Powerset 遞迴報告

41243103 林采儀

2024/10/18

1. 解題說明

給定一個包含 n 個元素的集合 S，請撰寫一個遞迴函數來計算 S 的所有子集，即 Powerset(S)。例如，如果 S = {a, b, c}，那麼 Powerset(S) = {(), (a), (b), (c), (a, b), (a, c), (b, c), (a, b, c)}。

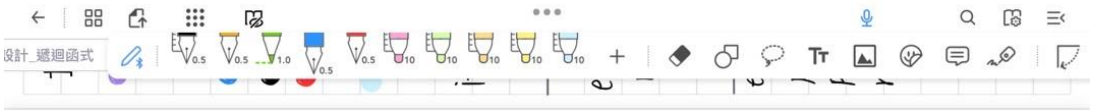
解法過程：

在研究解決此問題的方法後，我發現使用二進位表示來生成 Powerset 是一種直觀且高效的方法。這種方法的優點在於，它可以避免重複組合，並能保證列出所有的子集。

每個子集可以對應到一個 n 位的二進位數字，其中每個位元代表該子集是否包含對應的元素。這個概念與數位邏輯中的真值表非常類似，非 1 即 0，因此具有清晰的邏輯性與排列特性。

我一開始選擇了以 for 迴圈的方式來產生 powerset，這有助於我理解問題的本質。經過多次調整，才將其轉換為遞迴解法。

此外，我運用了一個生活中的例子來加深自己的理解：這類似於一個超市外送平台的訂單系統，每個子集就像是一張訂單，選擇不同的產品組合。



Bitmasking (位元掩碼法)

位元與數位邏輯真值表相似特性

① 可列出所有可能組合
② 2 進位表示，非 1 即 0
③ 必免組合重覆

powerset ⇒ 2^n ; n 為元素個數

n 元素個數即為有幾個位元

例: set={a, b, c}
n=3 ⇒ 3個元素、3個位元
$2^3=8$ ⇒ 8個子集合

c	b	a	子集合
0	0	0	{}
0	0	1	{a}
0	1	0	{b}
0	1	1	{a b}
1	0	0	{c}
1	0	1	{a c}
1	1	0	{b c}
1	1	1	{a b c}

初概念 ⇒ 迴圈

① 第一個 for(i=0; i<1000; ++i) ⇒ 外送的超市訂單 (1000的=進制)
從 000、001、010 ..... 111 ⇒ 每個都是一個訂單

② 第二個 for(j=0; j<3; ++j) ⇒ 店員依照訂單一個個對應貨架上商品
此的 j 是元素的陣列編號 set[0]、set[1]、set[2]

③ if(i&(1<<j)) ⇒ 若對應到訂單商品就將該商品放入袋子
i 和 j 做 and 邏輯運算

⇒ i=011 ; j=0 ; (1<<0)=001 ⇒ 011 & 001 = 001 ✓ set[0]

j=1 ; (1<<1)=010 ⇒ 011 & 010 = 010 ✓ set[1]

j=2 ; (1<<2)=100 ⇒ 011 & 100 = 000 X

因此店員用逐一比對方式正確找出訂單 i=011 的商品 ⇒ {a b}

(手稿)

2. 效能分析

在效能測試中，由於我一開始是用迴圈方式實現 powerset，後來改為遞迴，因此我將兩種方式的時間資源消耗進行了觀察比較。

時間複雜度：O(2^n)。

空間複雜度：O(2^n)（用於儲存子集），遞迴需要 O(n) 的堆疊空間。

遞迴版本：

- 當輸入集合較小時（例如 3 至 4 個元素），遞迴解法能夠有效且正確地列出所有子集，並且程式碼結構清晰易懂。
- 然而，隨著集合大小的增長（特別是超過 10 個元素），遞迴深度迅速增加，導致效能顯著下降。這是因為每次函式調用都會消耗堆疊空間，當集合元素數超過 12 時，甚至出現堆疊溢位的問題，導致程式無法正常運行。

for 迴圈版本：

- 相較於遞迴版本，for 迴圈的實現能更有效地處理大規模集合。由於其計算過程更為直接，能夠有效運用記憶體，並且不會受到遞迴深度的限制。
- 在測試中，for 迴圈版本在處理超過 10 個元素的集合時，仍能穩定運行並保持合理的執行時間。這使得其在處理大規模數據時具有明顯的優勢。

3. 測試與驗證

我選擇了不同大小的集合作為測試資料，這些測試資料的選擇旨在涵蓋從小型到大型的不同情境，以觀察兩種方法的性能表現和穩定性。

- **小型集合：** set[] = {'a', 'b', 'c'}
- **中型集合：** set[] = {'a', 'b', 'c', 'd'}
- **大型集合：** set[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'} 以及 {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l'}，以此類推到 14 個元素的集合。

以下是測試結果的摘要：

- **小型集合 (3 個元素)：**

  兩種版本均成功生成正確的冪集，且結果一致。遞迴版本的執行時間稍短。

- **中型集合 (4 個元素)：**

  兩種版本均成功生成正確的冪集，結果一致。遞迴版本的執行時間仍然短於迴圈版本。

- **大型集合 (11 到 14 個元素)：**

  - **遞迴版本：**在 11 和 12 個元素的集合中執行時間為 0.346 秒和 0.624 秒，但在處理 13 和 14 個元素的集合時，由於堆疊溢位而導致失敗。
  - **for 迴圈版本：**執行時間從 0.333 秒到 2.703 秒，隨著集合大小的增長而上升，且所有生成的冪集均正確。

- **測試結果顯示:**

  兩種方法在小型和中型集合中均能正確生成冪集。遞迴版本在時間上的表現普遍優於迴圈版本。但隨著集合大小的增長，遞迴版本在處理大於 12 個元素的集合時出現堆疊溢位問題。for 迴圈版本在處理大規模集合時顯示出更好的穩定性，無堆疊溢位的問題。

```
測資(11個) set[] ={ 'a', 'b', 'c','d','e','f','g','h','i','j','k' }
PowersetRecursion:
....
Elapsed time: 0.346304 seconds


測資(12個) set[] ={ 'a', 'b', 'c','d','e','f','g','h','i','j','k','l' }
PowersetRecursion:
....
Elapsed time: 0.624426 seconds


測資(13個) set[] ={ 'a', 'b', 'c','d','e','f','g','h','i','j','k','l','m' }
PowersetRecursion:
....
失敗
*/
```

遞迴測試結果

```
測資(11個) set[] = { 'a', 'b', 'c','d','e','f','g','h','i','j','k'}
PowersetIterative:
....
Elapsed time: 0.333289 seconds


測資(12個) set[] = { 'a', 'b', 'c','d','e','f','g','h','i','j','k','l' }
PowersetIterative:
....
Elapsed time: 0.770582 seconds


測資(13個) set[] ={ 'a', 'b', 'c','d','e','f','g','h','i','j','k','l','m' }
PowersetIterative:
....
Elapsed time: 1.37522 seconds


測資(14個) set[] ={ 'a', 'b', 'c','d','e','f','g','h','i','j','k','l','m','n' }
PowersetIterative:
....
Elapsed time: 2.70392 seconds
```

迴圈測試結果

4. 申論及心得

這次的作業讓我對兩種生成冪集的方法有了更深的了解：遞迴和迴圈。通過編寫和測試這兩種方法，我發現它們各有優缺點。

首先，遞迴方法在處理小型和中型集合時表現得非常好，執行速度也很快。這種方法的邏輯清晰，讓我能更輕鬆地理解問題的本質。但是，當集合變得較大時，遞迴深度會迅速增加，這會導致堆疊溢位，讓程序無法正常運行。雖然這種方法簡單明瞭，但在面對大量數據時，它的表現就不如預期了。

相比之下，迴圈方法在處理大集合時更加穩定，能夠有效利用記憶體，並且不會受到遞迴深度的限制。雖然它的實作相對複雜，但在處理大量資料時，能表現得更好，這讓我認識到在寫程式時，穩定性和效能也同樣重要。