題時會有什麼樣的想法，但講義內容其實已經有部分暗示解法，但實際遇到問題時並不會知道接下來的題目會用到解法的方向。

Graph

**DFS 跟 BFS 的區別在於一個是 stack 一個是 queue*

Adjacency Matrix/ List

Adjacency matrix 或 Adjacency list 通常適用於 graph 的題目，他們的目的是方便儲存點跟邊的資訊

Adjacency List

List 的存法怎麼設定因人而異，不過通常會先開好所有點的陣列然後再把它連到的點 push 進去。
今有一邊連結 x 和 y 兩點，若是無相圖（undirectly graph）則在 x 那格中存放 y，在 y 那格也會存放 x。若是有相圖（directly graph）且為 x 指向 y，那則在 x 那格中存放 y 即可

Adjacency Matrix

一開始開好一個 $n * n$ 的二維陣列，並將所有值都設為零。今有一邊連結 x 和 y 兩點，則把 x, y 那格改成 0。

如果圖的邊很稀疏時適合用 adjacency list，因為最佳情況可能只會用到 $O(m)$ 空間，但若是用 matrix 空間一定是 $O(n^2)$。一般狀況用哪個方式存資料都一樣，不過如果題目常常問 x, y 兩點中間有沒有邊的話就適合用 matrix 因為 $O(1)$ 即可知道結果。

轉換題目成 Graph 形式

將題目轉換成圖的要點就是定義好甚麼是點甚麼是邊，且有可能點會指向自己。以 Leetcode 200 題為例：
給定一個二維陣列包含 1 和 0，1 代表島 0 代表海，上下左右相鄰的 1 代表同個島，試問這個地圖裡有多少個島？
這題的點就是陣列裡值為 1 的元素，而邊則是由相鄰兩個 1 產生。想要知道有幾個島可以遍歷所有 1 的點，使用 DFS 把經過的點標起來，這樣走過的點就不會再被算到。

範例

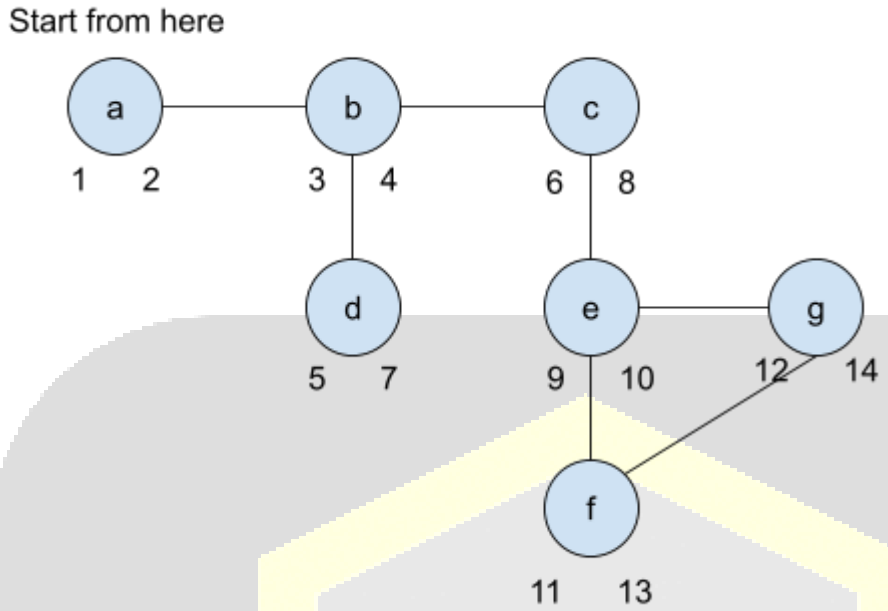
```
public class Solution {
      int n,m;
      static int[] dx = {0,0,1,-1};
      static int[] dy = {1,-1,0,0};
      static int directionCount = 4;
      private void dfs(int x, int y, int[][] grid) {
            grid[x][y] = 0;
```

```
            for (int i = 0 ; i < directionCount ; i++) {
                  int targetX = x + dx[i], targetY = y + dy[i];
                  if (targetX >= 0 && targetX < n && targetY >=0 &&
            targetY < m && grid[targetX][targetY] == 1 ) {
                        dfs(targetX, targetY, grid);
                  }
            }
      }
      public int numberOfIslands(int[][] grid) {
            n = grid.length;
            if (n == 0) {
                  return 0;
            }
            m = grid[0].length;
            int ans = 0;
            for ( int i = 0 ; i < n ; i++) {
                  for ( int j = 0 ; j < m ; j++) {
                        if (grid[i][j] == 1) {
                              dfs(i,j,grid);
                              ans++;
                        }
                  }
            }
            return ans;
      }
}
```

Breadth-first search (BFS)

BFS 的實作方式運用到 queue 的技巧，使用迴圈遍歷每個點，如果此點還未經過就把它塞進 queue。使用 while 迴圈判斷 queue 是否為空，不是的話就將裡面的點丟出來然後去看跟此點相連的其他點，並將和他相連的點且還未標記過的點（一樣可以設一個 visited[n] 的陣列紀錄點是否有走過）丟進 queue。

下圖為使用 BFS 遍歷圖時點進入跟移出 queue 的順序，點左下的數字代表進入右下角代表移出。



最短路徑

只有圖的邊長都是 1 的情況才可以用 BFS 實作找尋最短路徑，因為當邊長不一樣時我們得確保是邊長最短的點先被 pop 出來。以下兩種情況要以別的方式實作：

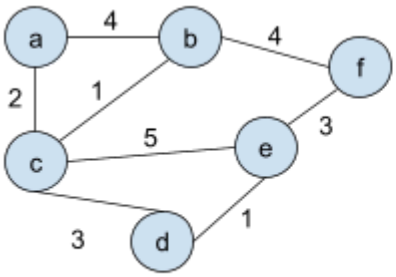
邊長只有 0 和 1

使用 double end queue 存放邊，如果邊長是 1 就將此點從 queue 後面放入，如果邊長是 0 則是將此點從 queue 前面放入，這樣才能讓 queue 維持前面數值比較小。這樣 0 的邊才會先被拿出來。

邊長長度都不同

使用 Dijkstra Algorithm 會需要用到 priority queue。實作方式為一開始全部點與原點的距離都設為無限大，並把原點的值改為 0 放入 queue，並依次更新與原點相連的點並將其放入 queue。之後我們需要將目前在 queue 離原點最近的點 pop 出來去更新其他點的值，這樣確保每個點能夠更新別人的前提是他已經不會在被更動了（也就是此點與原點的距離已經是是最小值）。更新點的時候要記得更改他的 parent（parent 代表是哪個點去更新這個點），這樣在最後才能由最後一個點推得最短路徑。
為甚麼每次都 pop queue 裡面最小的值能夠維持最短路徑，因為如果這個點已經是目前最小的值那就不可能有任何其他點能讓這個點的值變更小，因此可以拿這個點去更新其他點。

以下示範從 a 點到其他點的距離更新方式：



Point	Distance	parent
a	0	a
b	infinity	null
c	infinity	null
d	infinity	null
e	infinity	null
f	infinity	null

Calculate the distance from a to other points

Point	Distance	parent
a	0	a
b	4	a
c	2	a
d	infinity	null
e	infinity	null
f	infinity	null

pop a
push b, c

Point	Distance	parent
a	0	a
b	3	c
c	2	a
d	5	c
e	7	c
f	infinity	null

pop c
push d, e

Point	Distance	parent
a	0	a
b	3	c
c	2	a
d	5	c
e	7	c
f	7	b

pop b
push f

Point	Distance	parent
a	0	a
b	3	c
c	2	a
d	5	c
e	6	d
f	7	b

pop d

Point	Distance	parent
a	0	a
b	3	c
c	2	a
d	5	c
e	6	d
f	7	b

pop e