Homework 2

40947007S 資工113 張欀齡

1. 針對所需的相關資訊，以下為詳細說明：
    - 機器軟硬體規格
        - 機器：MacBook Pro (13-inch, 2019, Two Thunderbolt 3 ports)
        - 處理器：1.4 GHz 四核心Intel Core i5
        - 記憶體：8 GB 2133 MHz LPDDR3
        - 顯示卡：Intel Iris Plus Graphics 645 1536 MB
    - 作業系統：MAC OS
    - 開發軟體版本：MAC OS Montery 12.6.3
    - 如何執行程式
        - `python3 IDS.py`
        - `python3 IDASTAR.py`
    - 聯絡電話：0905632902
2. 測試用輸入檔說明

    使用python `random`函式來製作輸入檔，盤面大小為 1 - 60。以下為程式碼：

    ```
    import random

    with open("input.txt", "w") as f:
        n = random.randint(1, 60) # 隨機生成此筆測資盤面大小
        seq = [random.randint(0, 1) for _ in range(n)] # 隨機生成0 1數列
        f.write(" ".join(map(str, seq))) # 寫入檔案
    ```

    自己製作的測資：(前四筆為標靶遊戲中測資)

    下面用來舉例以及列出表現的部分都會利用這裡的測資，可對照測資編號。

測資編號	測資大小	測資
(1)	3	[1 1 1]
(2)	4	[1 1 1 1]
(3)	5	[1 1 1 1 1]
(4)	9	[1 1 1 1 1 1 1 1 1]
(5)	6	[1 0 0 1 1 0]
(6)	8	[0 1 1 0 1 1 1 0]
(7)	9	[0 1 0 0 1 0 1 0 0]
(8)	11	[0 1 0 1 0 0 1 1 0 1 0]
(9)	15	[1 0 0 0 1 1 0 0 1 1 1 0 1 0 0]
(10)	17	[0 1 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0]
(11)	20	[0 1 0 0 1 0 0 1 0 0 1 1 1 0 1 1 0 1 1 0]
(12)	23	[0 0 1 0 1 0 1 0 1 1 1 1 1 0 0 1 0 0 0 0 1 1 1]
(13)	30	[1 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 1 0 1 1 0 0 1 1 1 0 0 0 1 1]
(14)	30	[全部皆為1]
(15)	40	[0 1 1 0 0 0 1 1 0 0 1 1 0 0 1 1 0 1 1 1 1 1 1 0 0 1 0 0 1 1 1 1 0 0 1 0 1 0 1 0]
(16)	40	[全部皆為1]
(17)	50	[全部皆為1]
(18)	60	[全部皆為1]

3. 如何執行`IDS.py`以及`IDASTAR.py`

   詳細註解已寫在程式碼中。

   - IDS
     - 請見程式碼`IDS.py`
     - 如何執行：`python3 IDS.py`

- IDASTAR
  - 請見程式碼 `IDASTAR.py`
  - 如何執行：`python3 IDASTAR.py`

4. 第一支程式 `IDS.py` 詳細說明

   第一支程式使用的方法是有深度限制的遞迴版 DFS，以下詳細介紹。

   - 使用什麼方法

     使用的方法是有深度限制的遞迴版 DFS。

     盤面表示：盤面使用 python 的 `list` 來存，例如輸入 `1 1 1`，我的盤面就是 `[1, 1, 1]`，因為 python 輸入時會以字串的形式輸入，因此有先將資料 map 成整數型態再存入 list。

     盤面資訊儲存：需要儲存的資訊有走步、是否走過、目前深度、限制深度、衍生盤面、盤面大小。我將走步資訊、紀錄是否走過的陣列、衍生盤面分開紀錄，其餘資訊在遞迴時會一併更新傳入。

     走步產生：設置一個陣列 `move`，在每次遞迴一起傳入，紀錄走步資訊。

     判別重複：設置一個 dictionary，紀錄每個深度走過的盤面。

     如何取答案：判斷盤面所有數相加是否等於 0，若等於 0 代表已經找到解，會回傳一個 tuple，分別為 True 以及走步 `move`。

     是否為最佳解：用此種方法求得的解必為最佳解，因為是按照深度一層一層往下，一定會先找到步數最少的解。

     是否會跑不停：用此種方法在目前的嘗試下，盤面超過 30 就需要跑非常久，但是若讓它慢慢跑，最後會找到答案，不會跑不停。

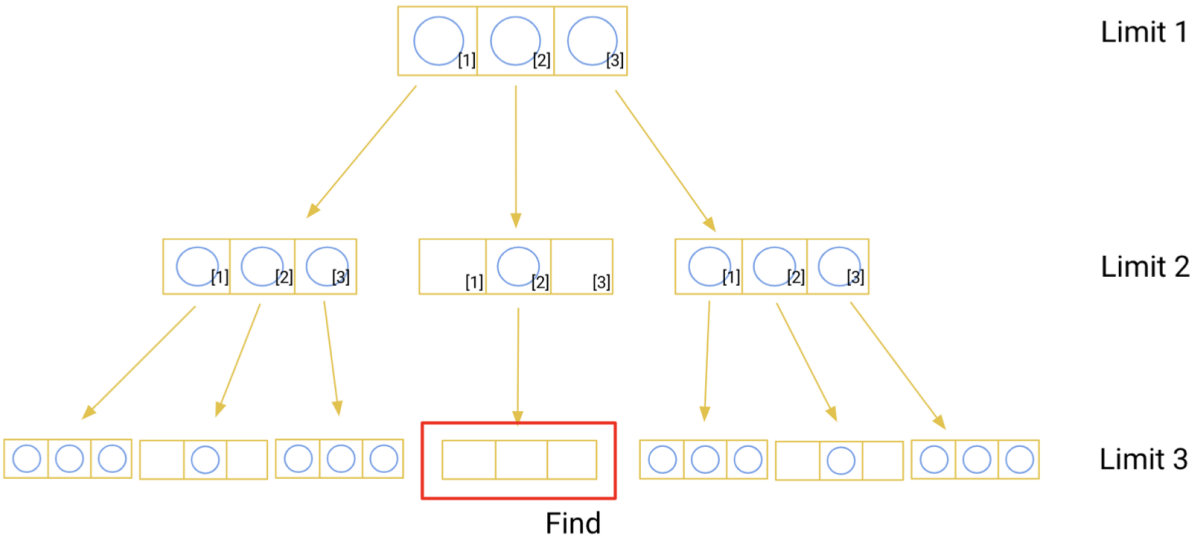
     記憶體是否會爆掉：在目前的嘗試下，盤面大小60以內記憶體不會爆掉。

     是否會無解：在癌細胞全滿的情況下，有一個規律的解法，因此無論盤面一開始為何，我們只需要讓衍生的癌細胞填滿所有盤面，就必定有解，而讓衍生的癌細胞填滿所有盤面這件事只需要按一些沒辦法讓細胞越來越少的位置就能夠達成。

   - 資料結構

     資料結構如下圖，每一次輸入遞迴式就是一個 list，例如：在 limit 1 時我按最左邊標號 [1] 的節點，結果還是會衍生出 [1, 1, 1] 的陣列，因此就會再將衍生後的細胞list [1, 1, 1]

傳入函式繼續做遞迴，直到 limit 超過或是找到解法。若在 limit 1 時我按下最中間標號 [2] 的節點，結果會衍生出 [0, 1, 0] 的陣列，就會走中間那條路，以此類推。



- 技術

  此支程式的核心技術分別為三個函式，以下詳細介紹（完整程式見 `IDS.py`）

  - `iterative_deepening_dfs(input_list, n)`

    傳入參數為最初始的輸入測資 list，以及盤面大小，這個函式主要在控制 limit 大小，例如在 limit 為 2 時 IDS 下去找發現沒有解，就會回傳到這裡，之後此函式就會再將 limit 加一，再做下一次的 IDS。

    函式程式碼：

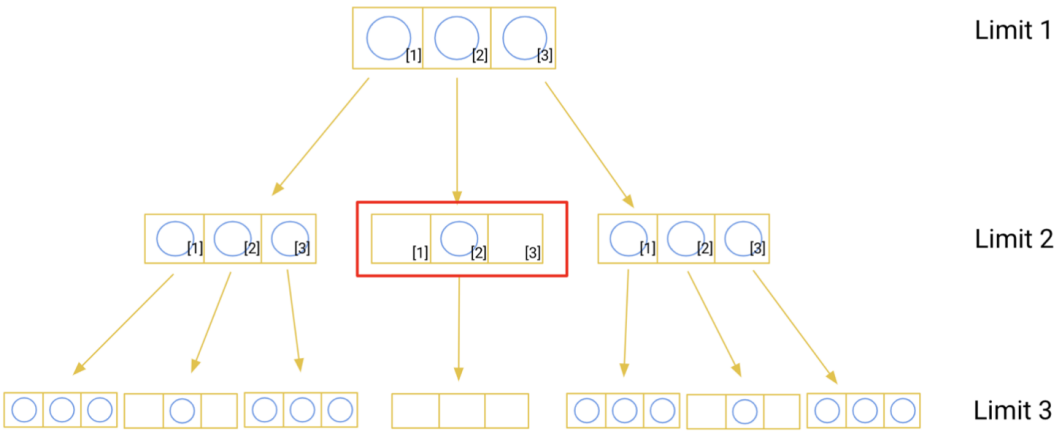
```
# 控制每個搜尋的深度limit
def iterative_deepening_dfs(input_list, n):
    depth = 0 # 初始值為0
    while True:
        move = []
        dic = {}
        for i in range(500):# 初始字典各深度list，為了紀錄該層相同盤面
            dic[i] = []
        result = depth_limited_dfs(input_list, 0, depth, n, move,
dic) #傳入最初盤面，開始做遞迴，回傳值為一tuple(是否找到解, 走步list)
        if result[0]:
            return result[1]# 若有解則回傳移動步驟
        depth += 1 #若沒找到解則把限制深度加一，再傳入遞迴
        if depth > 500: #目前設置若到深度500還沒有解，就停止程式
            return False
```

- `depth_limited_dfs(input_list, depth, limit, n, move, dic)`

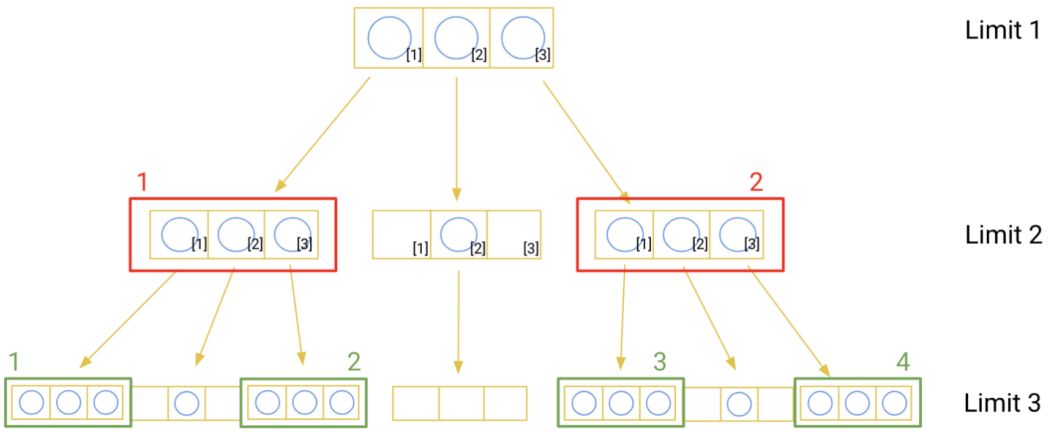
  主要在做遞迴的部分，傳入參數為每次遞迴輸入的盤面，目前深度，深度限制大小，盤面大小，紀錄走步移動之 list，紀錄走過節點之 dictionary。

  這個函式主要有三個判斷，第一個是判斷傳入的盤面上是否還有癌細胞，如果沒有，表示找到解法，則回傳成功以及走步資訊，如果還有就會去判斷目前的深度是否超過限制，若超過就回傳 False 並回到上一層 `iterative_deepening_dfs`，讓深度限制再多一層，若沒有超過深度限制則會用一個迴圈去選每一個癌細胞，並利用 `cell_grow` 來生成選了該位置的癌細胞後衍生的盤面結果，再把此結果繼續傳入此支函式 `depth_limited_dfs` 做遞迴。在這個函式中，做了兩個處理使其效率較佳：

  1. 在選擇要消滅哪個癌細胞時，只會點選有癌細胞的部分，沒有癌細胞的地方不會去選，例如在下圖紅色框框的地方，會去判斷每個位置是否有癌細胞，因此像是 [1], [3] 就不會被選到，所以可以節省一些不必要得遞迴。



  2. 在同一層 limit 若有重複相同的盤面不會再傳下去做遞迴，例如在 limit 2 的地方，可以看到用紅色框框匡起來的那兩個盤面相同，代表他們傳下去做遞迴的結果會是一樣的，因此只會做一次，只會做紅色標號 1 的那個盤面，在後面遇到紅色標號 2 的盤面時，因為該盤面前面已經被記錄做過了，因此不會去做該盤面的遞迴。同理，在 limit 3 時，有四個綠色框框的盤面是相同的，但也一樣只會做綠色標號 1 的盤面。做法是使用 python dictionary，紀錄深度與走過盤面 list 的對應，在傳入下一次遞迴時會先判斷該深度是否已經存在該盤面，若沒有，就做遞迴，若已經存在代表已經做過，就不做遞迴。該處理能夠提升很大的效率，在還沒使用這個技術時，我的程式跑到大約 limit 9 ~ 10 就需要跑好幾個小時，加入這個判斷後同樣的數據只需要跑 1 秒左右。



函式程式碼：

```
# 主要遞迴部分
def depth_limited_dfs(input_list, depth, limit, n, move, dic):
    if dic[depth].count(input_list) == 0: # 若此盤面在該層還會走過，將它加入字典
        dic[depth].append(input_list)
    depth += 1
    total = sum(input_list) # 計算盤面癌細胞數量
    if total == 0: # 若為0表示已完全消滅，則回傳True以及移動步驟
        return (True, move)
    elif depth >= limit: # 若深度已經超過，就回傳False
        return (False, None)
    else:
        for i in range(n): # 選取要消滅細胞，有n種可能
            if input_list[i] != 0: # 有癌細胞的地方才可以選
                m = move[:]
                m.append(i+1) # 紀錄走步
                new_cell = cell_grow(input_list, i) # 處理衍生細胞，傳回新的盤面
                if dic[depth].count(new_cell) == 0: # 判斷該盤面在該層是否展開過
                    result = depth_limited_dfs(new_cell, depth, limit, n, m, dic) # 沒有展開過就傳入遞迴
                    if result[0] == True:
                        return (True, result[1]) # 找到解法就回傳True，以及走步
    return (False, None)
```

- `cell_grow(input_list, index)`

該函式主要在處理消滅一個癌細胞後衍生的癌細胞盤面，傳入參數為消滅前的盤面以及選擇消滅的位置。這裡也有經過優化處理，使其效率更佳。

原本的做法：

使用 for 迴圈跑完整個盤面去做判斷看哪些需變成 1，此方法光是處理衍生盤面就花掉了 $O(n)$ 的時間複雜度，以及前面選擇消滅哪個癌細胞的時間複雜度 $O(n)$，時間複雜度就到了 $O(n^2)$，因此效率非常差。

優化的做法：

將原本盤面消滅癌細胞的位址修改為 0，再對此盤面做左移及右移，存成兩個陣列 `left_shifted`, `right_shifted` 並對這兩個陣列做 OR 的運算，得到的結果便是衍生的盤面。這個做法在處理衍生盤面只花了 $O(1)$ 的時間複雜度，因此效率較佳。

函式程式碼：

```
# 處理衍生細胞盤面
def cell_grow(input_list, index):
    left_shifted = input_list[:]
    right_shifted = input_list[:]
    left_shifted[index] = 0 # 將選擇消滅的細胞設為0
    right_shifted[index] = 0
    right_shifted.pop() # 處理右移
    right_shifted.insert(0, 0)
    left_shifted.pop(0) # 處理左移
    left_shifted.append(0)
    cell = np.bitwise_or(left_shifted, right_shifted).tolist() #使用
numpy函式做OR運算
    return cell
```

以上為實作 IDS 時用到的技術以及一些特殊處理，主要針對不處理重複盤面以及產生衍生癌細胞之盤面進行優化，且優化結果也相當明顯。

- 表現

這裡將優化前及優化後的數據進行比較，使用標靶遊戲中四個測資的表現如下：

優化處理指的是上述提到的

1. 在同一層 limit 若有重複相同的盤面不會再傳下去做遞迴
2. 將處理衍生盤面的計算降為 $O(n)$

優化前：

盤面	執行時間
[1, 1, 1]	0.000109 (s)
[1, 1, 1, 1]	0.000860 (s)
[1, 1, 1, 1, 1]	0.022374 (s)
[1, 1, 1, 1, 1, 1, 1, 1, 1]	好幾個小時（未跑完，衍生資料量太大）

優化後：

盤面	執行時間
[1, 1, 1]	0.000118 (s)
[1, 1, 1, 1]	0.000612 (s)
[1, 1, 1, 1, 1]	0.001661 (s)
[1, 1, 1, 1, 1, 1, 1, 1, 1]	0.063150 (s)

可以發現在資料量很小時，差異不大，但當資料量大於 5 之後，沒有優化的版本因為一直跑很多重複的盤面，導致時間成倍數成長，因此優化後的版本表現較好。

以下使用測資編號（見上面第二題題目自己製作測資之表格）來進行舉例

舉例：

輸入測資編號(4)的測資，執行`python3 IDS.py`，我們可以得到輸出結果為

```
Total run time = 0.06315016746520996 seconds.
An optimal solution has 14 moves:
2 3 4 5 6 7 8 2 3 4 5 6 7 8
```

輸入測資編號(6)的測資，執行`python3 IDS.py`，我們可以得到輸出結果為

```
Total run time = 0.02074599266052246 seconds.
An optimal solution has 9 moves:
6 3 2 2 3 4 5 6 7
```

輸入測資編號(9)的測資，執行`python3 IDS.py`，我們可以得到輸出結果為

```
Total run time = 1.8894200325012207 seconds.
An optimal solution has 20 moves:
6 9 10 11 12 13 14 2 3 4 5 6 7 8 9 10 11 12 13 14
```

輸入測資編號(11)的測資，執行`python3 IDS.py`，我們可以得到輸出結果為

```
Total run time = 20.815766096115112 seconds.
An optimal solution has 29 moves:
5 10 11 12 13 14 15 16 17 18 19 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5
4 3 2
```

輸入測資編號(13)的測資，執行`python3 IDS.py`，我們可以得到輸出結果為

```
Total run time = 1015.6024141311646 seconds.
An optimal solution has 43 moves:
1 16 17 18 19 20 21 22 23 24 25 26 27 28 29 29 28 27 26 25 24 23 22 21
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2
```

輸入測資編號(14)的測資，執行`python3 IDS.py`，我們可以得到輸出結果為

```
Total run time = 654.6348621845245 seconds.
An optimal solution has 56 moves:
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 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7
6 5 4 3 2
```

輸入測資編號(16)的測資，執行`python3 IDS.py`，我們可以得到輸出結果為

```
Total run time = 10105.787791013718 seconds.
An optimal solution has 76 moves:
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 39 38 37 36 35 34 33 32 31 30 29 28
27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2
```

- 耗用的時間及空間
  - 時間複雜度

    這支程式使用的是迭代加深深度優先搜尋演算法，其時間複雜度取決於搜尋深度的上限以及每個節點的擴展次數。每個節點最多擴展 n 次（n 是給定的盤面長度），因此時間複雜度為 $O(n^d)$，其中 d 是搜尋的深度。但是在此實現中，使用了一個

字典來儲存已訪問的節點，從而避免了訪問已經訪問過的節點，這樣可以大大減少節點數，從而減小時間複雜度。因此，實際的時間複雜度可能比 $O(n^d)$ 小得多，取決於實際搜索的節點數。

- 空間複雜度

  這個程式的空間複雜度取決於兩個因素：搜尋深度的上限和每個節點擴展出的節點數量。

  在這個程式中，我們使用了一個字典 dic 來儲存已經訪問過的節點，並且在每個深度上獨立使用一個 list 來存儲已經訪問過的節點，因此空間複雜度與搜尋深度的上限有關，即為 $O(dn)$，n 是給定的盤面長度，d 是搜尋的深度。

  此外，在每個節點擴展時，都會創建一個新的 cell 列表來代表新的盤面，即為 $O(n)$。

  input_list 儲存了輸入數據，是一個長度為 n 的整數列表，佔用 $O(n)$ 的空間

  因此，綜合考慮這些因素，這個程式的空間複雜度為 $O(dn)$，不過由於有使用一個字典來儲存已訪問的節點，從而避免了訪問已經訪問過的節點，這樣可以大大減少節點數，因此實際的空間複雜度也可能比較小。

- 能解到多大盤面

  目前嘗試到盤面大小為 50 的測資，其執行時間如下表：

盤面大小（括號內為測資編號）	**執行時間**
20 (11)	20.815766 (s)
30 (14)	654.634862 (s)
40 (16)	10105.787791 (s)
50 (17)	133379.965153 (s)

5. 第二支程式 `IDASTAR.py` 詳細說明

第二支程式使用的方法是有 f - cost 限制的遞迴版 DFS，IDA*，以下詳細介紹。

- 使用什麼方法

  第二支程式使用的方法是有 f - cost 限制的遞迴版 DFS，IDA*，以下詳細介紹。

  使用的方法是有 f - cost 限制的遞迴版 DFS。

盤面表示：盤面一樣使用 python 的`list`來存，例如輸入`1 1 1`，我的盤面就是`[1, 1, 1]`，因為 python 輸入時會以字串的形式輸入，因此有先將資料 map 成整數型態再存入 list。

Heuristic funtion：Heuristic funtion 為選取這個消滅節點後，衍生盤面的癌細胞數量。

因此 f - cost 公式為：$f(n) = g(n) + h(n)$

$g(n)$為已消滅細胞數，$h(n)$ 為 Heuristic funtion，$f(n)$為 f limit。

盤面資訊儲存：需要儲存的資訊有走步、是否走過、目前深度、f - cost 限制、next - f、衍生盤面、盤面大小。我將走步資訊、紀錄是否走過的陣列、衍生盤面分開紀錄，其餘資訊在遞迴時會一併更新傳入。

走步產生：設置一個陣列`move`，在每次遞迴一起傳入，紀錄走步資訊。

判別重複：設置一個 dictionary，紀錄每個深度走過的盤面。

如何取答案：判斷盤面所有數相加是否等於 0，若等於 0 代表已經找到解，會回傳一個 tuple，分別為 True 以及走步`move`。

是否為最佳解：用此種方法求得的解必為最佳解，因為每次都是選總消滅細胞數最少的，一定會先找到步數最少的解。

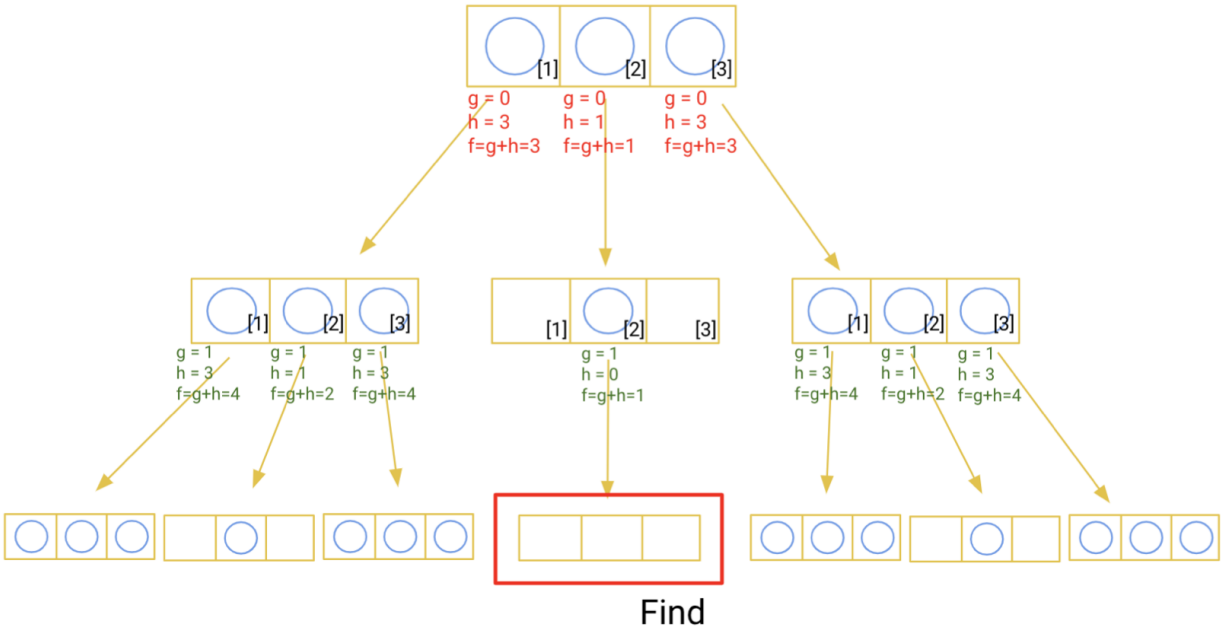
是否會跑不停：用此種方法在目前的嘗試下，盤面超過 50 就需要跑非常久，但是若讓它慢慢跑，最後會找到答案，不會跑不停。

記憶體是否會爆掉：在目前的嘗試下，盤面大小60以內記憶體不會爆掉。

是否會無解：在癌細胞全滿的情況下，有一個規律的解法，因此無論盤面一開始為何，我們只需要讓衍生的癌細胞填滿所有盤面，就必定有解，而讓衍生的癌細胞填滿所有盤面這件事只需要按一些沒辦法讓細胞越來越少的位置就能夠達成。

- 資料結構

儲存盤面的方式和 IDS 相同，但 limit 的方式改變了。例如：可以看到第一層紅色的字，g 代表已消滅細胞數，因此在第一層都是 0（還未消滅任何細胞），h 代表選取這個消滅節點後，衍生的癌細胞數量，所以在第一層 [1] 的位置，因為若選擇消滅了 [1] 的癌細胞，衍生出來的癌細胞仍然有 3 顆，故 $h = 3; f = g + h = 0 + 3 = 3$，若選擇消滅 [2] 的癌細胞，衍生出來的癌細胞有 1 顆，故 $h = 1; f = g + h = 0 + 1 = 1$，以此類推。最初我將 f - limit 設為 0，因此每個節點都沒辦法往下搜尋，接著 IDA* 會將 f - limit 更新為 1，因為 1 是目前最小的，接著繼續往下搜尋。



- 技術

  此支程式的核心技術分別為三個函式，主要遞迴的方式和IDS相同，以下詳細介紹

  - `iterative_deepening_dfs(input_list, n)`

    傳入參數為最初始的輸入測資 list，以及盤面大小，這個函式主要在控制 limit_f大小，例如在 limit_f 為 0 時遞迴下去找發現沒有解，就會回傳到這裡，之後此函式就會將 limit_f 更新為 next_f，再做下一次的搜尋。

    函式程式碼：

```
# 控制每個搜尋的f-limit
def iterative_deepening_dfs(input_list, n):
    limit_f = 0 # 初始f的limit設為0
    next_f = 100000 # 將next_f設為一個很大的數字
    while True:
        move = []
        dic = {}
        for i in range(300):# 初始字典各深度list，為了紀錄該層相同盤面
            dic[i] = []
        result = depth_limited_dfs(input_list, 0, n, move, dic,
limit_f, next_f)
        if result[0]:
            return result[1] # 若有解則回傳移動步驟
        limit_f = result[1] # 若沒找到解則把limit_f限制設為next_f，再傳
入遞迴，這裡的result[1]是next_f
```

- `depth_limited_dfs(input_list, depth, n, move, dic, limit_f, next_f)`

  主要在做遞迴的部分，傳入參數為每次遞迴輸入的盤面，目前深度，盤面大小，紀錄走步移動之 list，紀錄走過節點之 dictionary，目前的 limit_f，下一次的next_f。

  接著計算 Heuristic funtion， g 函式，並用以下公式計算 f：

  $$f(n) = g(n) + h(n)$$

  這個函式主要有三個判斷，第一個是判斷傳入的盤面上是否還有癌細胞，如果沒有，表示找到解法，則回傳成功（True）以及走步資訊，如果還有癌細胞就會去判斷目前的深度是否超過 limit_f，若超過就更新 next_f，最後若都沒有可搜尋的點就回到上一層 `iterative_deepening_dfs` 讓 limit_f 更新為 next_f，若目前 f 沒有超過 limit_f 則會用一個迴圈去選每一個癌細胞，並利用 `cell_grow` 來生成選了該位置的癌細胞後衍生的結果，再把此結果繼續傳入此支函式 `depth_limited_dfs` 做遞迴。在這個函式中，一樣有做兩個處理使其效率較佳：

  1. 在選擇要消滅哪個癌細胞時，只會點選有癌細胞的部分
  2. 在同一層 limit 若有重複相同的盤面不會再傳下去做遞迴

  詳細優化的做法在 IDS 有詳細說明。

  函式程式碼：

```
# 主要遞迴部分
def depth_limited_dfs(input_list, depth, n, move, dic, limit_f,
next_f):
    # 設置一個global變數紀錄next_f
    global nf
    nf = next_f
    if dic[depth].count(input_list) == 0: # 若此盤面在該層還會走過，將它
加入字典
        dic[depth].append(input_list)

    # 計算f = g + h
    g = depth - 1
    h = sum(input_list)
    f = g + h

    depth += 1
    total = sum(input_list) # 計算盤面癌細胞數量
    if total == 0: # 若為0表示已完全消滅，則回傳True以及移動步驟
        return (True, move)
    elif f > limit_f and depth > 1: # 若f已經超過限制，就更新next_f，並
回傳False及next_f
        next_f = min(next_f, f) # 取最小的next_f，同時一併更新global nf
        nf = next_f
        return (False, next_f)
    else:
        for i in range(n): # 選取要消滅細胞，有n種可能
            if input_list[i] != 0: # 有癌細胞的地方才可以選
                m = move[:]
                m.append(i+1) # 紀錄走步
                new_cell = cell_grow(input_list, i) # 處理衍生細胞，傳
回新的盤面
                if dic[depth].count(new_cell) == 0: # 判斷該盤面在該層
是否展開過
                    result = depth_limited_dfs(new_cell, depth, n,
m, dic, limit_f, nf) # 沒有展開過就傳入遞迴
                    if result[0] == True:
                        return (True, result[1]) # 找到解法就回傳
True，以及走步
    return (False, nf)
```

- `cell_grow(input_list, index)`

  該函式主要在處理消滅一個癌細胞後衍生的癌細胞盤面，傳入參數為消滅前盤面以及選擇消滅的位置。這裡也有經過優化處理，使其效率更佳，優化的方式和IDS 相同，在 IDS 有詳細說明。

以上為實作 IDA* 時用到的技術以及一些特殊處理，除了和 IDS 相同的優化處理外，因為多了 Heuristic funtion，在每次搜尋時都會選擇更有可能的節點去訪問，因此又更加有效率。

- 表現

  使用標靶遊戲中四個測資的表現如下：

盤面	執行時間
[1, 1, 1]	0.000066 (s)
[1, 1, 1, 1]	0.000091 (s)
[1, 1, 1, 1, 1]	0.000246 (s)
[1, 1, 1, 1, 1, 1, 1, 1, 1]	0.011418 (s)

  和 IDS 做比較：（以下為 IDS 結果）

盤面	執行時間
[1, 1, 1]	0.000118 (s)
[1, 1, 1, 1]	0.000612 (s)
[1, 1, 1, 1, 1]	0.001661 (s)
[1, 1, 1, 1, 1, 1, 1, 1, 1]	0.063150 (s)

  盤面較大的資料比較：

演算法	盤面大小 (括號內為測資編號)	執行時間	盤面大小 (括號內為測資編號)	執行時間
IDS	20 (11)	20.815766 (s)	30 (13)	1015.602414 (s)
IDA*	20 (11)	1.630926 (s)	30 (13)	42.652381 (s)

  可以很明顯地看到，當盤面大小越大，他們的執行時間差異越明顯，在盤面大小為20時，使用 IDS 需要執行約 20 秒，但 IDA 卻只需要約 1 秒就可以完成，約為 20 倍的差距。在盤面大小為 30 時，使用 IDS 需要執行約 1015 秒，但 IDA 卻只需要約43 秒就可以完成，約為 23 倍的差距。由此可見 IDA* 效率較好。

舉例 ：

輸入測資編號(4)的測資，執行`python3 IDASTAR.py`，我們可以得到輸出結果為

```
Total run time = 0.011418819427490234 seconds.
An optimal solution has 14 moves:
2 3 4 5 6 7 8 2 3 4 5 6 7 8
```

輸入測資編號(6)的測資，執行`python3 IDASTAR.py`，我們可以得到輸出結果為

```
Total run time = 0.002377033233642578 seconds.
An optimal solution has 9 moves:
6 3 2 2 3 4 5 6 7
```

輸入測資編號(9)的測資，執行`python3 IDASTAR.py`，我們可以得到輸出結果為

```
Total run time = 0.21763014793395996 seconds.
An optimal solution has 20 moves:
6 9 10 11 12 13 14 2 3 4 5 6 7 8 9 10 11 12 13 14
```

輸入測資編號(11)的測資，執行`python3 IDASTAR.py`，我們可以得到輸出結果為

```
Total run time = 1.6309261322021484 seconds.
An optimal solution has 29 moves:
5 10 11 12 13 14 15 16 17 18 19 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5
4 3 2
```

輸入測資編號(13)的測資，執行`python3 IDASTAR.py`，我們可以得到輸出結果為

```
Total run time = 42.652381896972656 seconds.
An optimal solution has 43 moves:
1 16 17 18 19 20 21 22 23 24 25 26 27 28 29 29 28 27 26 25 24 23 22 21
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2
```

輸入測資編號(14)的測資，執行`python3 IDASTAR.py`，我們可以得到輸出結果為

```
Total run time = 55.70100998878479 seconds.
An optimal solution has 56 moves:
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 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7
6 5 4 3 2
```

輸入測資編號(15)的測資，執行`python3 IDASTAR.py`，我們可以得到輸出結果為

```
Total run time = 493.45859932899475 seconds.
An optimal solution has 69 moves:
32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9
8 7 6 5 4 3 2 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
```

輸入測資編號(16)的測資，執行`python3 IDASTAR.py`，我們可以得到輸出結果為

```
Total run time = 581.2950339317322 seconds.
An optimal solution has 76 moves:
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 39 38 37 36 35 34 33 32 31 30 29 28
27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2
```

輸入測資編號(17)的測資，執行`python3 IDASTAR.py`，我們可以得到輸出結果為

```
Total run time = 5312.870357036591 seconds.
An optimal solution has 96 moves:
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 49 48
47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24
23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2
```

輸入測資編號(18)的測資，執行`python3 IDASTAR.py`，我們可以得到輸出結果為

```
Total run time = 29562.276843070984 seconds.
An optimal solution has 116 moves:
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 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44
43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2
```

- 耗用的時間及空間
  - 時間複雜度

    這支程式使用的是迭代加深深度優先搜尋演算法，其時間複雜度取決於搜尋深度的上限以及每個節點的擴展次數。每個節點最多擴展 n 次（n 是給定的盤面長度），因此時間複雜度為 $O(n^d)$，其中 d 是搜尋的深度。但是，在此實現中，使用了一個字典來存儲已訪問的節點，從而避免了訪問已經訪問過的節點，這樣可以大大減

少節點數，從而減小時間複雜度。且因為此算法有加入Heuristic funtion，會盡量往最有可能有解的地方去訪問。因此，實際的時間複雜度可能比 $O(n^d)$ 小得多，取決於實際搜索的節點數。

- 空間複雜度

  這個程式的空間複雜度取決於兩個因素：搜尋深度的上限和每個節點擴展出的節點數量。

  在這個程式中，我們使用了一個字典 dic 來儲存已經訪問過的節點，並且在每個深度上獨立使用一個 list 來存儲已經訪問過的節點，因此空間複雜度與搜尋深度的上限有關，即為 $O(dn)$，n 是給定的盤面長度，d 是搜尋的深度。

  此外，在每個節點擴展時，都會創建一個新的 cell 列表來代表新的節點，因此空間複雜度與每個節點擴展出的節點數量有關，即為 $O(n)$。

  input_list 儲存了輸入數據，是一個長度為 n 的整數列表，佔用 $O(n)$ 的空間

  因此，綜合考慮這些因素，這個程式的空間複雜度為 $O(dn)$，不過由於有使用一個字典來儲存已訪問的節點，從而避免了訪問已經訪問過的節點，這樣可以大大減少節點數，且還有加入Heuristic funtion，因此實際的空間複雜度也可能比 $O(dn)$ 小得多。

- 能解到多大盤面

  目前嘗試到盤面大小為60的測資，其執行時間如下表：

  實際執行時間會因測資不同，跑的節點數不同而不同。

盤面大小（括號內為測資編號）	**執行時間**
20 (11)	1.630926 (s)
30 (13)	42.652381 (s)
40 (15)	493.458599 (s)
50 (17)	5312.870357 (s)
60 (18)	29562.276843 (s)

6. 說明此次作業碰到的狀況及困難

   在此次作業中，主要遇到三個比較大的困難，分別為：

1. 沒有寫過 IDS 演算法，因此花了一點時間在理解 IDS 演算法，理解了單純的演算法過後，起初不太知道該如何用這個演算法套用到我們要的標靶治療題目上，解決方法是我將細胞分裂的過程、選擇消滅哪一顆癌細胞會衍生出哪種盤面畫出來，慢慢理解與消化，最後成功寫出了使用 IDS 演算法的解法。

2. 第二個問題是當盤面越來越大會有跑太久的問題，因為每種盤面一直展開下去，最一開始寫的 IDS（沒有任何優化，重複的盤面會繼續展開），盤面大小為 9 的跑到大約 limit 7 資料量就會非常大，為了實驗我讓它跑了一整晚，隔天大約能跑到大約limit 10 ，後來因為資料量實在太大沒有繼續跑。為了解決此問題，花了很多時間在看程式碼哪個部分有辦法優化或是紀錄相同的盤面，後來也有成功找到，詳細的優化方法在上面 IDS 技術部分有說明。

3. Heuristic funtion 的問題，花了蠻多時間在思考該如何決定 Heuristic funtion，經過許多嘗試後，決定將 Heuristic funtion 設為選取消滅該顆癌細胞後，衍生的癌細胞數，所以 limit f 就將它設為目前已消滅細胞數加上選取消滅該顆癌細胞後，衍生的癌細胞數，這樣得出來的解一定會是最佳解，因為我們每次走的都會是總消滅數最少的選擇。

我認為此次作業雖然需要花費很多時間來完成，但透過這個作業學到很多，其中也有遇到一些大大小小的問題，例如衍生細胞的處理、遞迴參數的傳遞與運作，透過這次作業也對一些觀念更加清楚了。

7. 參考文獻來源

此部分只有參考純 IDS 演算法概念與寫法，理解過後自己依照題目需求重寫，因此沒有用在程式中哪個部分。

- ChatGPT : IDS 演算法、python 語法
- 迭代深化深度優先搜尋 - 維基百科，自由的百科全書
- Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS) - GeeksforGeeks

8. 額外加分

在某個n範圍內，證明所有初始盤面都有解：

目前嘗試到n = 10，所有盤面均有解，但礙於資料量太大沒有交上來，可以見 `bonus.txt` 為盤面大小為5的所有種可能，以及其相對應的最佳解，可以看到每種初始盤面均有解。使用的方法是利用程式跑所有初始盤面得到所有最佳解。

殘局庫：我們可以將上述每種初始盤面的最佳解紀錄起來，存入殘局庫，這樣以後遇到這種情形的盤面，就可以直接抓答案，提升效率。