摘要

本研究探討矩形中分割正方形的問題，主要目標是撰寫程式來解決此問題，並比較不同演算法的差異性。研究動機源於對圖形拆分與組合的興趣，並希望運用資料結構與演算法來解決相關問題。研究方法涵蓋六種不同的演算法：輾轉相除法、動態規劃、二維枚舉、一維枚舉、線段樹與線段樹搭配差分，並透過實驗比較各方法的正確性與執行效率。結果顯示，動態規劃兼具正確性與效率，而一維枚舉在能求得最佳解的方法中效率最高。此外，透過線段樹進行優化時，常數過大可能會影響效能。研究結論指出，未來若欲進一步優化執行時間，需克服常數過大的問題，以提升整體運算效率。

壹、前言

一、研究動機

小時候，我們喜歡玩七巧板，透過不同形狀的組合拼出各種圖案，讓我們對圖形的拆分與組合產生了濃厚的興趣。在學習過程中，我們接觸到了完美正方形（Perfect Squared Rectangles）與其一系列的問題，這類問題需要在矩形與正方形中切割出數個互不重疊的較小正方形，讓我們深感好奇。因此，我們希望運用學習到的資料結構與演算法解決在矩形中分割正方形的問題，並探討不同演算法對此問題的差異性。

二、研究目的

（一）撰寫能夠解決矩形中分割正方形的程式

（二）探討不同資料結構與演算法解決矩形中分割正方形的差異性

三、文獻回顧

（一）矩形中分割正方形

矩形中分割正方形問題如下：給定正整數 n 與 m，將長為 n ，寬為 m 的矩形（以下簡稱為 $n \times m$ 矩形）分割為數個邊長亦為正整數的正方形，使得矩形恰好被這些正方形分割完。求最佳的分割方法至少需要分割出多少個正方形。若將此問題的切割方式限制於水平或垂直切割一分為二，則此問題即為一經典的動態規劃題目，並且複雜度為 $O(nm(n+m))$ 。

進行本研究前，我們參考了歷屆科展的研究成果。我們參考了葉龍泉等人於 1989 年的的研究《矩形中分割方形》及其撰寫的程式。由於矩形的切割方式不一定僅限於水平或垂直

切割（如圖 1，13×11 矩形無法僅由水平或垂直切割得到最佳分割方式），該程式的運作方式為在矩形內的某個位置，嘗試枚舉所有可放入該位置的正方形，並在放置後繼續尋找下一個可放置的位置。雖然將程式改寫為 C++ 後處理邊長約為 30 到 40 的矩形需要至多 30 秒，但此方法仍能有效遍歷不同矩形切割的方式，為後續研究提供了重要的基礎。此外，此研究也將最佳切割法與輾轉相除法互相比較，並探討兩者的數學性質。

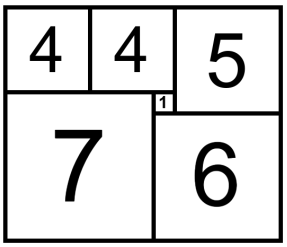


圖 1: 13×11 矩形的最佳分割方式（自行繪製）

（二）前綴和與差分

前綴和與差分是用於處理陣列區間操作的技巧。前綴和透過預處理，記錄每個位置之前所有元素的累加總和，使得查詢區間總和可以在 $O(1)$ 時間內完成。對於一個陣列 A，可以建立前綴和陣列 S，其中 $S[i]$ 表示從開頭到第 i 個元素的總和。查詢區間 $[l, r]$ 的總和時，只需計算 $S[r] - S[l-1]$，適合頻繁查詢但無更新的情況。

差分則是一種透過儲存變化量來進行區間修改的技巧。對於陣列 A，建立差分陣列 D，其中 $D[i]$ 表示 $A[i]$ 相較於 $A[i-1]$ 的變化量。這樣一來，對某個區間 $[l, r]$ 進行加法操作時，只需修改 $D[l]$ 和 $D[r+1]$，查詢時透過前綴和恢復陣列 A。區間修改的時間複雜度為 $O(1)$ ，適合頻繁修改但查詢較少的情況。

（三）線段樹

線段樹是一種用於處理區間查詢與區間更新問題的資料結構。線段樹將資料劃分為數個區間，每個區間對應一個二元樹節點（如圖 2），並儲存該區間的相關資訊。線段樹的基本操作包括：

1. 建樹：根據初始資料建立線段樹，複雜度 $O(n)$ 。

2. 查詢：查詢一個區間內的資訊，如總和、最大值或最小值，複雜度 $O(\log n)$ 。

3. 修改：對單點或區間內的數值進行修改，並更新對應的節點資訊，無論單點或區間修改，複雜度皆為 $O(\log n)$ 。

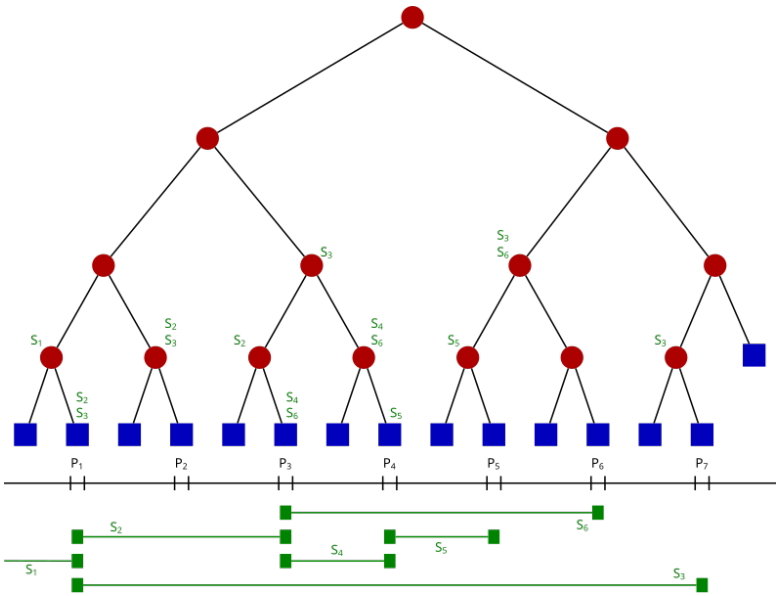


圖 2: 線段樹的結構（參考資料三）

貳、研究設備及器材

一、硬體：筆記型電腦。CPU：Intel Core i7-9750H。

二、軟體：Visual Studio Code 1.88.0。

參、研究過程或方法

一、研究流程

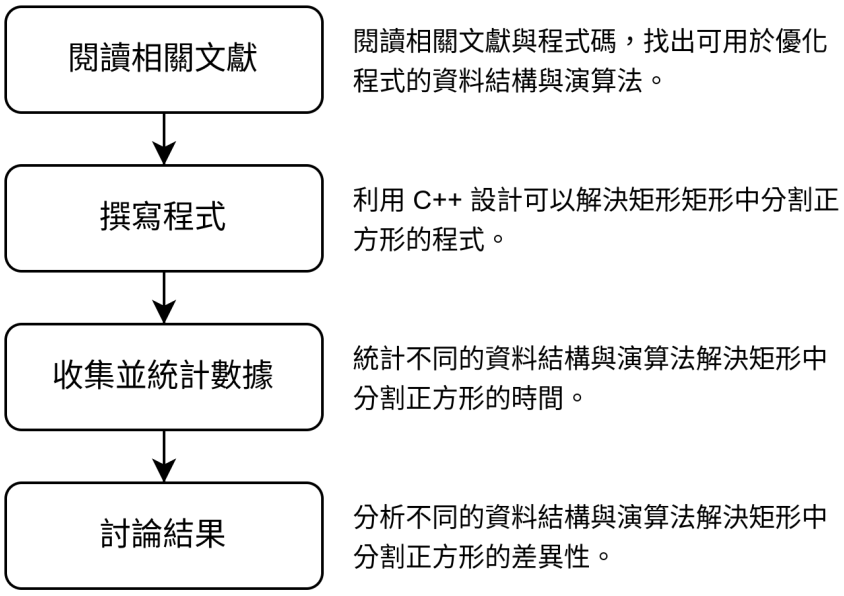


圖 3: 研究流程圖（自行繪製）

二、程式的選擇

本次研究中，我們總共選擇並撰寫了 6 支程式進行比較：

（一）輾轉相除法

每次都在矩形中切出邊長最大的正方形，若以 $F_1(n,m)$ 表示此方法得到的正方形個數，$F_1(n,m)$ 即為以輾轉相除法求 n 與 m 的最大公因數過程中所有商的總和。也因此 $F_1(n,m)$ 可以很容易地以遞迴實作（如圖 4）（其中 $n \bmod m$ 表示 n 除以 m 的餘數）。雖然此方法得到的正方形個數並不是最少的，但此方法執行速度極快，複雜度為 $O(\log \min(n,m))$，並且是一個具有系統的切割方式。

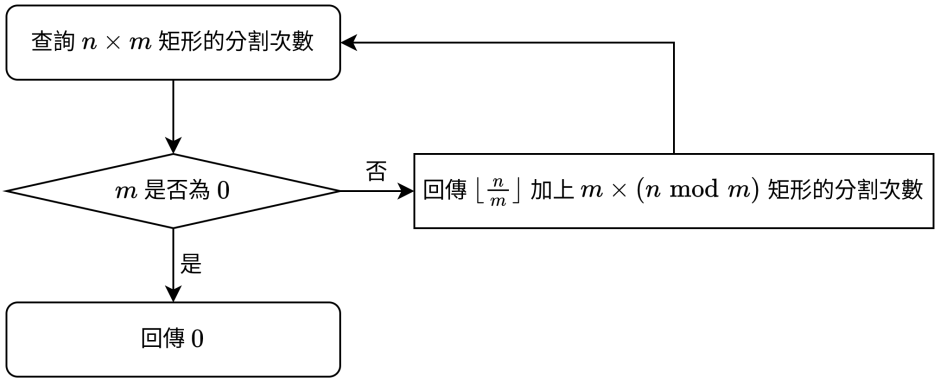


圖 4: 輾轉相除法流程圖（自行繪製）

（二）動態規劃

若將此問題的切割方式限制於水平或垂直切割一分為二，則此問題即為一經典的動態規劃題目。令 $F_2(i, j)$ 為 $i \times j$ 矩形切割出的最少正方形個數，則：

$$F_2(i,j) = \begin{cases} 1, \ i = j \\ \min(\min_{1 \le k < i}(F_2(k,j) + F_2(i-k,j)), \min_{1 \le l < j}(F_2(i,l) + F_2(i,j-l))), \ i \neq j \end{cases} \tag{1}$$

雖然此方法得到的正方形個數亦不是最少的，但此方法相對於下列可得到最佳解的程式仍然快上許多，並且較輾轉相除得到的結果佳，因此我們也將動態規劃用於初始化下列可得到最佳解的程式中的正方形數量最小值，複雜度為 $O(nm(n+m))$。

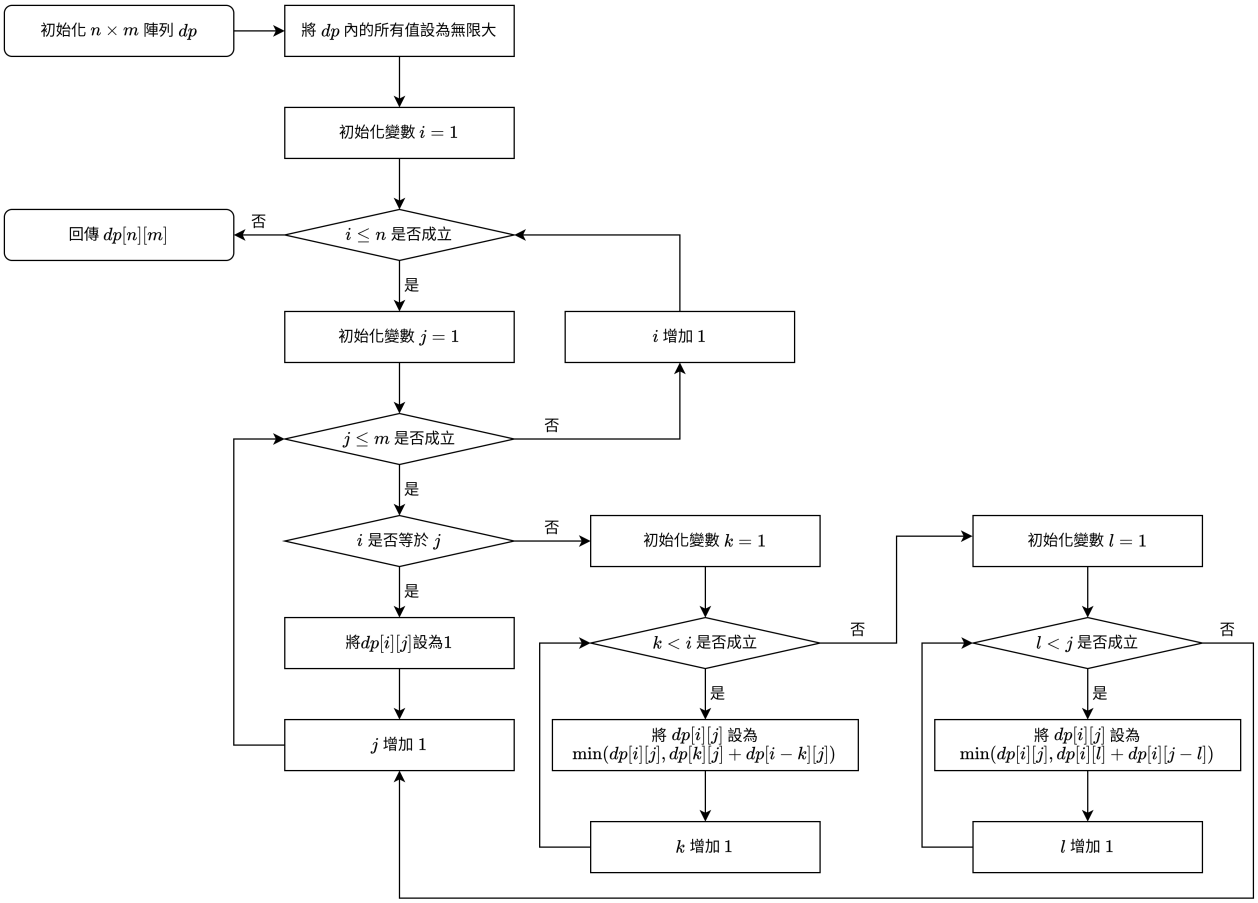


圖 5: 動態規劃流程圖（自行繪製）

（三）二維枚舉

此方法即為《矩形中分割方形》中提出的演算法，在介紹此方法前，首先將 $n \times m$ 矩形內的所有格子都訂定一座標，如圖 6。

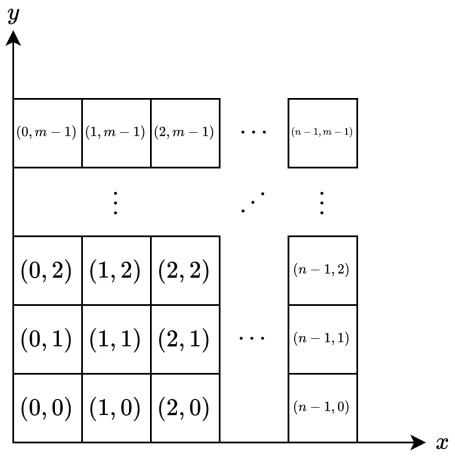


圖 6: $n \times m$ 矩形內的座標（自行繪製）

此演算法的運作方式為在矩形內的某個位置，嘗試枚舉所有可放入該位置的正方形，並在放置後繼續尋找下一個可放置的位置。

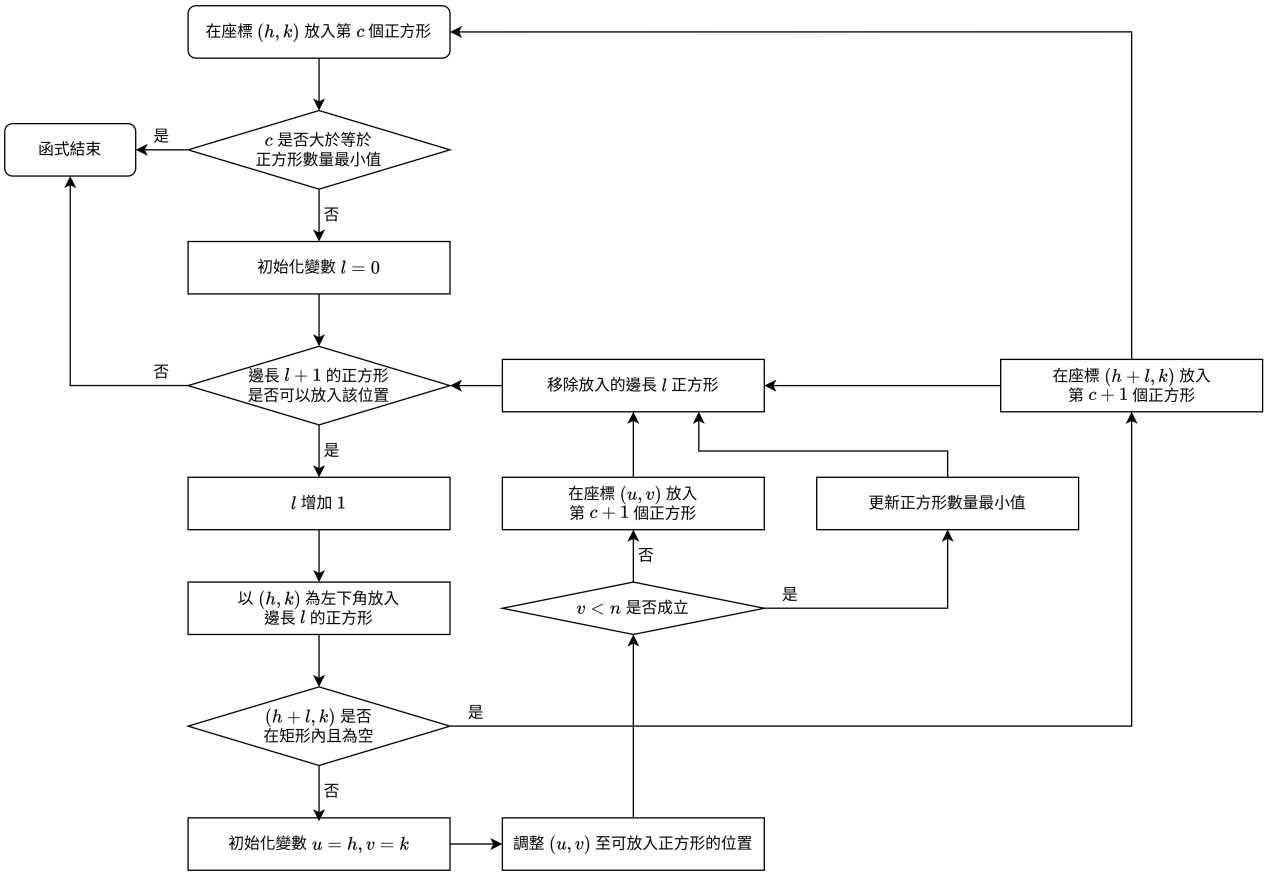


圖 7: 二維枚舉流程圖（自行繪製）

可以放入一位置的正方形表示不會發生 x 座標超過矩形範圍（圖 8）、y 座標超過矩形範圍（圖 9）、覆蓋到其他正方形（圖 10）三種情況，因此以 (h,k) 為左下角放入邊長為 $l+1$ 的正方形時必須滿足 $h+l<m$ 、$k+l<n$ 及 $(h+l,k)$ 為空。

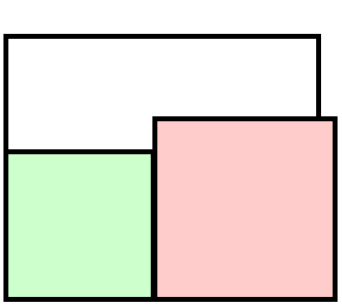


圖 8: x 座標超過矩形範圍

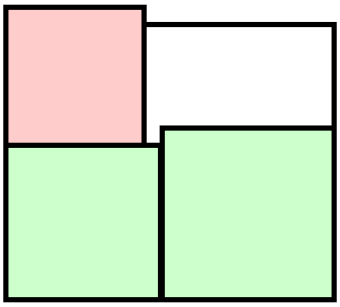


圖 9: y 座標超過矩形範圍

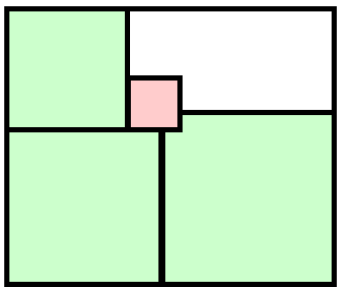


圖 10: 覆蓋到其他正方形

（皆為自行繪製）

將 $n \times m$ 矩形的座標一一對應到二維陣列 a ，填入一邊長 l 正方形的方式為選定正方形的左下角 (h,k) ，並將陣列中 $a[h][k]$ 到 $a[h+l-1][k+l-1]$ 的範圍內全部填入數字 l，而刪除此正方形的方式為在同一範圍填入數字 0。

尋找下一個可放置正方形的位置則分為兩種情況：如果放入的正方形的右方（如果以 (h,k) 為左下角放入邊長 l 的正方形，此座標為 $(h+l,k)$ ）在矩形內，則此位置即為下一個可放置正方形的位置；否則從 (h,k) 由下至上，由左至右逐列掃描，當掃描時的 x 座標超過矩形的範圍時就換行，直到找到空的座標，即為下一個可放置正方形的位置。若在矩形內找不到空的座標，則此正方形已經被填滿，此時更新更新正方形數量最小值。

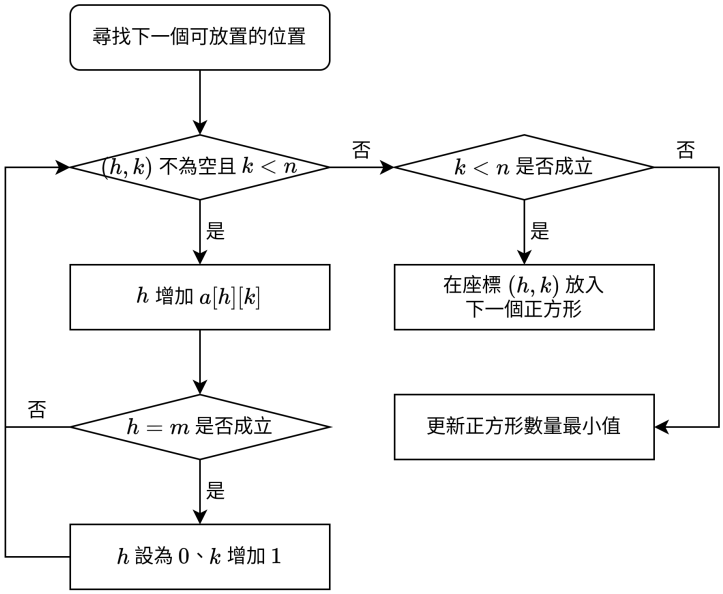


圖 11: 尋找可放置正方形位置流程圖（自行繪製）

（四）一維枚舉

由二維枚舉的程式，我們發現每次尋找下一個可放置正方形的位置時，因為尋找方式皆為由下至上，由左至右逐列掃描，所以找到的位置必定是矩形中 y 座標最小且為空的位置，如果有多個這樣的位置，取 x 座標較小者。因此，如果座標 (h,k) 不為空，那麼座標 (h,i) 對所有 $1 \le i < k$ 必定不為空，所以可以將原本的二維陣列化簡為一維陣列 a' ，而 $a'[h]$ 代表矩形中 x 座標為 h 處的高度。而原本的二維枚舉程式也有一些需要修改的部分。

首先，判斷正方形可否放入矩形時無法直接知道 $(h+l,k)$ 是否為空，此時可以改為判斷 $a'[h]$ 是否等於 $a'[h+l]$ 。圖 12 與圖 13 中的紅色座標代表 (h,k)，黃色座標代表 $(h+l,k)$。在圖 12 的情況下，(h,k) 為空，$(h+l,k)$ 亦為空，且已知 $a'[h]=k-1$ ，則 $a'[h+l]$ 必定小於等於 $k-1$ ，又因為 (h,k) 必為所有空的位置中，y 座標最小者，因此 $a'[h+l]$ 必定大於等於 $k-1$，所以 $a'[h]=k-1=a'[h+l]$。而在圖 13 的情況下，$(h+l,k)$ 不為空，因此 $a'[h+l]\ge k$，所以 $a'[h]=k-1\ne a'[h+l]$。

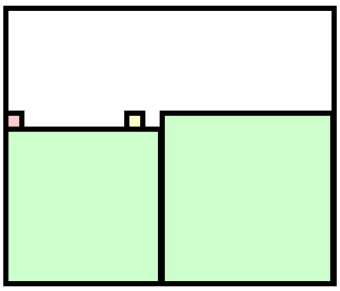


圖 12: $(h+l,k)$ 為空

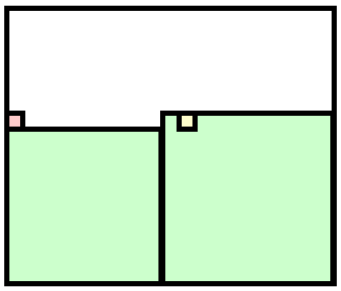


圖 13: $(h+l,k)$ 不為空

（皆為自行繪製）

如果 $a'[h]$ 到 $a'[h+l-1]$ 範圍中全部的數值皆為 $k-1$ ，那麼以 (h,k) 為左下角填入一邊長 l 正方形的方式為在 $a'[h]$ 到 $a'[h+l-1]$ 的範圍中全部加上數值 l。而若要尋找下一個可放置正方形的位置，只需要在整個 a' 陣列中找到最小值發生的位置 u 及最小值 v，(u,v) 即為欲尋找的位置。

（五）線段樹

由於一維枚舉程式中對 a' 的操作僅有單點詢問、區間加值、區間詢問三種操作，因此可以用線段樹進行優化。為了使以下流程圖簡潔，定義節點 id 的左子節點為 lc，右子節點為 rc。因本次研究詢問的區間必為整個線段樹，欲詢問的值位於根節點，因此沒有另外實作區間詢問的函式。

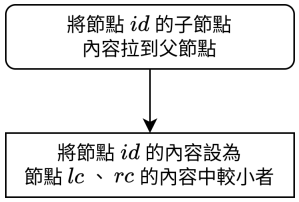


圖 14: 節點內容上拉（自行繪製）

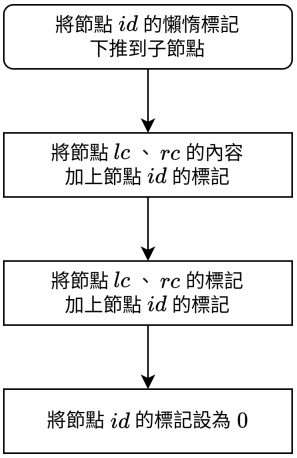


圖 15: 懶惰標記下推（自行繪製）

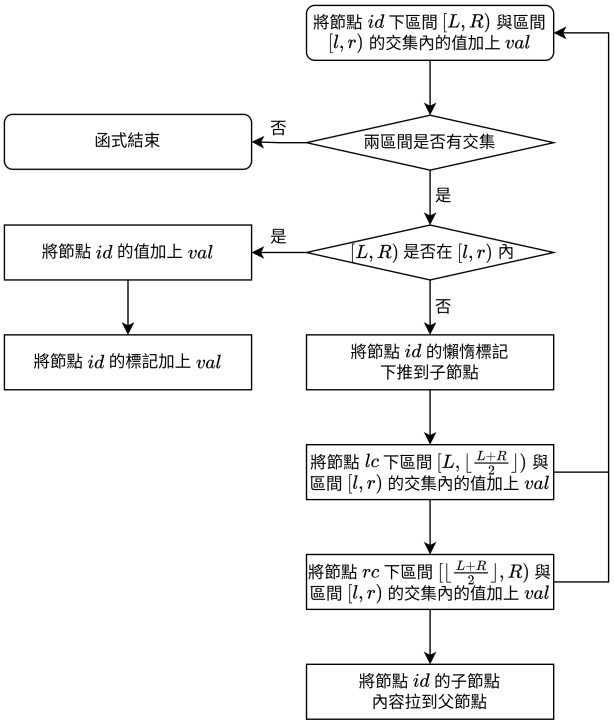


圖 16: 區間更新（自行繪製）

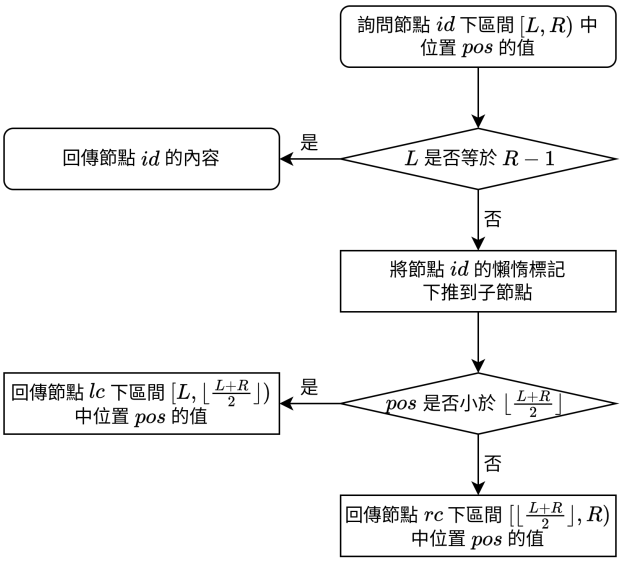


圖 17: 單點查詢（自行繪製）

因為本次研究在詢問區間時不僅需要知道區間的最小值，還需要知道最小值發生的位置，因此我們的作法為在每個節點的內容中放入兩個值，第一個值為節點本身的數值，第二個值為節點中的值所對應到的索引。在節點內容上拉時首先取節點本身的數值較小者，若都相同則取節點所對應到的索引較小者。也因此，雖然一開始線段樹內的所有值皆為 0 ，但仍然需要建樹。

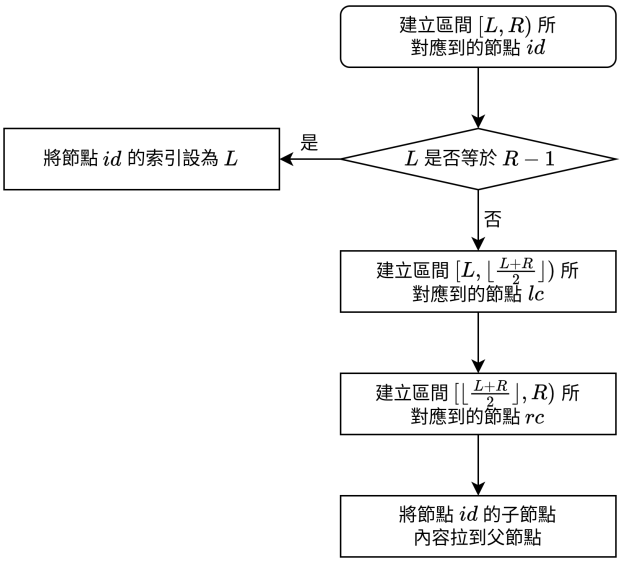


圖 18: 建樹（自行繪製）

將一維枚舉程式中單點詢問、區間加值、區間詢問的部分以線段樹取代，即完成線段樹的程式。

（六）線段樹搭配差分

線段樹的程式中，雖然更新與詢問區間都可以在時間複雜度 $O(\log n)$ 做到，但執行過程有過多單點詢問的部分，導致整體執行時間增加。我們發現單點詢問的順序依序為 $a'[h]$ 、$a'[h+1]$ 、$a'[h+2]\ldots$ ，因此可以在區間更新時紀錄 a' 的差分，另這個差分為 d ，詢問 $a'[h+l]$ 時因為已經知道 $a'[h+l-1]$ ，因此只要將 $a'[h+l-1]$ 加上 $d[h+l]$ 即可得到 $a'[h+l]$。

三、實驗的選擇

我們預計使用這 6 支程式做下列 3 個實驗。

（一）輾轉相除法與動態規劃解決此問題的正確性

雖然輾轉相除法與動態規劃兩種作法都不一定可以得到最佳解，但兩者仍是具有系統的切割方式，因此我們想將這兩者的解與最佳解互相比較。OEIS 數列 A219158 即提供了邊長為 1 ~ 388 的所有正方形最少需分割出的正方形數量。

（二）程式的執行效率

比較動態規劃、二維枚舉、一維枚舉、線段樹、線段樹搭配差分程式執行的效率。因可以預期輾轉相除法的執行速度解決不同矩形的分割時幾乎相同，因此不加以比較。而除了動態規劃外的程式複雜度皆難以估計，因此我們決定以執行的時間做比較。

（三）以長邊為 x 軸執行時間與以短邊為 x 軸執行時間的比較

比較一維枚舉程式中以長邊為 x 軸的執行時間與以短邊為 x 軸的執行時間。我們會選用「程式的執行效率」實驗中執行速度最快的程式進行此實驗。

肆、研究結果

一、輾轉相除法與動態規劃解決此問題的正確性

我們收集了輾轉相除法、動態規劃兩種作法在矩形邊長為 1 ~ 30 時得到的分割次數。因三維的圖表難以呈現結果，我們將矩形以 OEIS 數列 A219158 相同的編號方式編號：1×1 矩形編號 1、2×1 矩形編號 2、2×2 矩形編號 3、3×1 矩形編號 4...，並附上矩形長邊對應的編號區間（表 1），並且在以下的所有圖表中，將矩形長邊為偶數的區域顏色加深，以便觀察。最後我們決定呈現「輾轉相除法結果除以最佳解」（圖 19）與「動態規劃結果除以最佳

解」（圖 20）兩圖表。

矩形長邊	對應的區間	矩形長邊	對應的區間	矩形長邊	對應的區間
1	[1, 1]	11	[56, 66]	21	[211, 231]
2	[2, 3]	12	[67, 78]	22	[232, 253]
3	[4, 6]	13	[79, 91]	23	[254, 276]
4	[7, 10]	14	[92, 105]	24	[277, 300]
5	[11, 15]	15	[106, 120]	25	[301, 325]
6	[16, 21]	16	[121, 136]	26	[326, 351]
7	[22, 28]	17	[137, 153]	27	[352, 378]
8	[29, 36]	18	[154, 171]	28	[379, 406]
9	[37, 45]	19	[172, 190]	29	[407, 435]
10	[46, 55]	20	[191, 210]	30	[436, 465]

表 1: 矩形長邊對應的區間

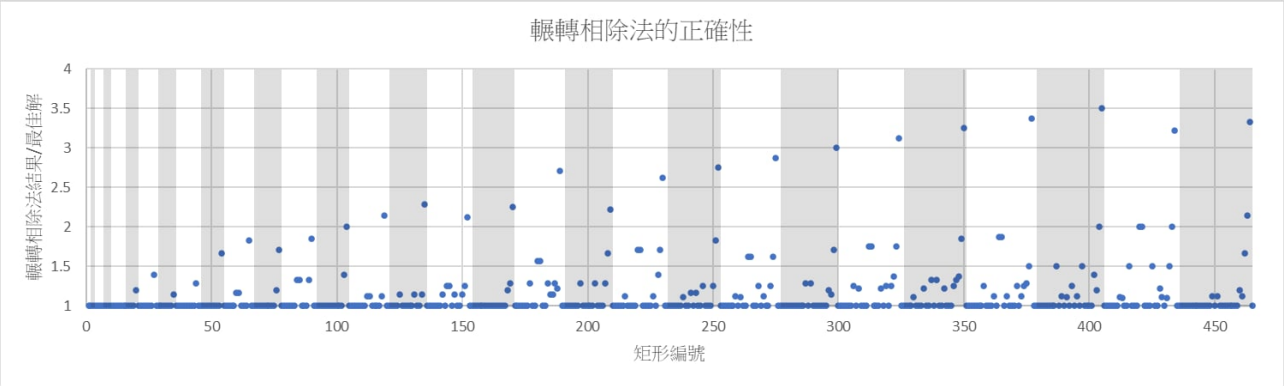


圖 19: 輾轉相除法的正確性（自行繪製）

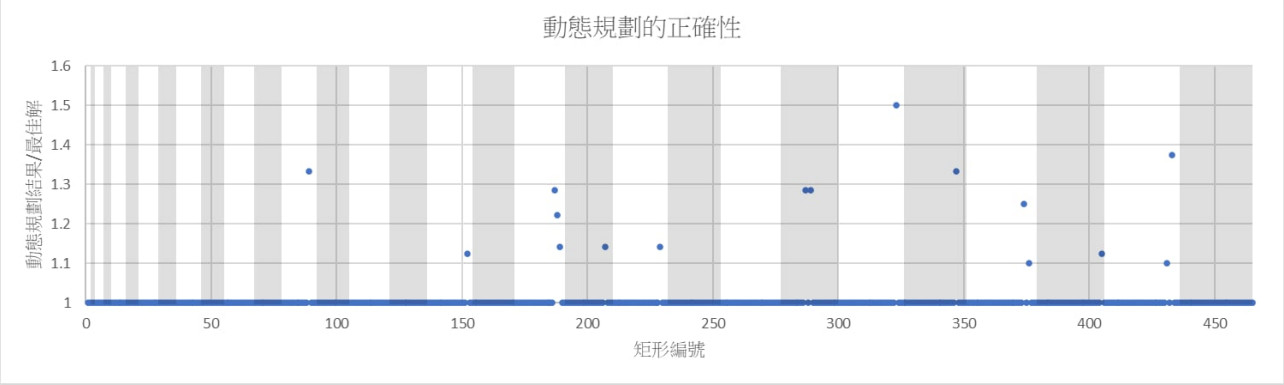


圖 20: 動態規劃的正確性（自行繪製）

二、程式的執行效率

以下五張圖分別為動態規劃、二維枚舉、一維枚舉、線段樹與線段樹搭配差分的執行時間，並同樣依照矩形的邊號排列。為了減少誤差，對於所有矩形，我們都執行 10 次程式，並且去除執行時間最長的兩次與最短的兩次後平均。

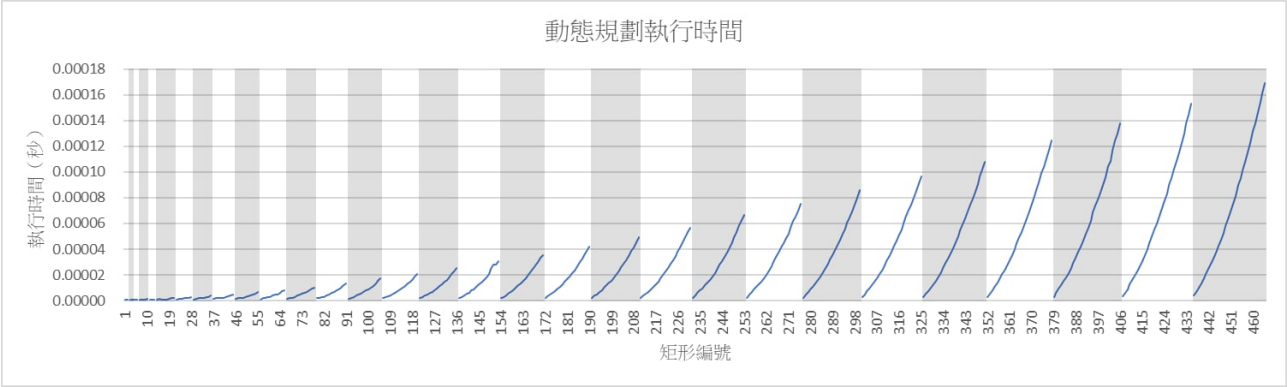


圖 21: 動態規劃執行時間（自行繪製）

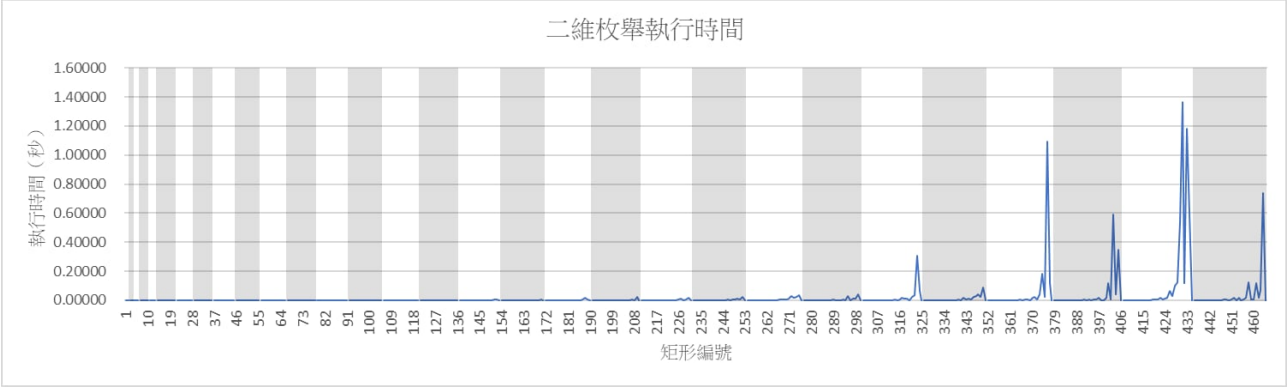


圖 22: 二維枚舉執行時間（自行繪製）

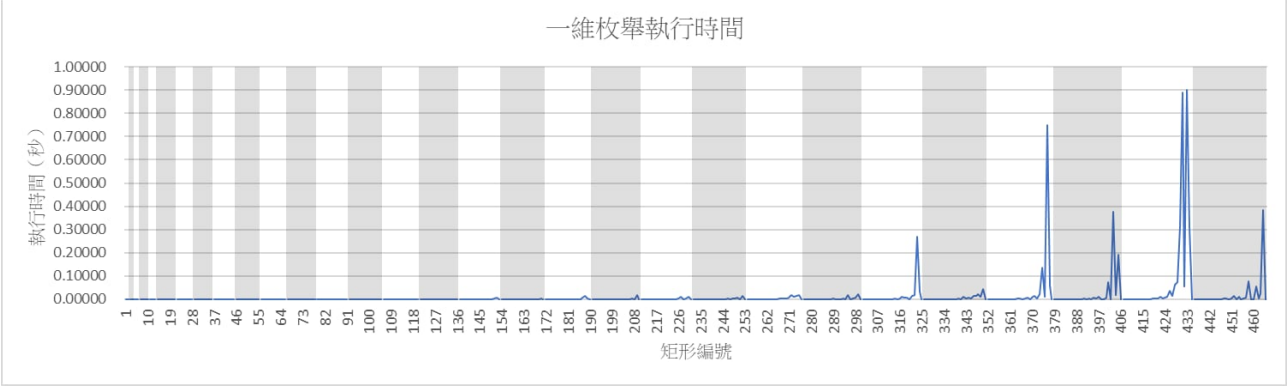


圖 23: 一維枚舉執行時間（自行繪製）

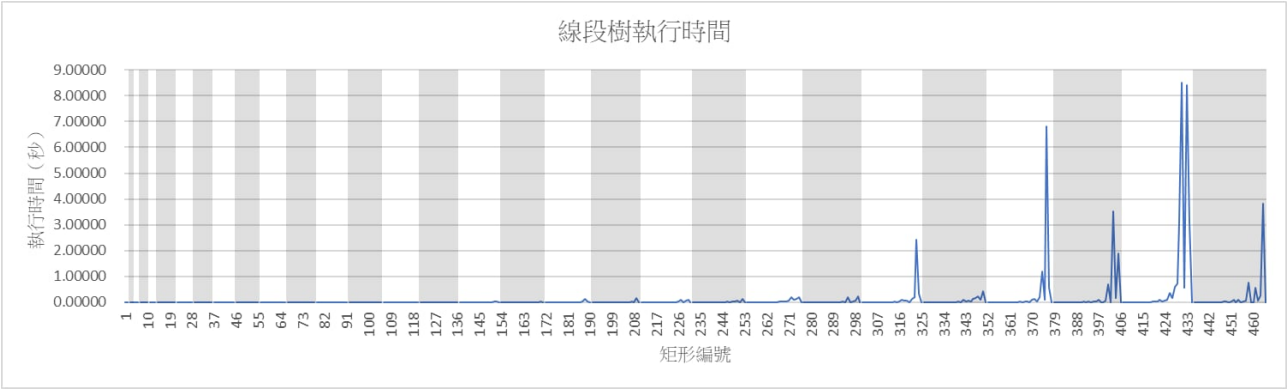


圖 24: 線段樹執行時間（自行繪製）

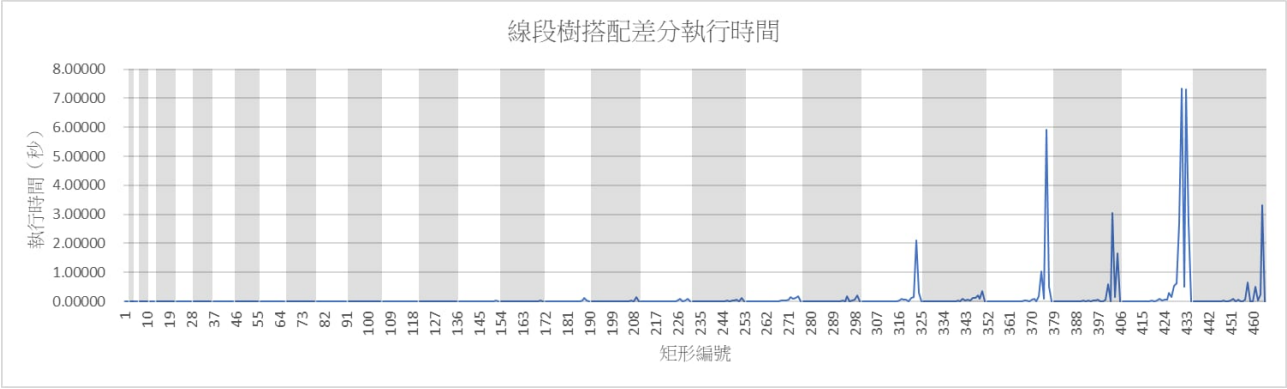


圖 25: 線段樹搭配差分執行時間（自行繪製）

另外，為了比較由二維枚舉轉變為一維枚舉的執行效率優化，以及由一維枚舉轉變為線段樹的執行效率優化，我們將一維枚舉執行時間: 二維枚舉執行時間與線段樹執行時間: 一維枚舉執行時間也繪製成圖表。

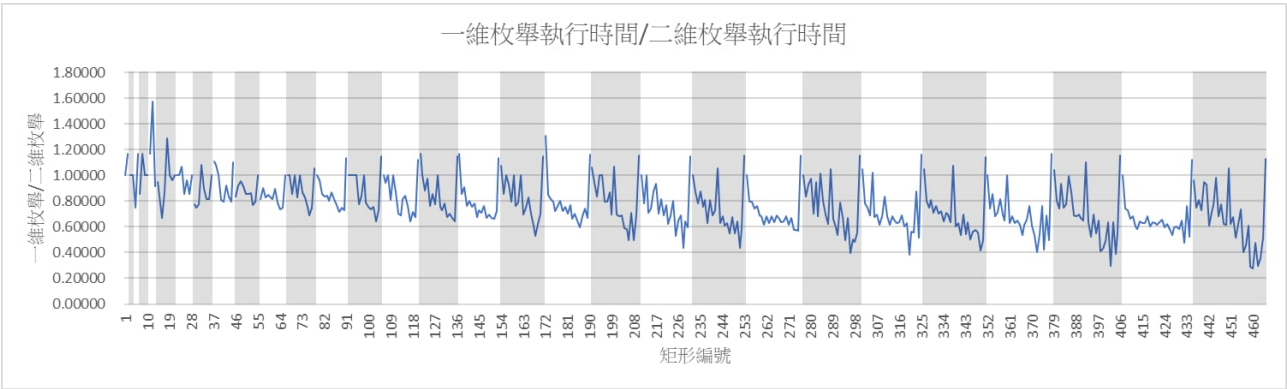


圖 26: 一維枚舉執行時間/二維枚舉執行時間（自行繪製）

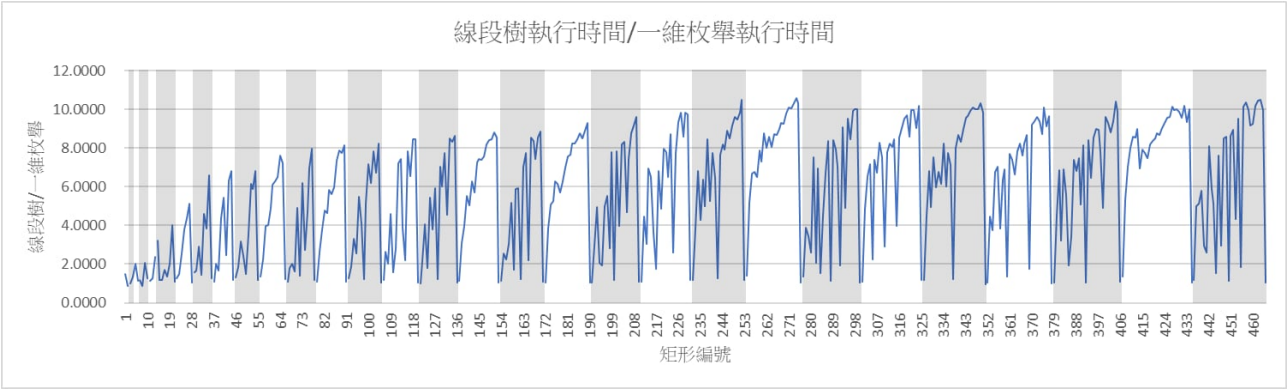


圖 27: 線段樹執行時間/一維枚舉執行時間（自行繪製）

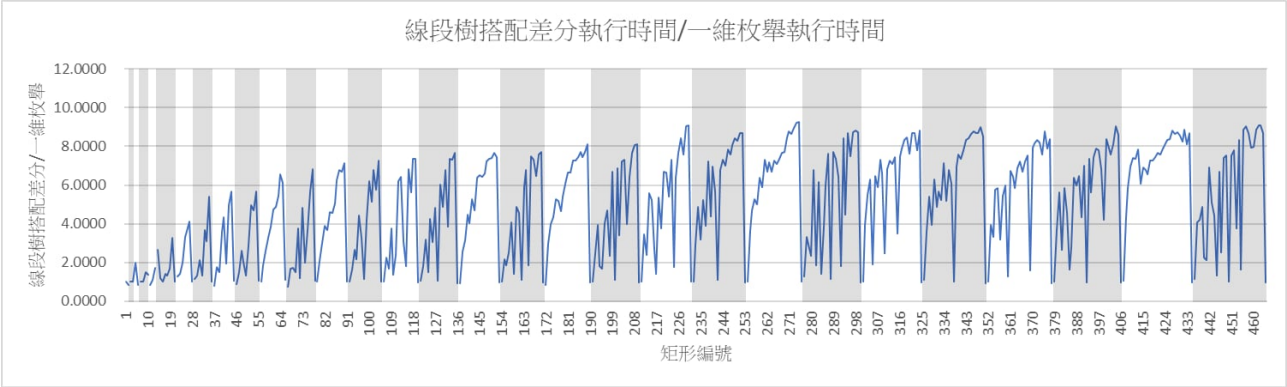


圖 28: 線段樹搭配差分執行時間/一維枚舉執行時間（自行繪製）

三、以長邊為 *x* 軸執行時間與以短邊為 *x* 軸執行時間的比較

「程式的執行效率」實驗中執行速度最快的程式為一維枚舉，因此以一維枚舉程式進行此實驗。圖 29 為一維枚舉程式中以長邊為 *x* 軸執行時間與以短邊為 *x* 軸執行時間的比較。

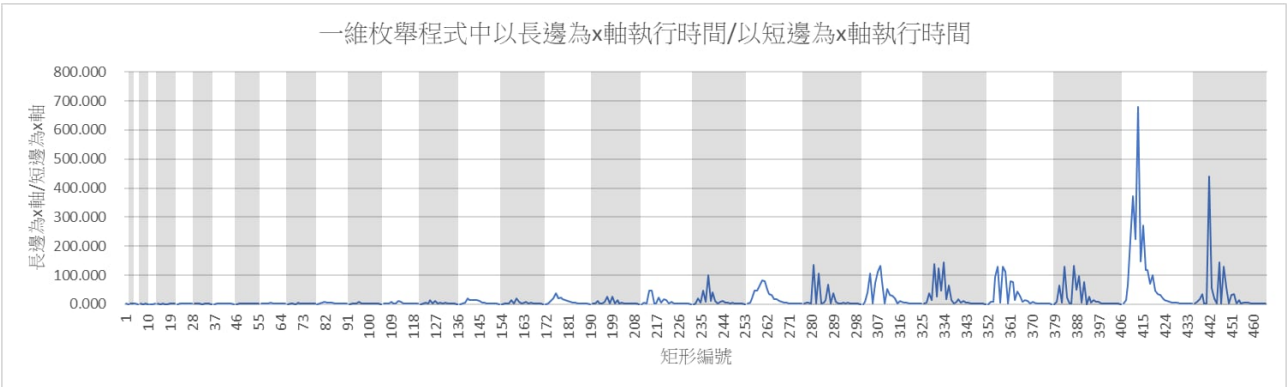


圖 29: 以長邊為 *x* 軸執行時間/以短邊為 *x* 軸執行時間（自行繪製）

伍、討論

一、輾轉相除法與動態規劃解決此問題的正確性

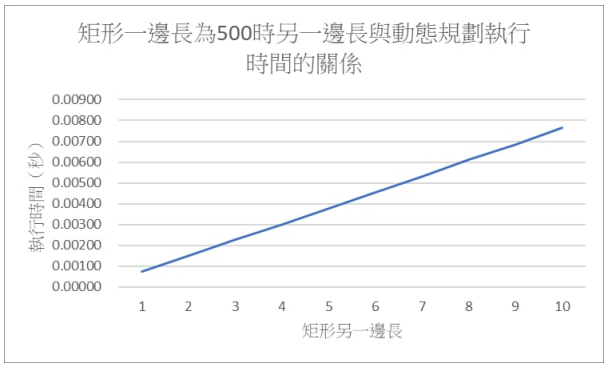
我們發現輾轉相除法處理的矩形如果愈接近正方形（即兩邊長愈相近）而非正方形，第一次切割後會剩下一細長的矩形，導致僅能切出較小的正方形，得到的結果就會較最佳解差上不少，尤其在切割 $n \times (n-1)$ 矩形時，$F_1(n, n-1) = 1 + F_1(n-1, 1) = 1 + (n-1) = n$ ，從圖 19 可以發現從 $n = 6$ 開始，所有 $n \times (n-1)$ 矩形以輾轉相除法分割都較最佳解差，最差的分割結果發生於 28×27 矩形（矩形編號 405），其中輾轉相除法分割出 28 個正方形，而最佳解僅分割出 8 個正方形，較最佳解差 3.5 倍。而邊長 1 ~ 30 的矩形內，動態規劃僅在 16 種矩形中會得到比最佳解差的結果（表 2），其中最差的分割結果發生於 25×23 矩形（矩形編號 323），僅比最佳解差 1.5 倍。因此使用動態規劃是一個兼具正確性與效率的方法。

矩形編號	矩形	動態規劃結果	最佳解	動態規劃結果/最佳解
89	13×11	8	6	1.333333
152	17×16	9	8	1.125000
187	19×16	9	7	1.285714
188	19×17	11	9	1.222222
189	19×18	8	7	1.142857
207	20×17	8	7	1.142857
229	21×19	8	7	1.142857
287	24×11	9	7	1.285714
289	24×13	9	7	1.285714
323	25×23	12	8	1.500000
347	26×22	8	6	1.333333
374	27×23	10	8	1.250000
376	27×25	11	10	1.100000
405	28×27	9	8	1.125000
431	29×25	11	10	1.100000
433	29×27	11	8	1.375000

表 2: 動態規劃結果比最佳解差的矩形

二、程式的執行效率

首先，我們觀察到當動態規劃處理的矩形在一邊長 n 固定時，若另一邊長為 m，在 m 小於 n 的情況下，執行時間大致與 m 呈正比（如圖 21）。已知動態規劃的時間複雜度為 $O(nm(n+m))$，若將 n 視為常數，則程式的複雜度為 $O(nm^2+n^2m)$，$m < n$ 時，m 的一次項係數 n^2 遠大於二次項係數 n，因此雖然 $O(nm^2+n^2m) = O(m^2+m) = O(m^2)$，但此時一次項影響較大，因此圖形呈現一次函數。不過也可以依 $O(m^2)$ 的複雜度推得當 m 遠大於 n 時，圖形將會呈現二次函數。為了驗證此關係，我們將矩形一邊長為 500 時另一邊長 1 ~ 10 與執行時間的關係作圖，以及將矩形一邊長為 10 時另一邊長 1 ~ 500 與執行時間的關係作圖，並且對此二圖的座標軸取對數 log 後繪製回歸直線，以驗證一次函數或二次函數的關係。



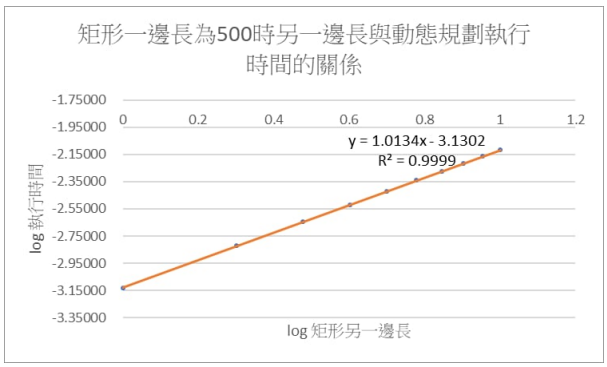
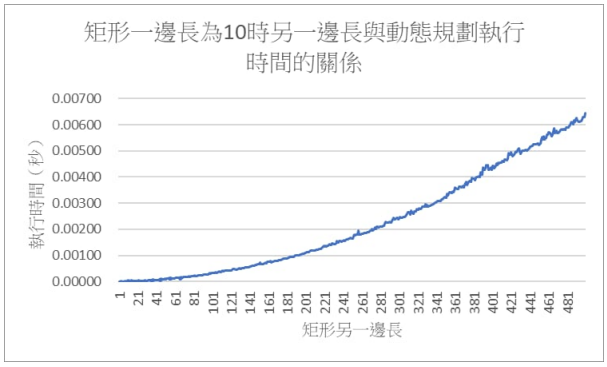


圖 30: 矩形一邊長為 500 時另一邊長與動態規劃執行時間的關係（自行繪製）

圖 31: 矩形一邊長為 500 時 log 另一邊長與 log 動態規劃執行時間的關係（自行繪製）



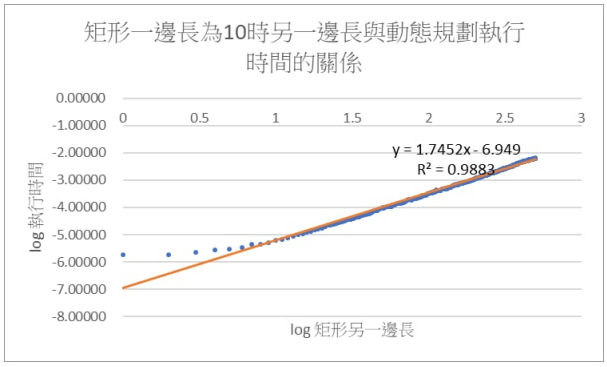


圖 32: 矩形一邊長為 10 時另一邊長與動態規劃執行時間的關係（自行繪製）

圖 33: 矩形一邊長為 10 時 log 另一邊長與 log 動態規劃執行時間的關係（自行繪製）

作圖後可以發現圖 30 中的關係呈線性，而圖 31 中的回歸直線斜率為 1.0134，接近 1，與預期的結果符合。圖 32 中的關係大致呈現二次函數，而圖 33 中的回歸直線斜率為 1.7452，與預期的結果 2 有些微的差距，可能的原因為函數仍然受一次項係數的影響，導致得到的斜率較 2 小，但結果仍然較接近 2，與預期相符。

我們也觀察到二維枚舉、一維枚舉、線段樹、線段樹搭配差分的執行時間高峰位置幾乎相同，我們推測若要使程式的執行時間加長，不只需要較大的邊長，以動態規劃初始化得到的正方形數量最小值也需要較大。我們將一維枚舉程式執行時間超過 0.1 秒的矩形列出（表 3），發現這些矩形的兩邊長都超過 20 ，並且以動態規劃初始化得到的正方形數量都大於 9 ，與我們推測的相符。

矩形編號	矩形	動態規劃結果	最佳解	一維枚舉執行時間（秒）
323	25×23	12	8	0.268418
374	27×23	10	8	0.136467
376	27×25	11	10	0.748612
403	28×25	10	10	0.376930
405	28×27	9	8	0.191599
430	29×24	10	10	0.314148
431	29×25	11	10	0.889001
433	29×27	11	8	0.901205
434	29×28	9	9	0.308536
464	30×29	9	9	0.383294

表 3: 一維枚舉程式執行時間超過 0.1 秒的矩形

從圖 26 中可以觀察到在大部分情況下，一維枚舉執行時間皆較二維枚舉執行時間短，處理 30×25 矩形（編號 460）時此兩執行時間的比例最低，為 0.277768，即執行速度快了 3.6 倍。而由一維枚舉轉變為線段樹的執行效率反而變差了，線段樹執行時間與一維枚舉執行時間的比例（圖 27）最高超過 10，線段樹搭配差分執行時間與一維枚舉執行時間的比例（圖 28）最高也超過 9，我們認為可能的原因為雖然區間更新、區間詢問的複雜度皆可以下降至 $O(\log n)$，但線段樹的常數較直接修改陣列內容大，又本研究使用的陣列長度小，至多僅為 30，因此使用線段樹的優化程度不大，反而受常數的影響而使執行時間變慢。

三、以長邊為 x 軸執行時間與以短邊為 x 軸執行時間的比較

為了方便觀察長邊為 x 軸與以短邊為 x 軸的差異，我們選擇 5×3 矩形並手動將此矩形所有的枚舉過程列出，如圖 34 與 35。

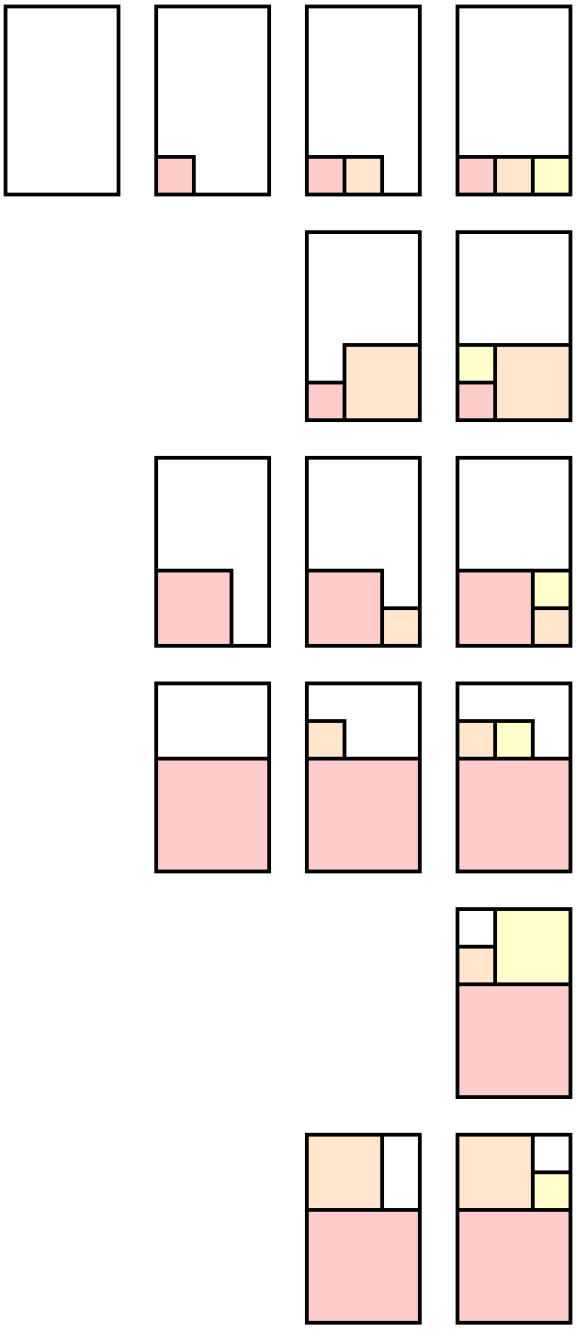


圖 34: 5×3 矩形以短邊為 x 軸的枚舉過程

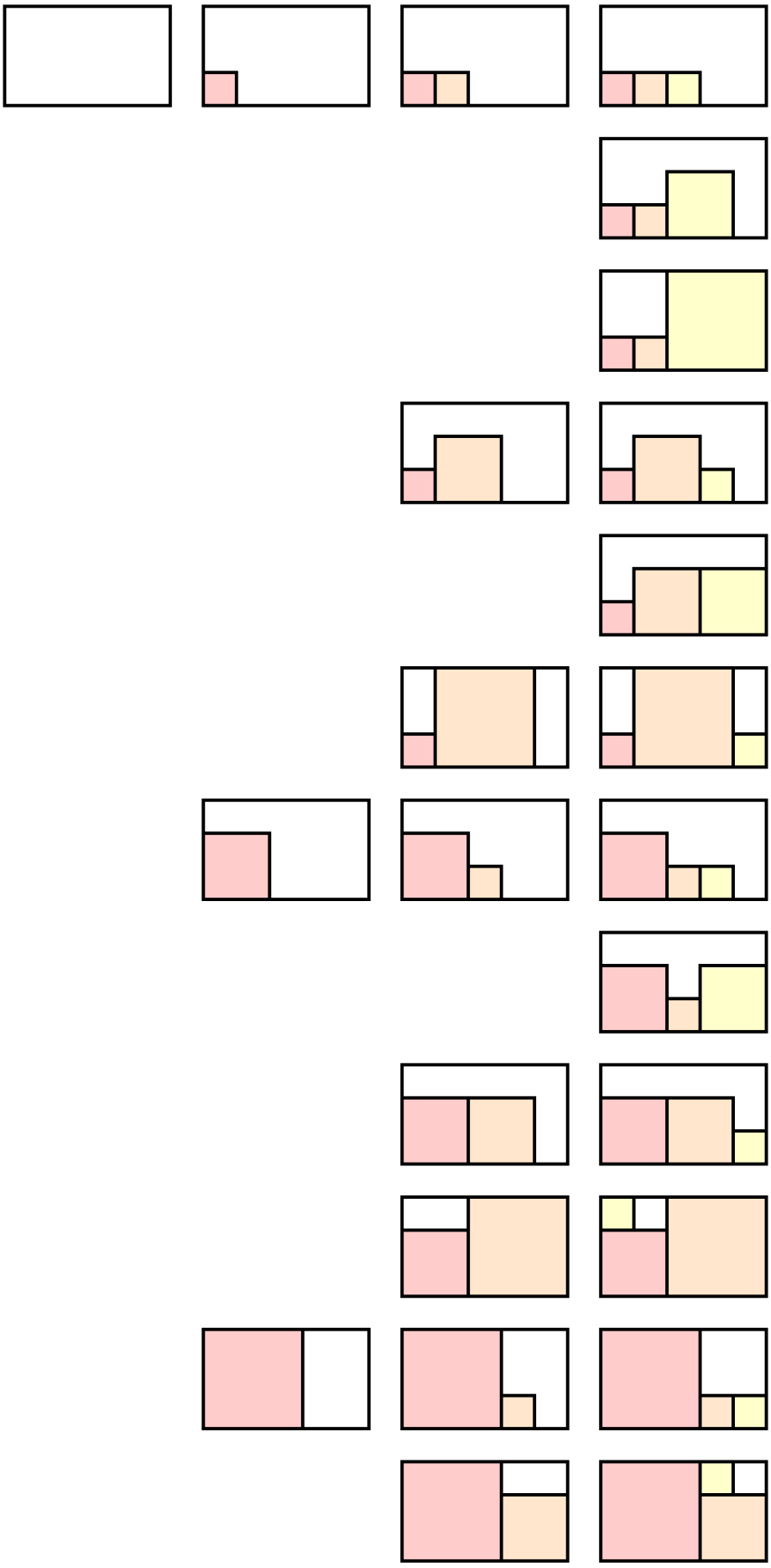


圖 35: 5×3 矩形以長邊為 x 軸的枚舉過程

我們發現若以短邊為 x 軸，矩形的底部較容易被填滿，必須往上堆疊，就比較容易受到「不能覆蓋到其他正方形」的限制（圖 10）。反之，若以長邊為 x 軸，矩形比較容易向右放置，底部不容易被填滿，會較以短邊為 x 軸有更多平坦的空間能放置更大的正方形，因此就會嘗試較多種放置的方法。因此以短邊為 x 軸在一般的情況下會較以長邊為 x 軸執行效率更高。

陸、結論

一、動態規劃在此問題中是一個兼具正確性與效率的方法。

二、一維枚舉是可以得到最佳解程式中效率最高的方法。

三、若未來欲以資料結構優化執行時間，必須克服常數過大的問題。

四、以短邊為 x 軸在一般的情況下會較以長邊為 x 軸執行效率更高。

柒、參考文獻資料

一、矩形中分割方形 (葉龍泉等，1989)。

二、長方體中切割正立方體之研究 (林正倫、吳培綸，2006)。

三、A drawing of a segment tree (a data structure in computer science)(Cafce25，2017)。