Subset

問題描述

Given an integer array nums of unique elements, return all possible subsets (the power set).

The solution set must not contain duplicate subsets. Return the solution in any order.

Example 1:

Input: nums = [1,2,3]
Output:
[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

圖一 輸入範例1

Example 2:

Input: nums = [0]
Output: [[],[0]]

圖二 輸入範例2

Constraints:

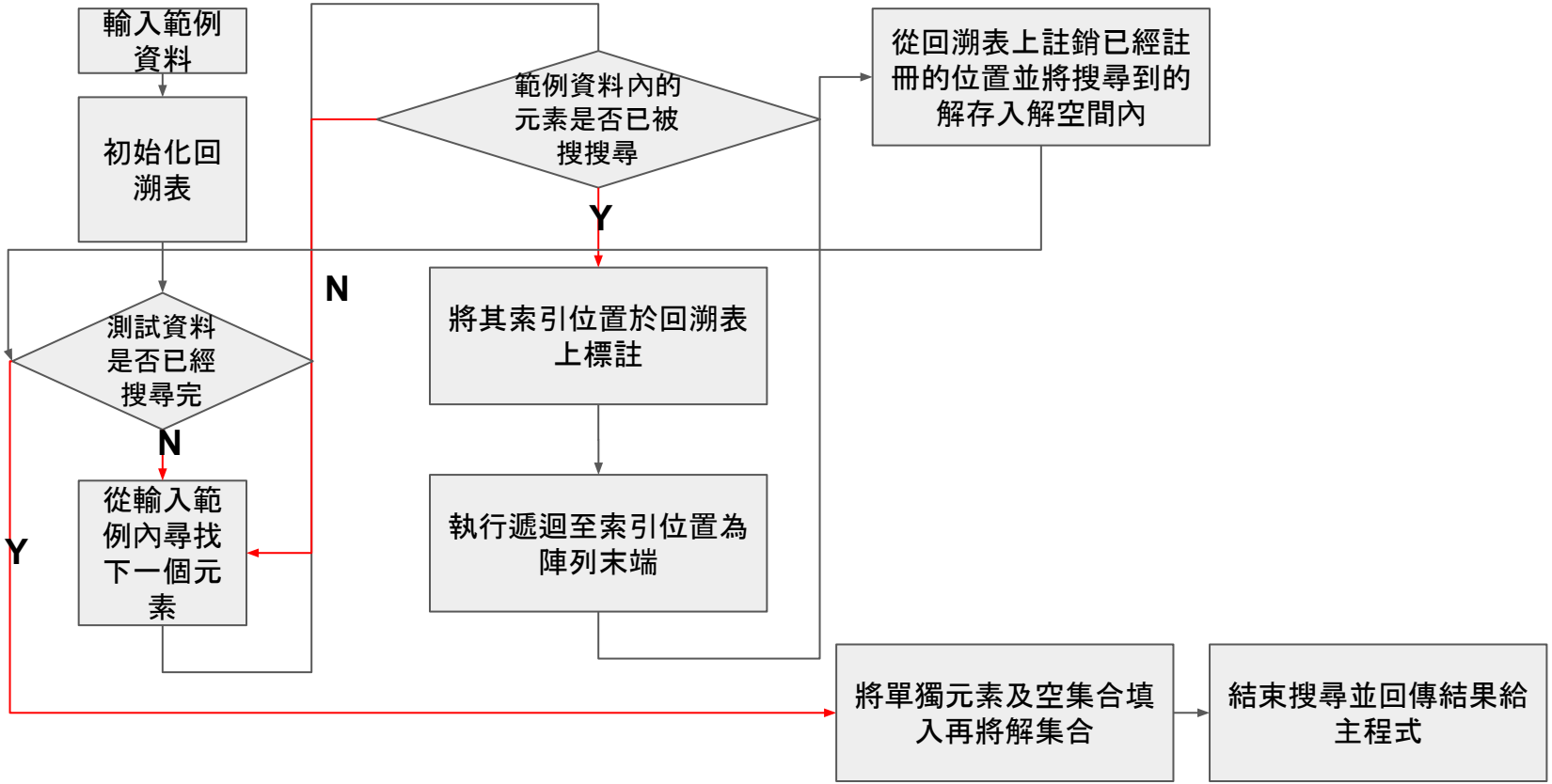
1 <= nums.length <= 10
-10 <= nums[i] <= 10
All the numbers of nums are unique.

圖三 輸入範例3

解題思路

- 我想這樣說的，毫無爭議的這題是一題演算法題，論窮舉我想Backtracking是一個好的選擇

原先程式邏輯


輸入範例資料
初始化回溯表
測試資料是否已經搜尋完
N
從輸入範例內尋找下一個元素
Y
範例資料內的元素是否已被搜搜尋
Y
N
將其索引位置於回溯表上標註
執行遞迴至索引位置為陣列末端
從回溯表上註銷已經註冊的位置並將搜尋到的解存入解空間內
將單獨元素及空集合填入再將解集合
結束搜尋並回傳結果給主程式

設計上的問題

- git上面有Code，可以看history有v1及v2的差別。這邊就點有修改的關鍵點。

```
public void doWork(int[] nums,List<Integer>
list,List<List<Integer>>result){
    for(int i=0;i<nums.length;i++){
        if(table[i]==0) {
            table[i] = 1;
            list.add(nums[i]);
            doWork(nums, list,result);
            table[i] = 0;
            list.sort((Integer o1, Integer o2)-> o1-o2);
            if (!result.contains(list)){
                result.add(list);
            }
            list=new ArrayList<>();
        }
    }
}
```

圖四 原版程式

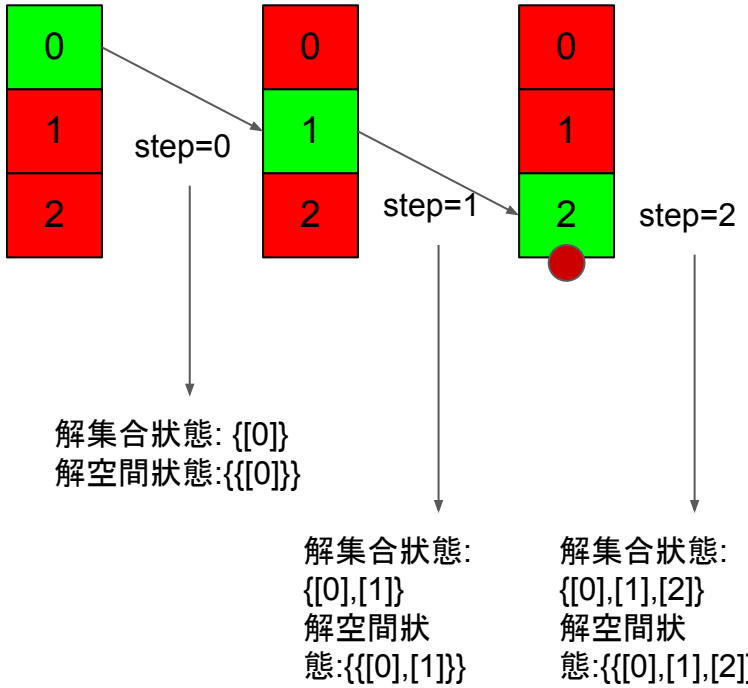
```
public void doWork(int[] nums,int step,List<Integer>
list,List<List<Integer>>result){

    ArrayList<Integer> cpy_list=new ArrayList<Integer>();
    cpy_list=(ArrayList<Integer>)((ArrayList<Integer>)list).clone();
    result.add(cpy_list);

    for(int i=step;i<nums.length;i++){
        list.add(nums[i]);
        doWork(nums, i+1,list,result);
        list.remove(list.size()-1);

    }
}
```

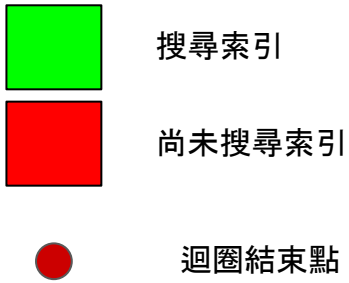
圖五 新版程式

錯誤檢討

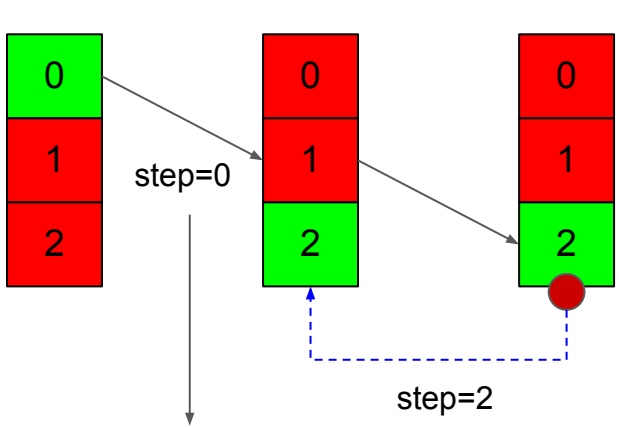
- 舊版的程式中包含了大量的重複搜尋，在處理測試資料矩陣長度為 10的時候會消耗掉大量的時間。
  - ➔ ex:[2,1,3] ->[1,2,3]為相同的解集合，這個原因是因為，每一代都當成獨立的一個階段，原版的backtracking只是依賴回溯表(參數名稱為table)檢驗到這次沒有出現過的然後再一次的搜尋。
- 新版的程式中將將index也作為參數傳遞至下一個階段，所以搜尋的 loop不需要再次的搜尋已經搜尋過的矩陣元素，所以也沒有回溯表的設計。

新版backtracking執行流程-1



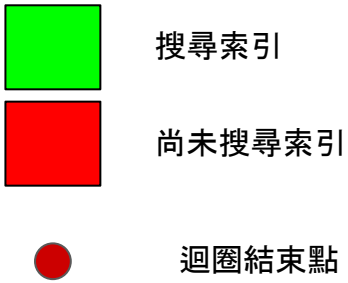


新版backtracking執行流程-2

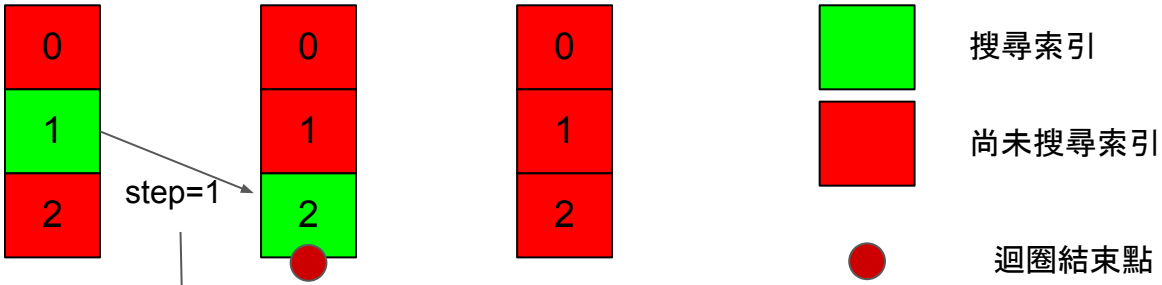


解集合狀態: {[0]}
解空間狀態:{{[0]}}

解集合狀態:
{[0],[2]}
解空間狀
態:{{[0],[1],[2]},
{[0],[2]}}



新版backtracking執行流程-3

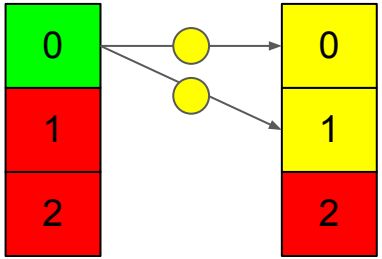


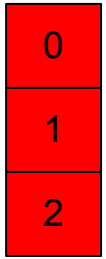
解集合狀態: {[1]}
解空間狀態:{{[0],[1],[2]},{[0],[2]},{[1]}}

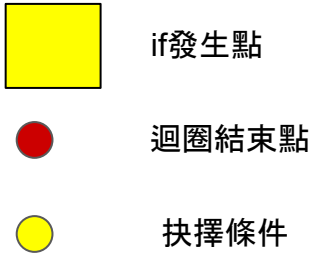
step=2

解集合狀態:
{[1],[2]}
解空間狀態:{{[0],[1],[2]},{[0],[2]},{[1]},{[1],[2]}}

舊版backtracking執行流-1



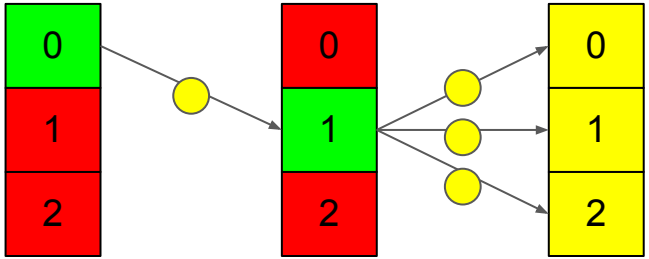




第一次呼叫

table[0]=1
解集合狀態: {[0]}
解空間狀態:{{[0]}}

舊版backtracking執行流-2

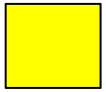


第一次呼叫

table[0]=1
解集合狀態: {}
解空間狀態:{}

第一次呼叫

table[0]=1,table[1]=1
解集合狀態: {}
解空間狀態:{}

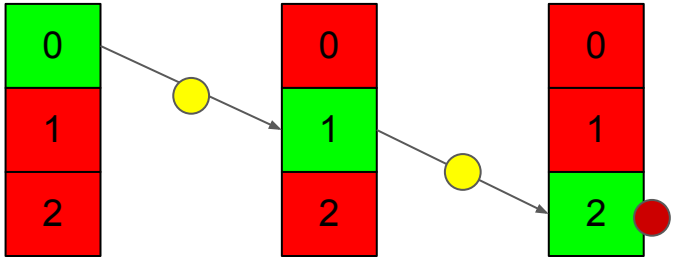


if發生點

迴圈結束點

抉擇條件

舊版backtracking執行流-3



第一次呼叫

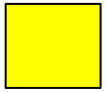
table[0]=1
解集合狀態: {}
解空間狀態:{}

第一次呼叫

table[0]=1,table[1]=1
解集合狀態: {[0],[1]}
解空間狀態:{}

table[0]=1,table[1]=1
table[2]=1

解集合狀態: {[0],[1],[2]}
解空間狀態:{{[0],[1],[2]}}

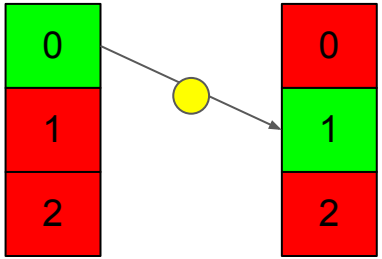


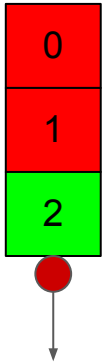
if發生點

迴圈結束點

抉擇條件

舊版backtracking執行流-3



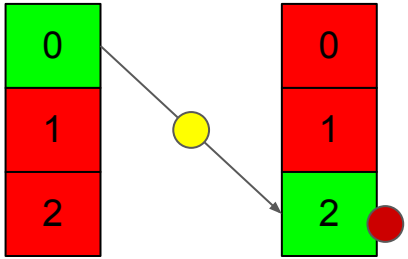


table[2]=0
解集合狀態: {} ← new list
解空間狀態:{{[0],[1],[2]}}

迴圈結束點

抉擇條件

舊版backtracking執行流-4



迴圈結束點

抉擇條件

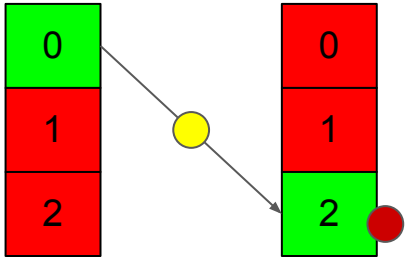
第一次呼叫

table[0]=1
解集合狀態: {[0]}
解空間狀態:{}

第二次呼叫

table[0]=1,table[1]=0
table[2]=1
解集合狀態: {[0],[2]}
解空間狀態:{{[0],[1],[2]},{[0],[2]}}

舊版backtracking執行流-4



第一次呼叫

table[0]=1
解集合狀態: {[0]}
解空間狀態:{}

第二次呼叫

table[0]=1,table[1]=0
table[2]=0
解集合狀態: {} ←
new list
解空間狀
態:{{[0],[1],[2]},{[0],[2]
}}

迴圈結束點

抉擇條件