PA2 report

1. MPS 實做所使用的資料結構

(1) **讀檔存 chords 資訊**：在程式的一開始我開了一個大小是 points 數量的 vector<int> dictionary(N)去存取在這個 circle 中哪兩個點是連接在一起的。舉例來說，若是 input file 中顯示的連接關係是 0 5，我就會存 dictionary[0] = 5，dictionary[5] = 0，若連接關係是 1 11，我就會存 dictionary[1] = 11，dictionary[11] = 1。如此一來當後面要找點與點的連接資訊時就可以很快的在這個 vector 中查到。

(2) **初始化 table M**：從這部份開始我嘗試過 top-down 與 bottom-up 兩種方法，詳細兩種方法的差別我會在第二部分再說明，這邊先說實做的部分。

Top-down：

這邊我把 table M 開成是下三角的形式，也就是第一橫列 array 大小是 1、第二橫列 array 大小是 2 一直到第 N 橫列 array 大小是 N。這樣一來可以節省記憶體，相較於直接開記憶體空間N^2的 table 省下了約略一半的空間。最後初始化所有值為-1，做為記憶該 subproblem 是否解過而用。

Bottom-up：

一樣把 table M 開成是下三角的形式，不過這次只要初始化 table M 對角線的值為 0 就好，也較是 M[i][i] = 0，i from 0 to N – 1。這個 table 做為運算每個 subproblem 用，每一個 subproblem 都會解過，最後會把 table 填滿。

(3) **計算 number of maximum planar subset**：一樣分成 top-down 與 bottom-up 兩種方法，然後因為存的是下三角的關係，與 HW2 的 MPS 那題不同的是，i 與 j 的關係是有點像做了 matrix transport 一樣，M[i][j]會變成 M[j][i]來表示，不過仍是 i > j。

Top-down：

用 recursive function 的方式去實做，function MPS(i, j)一開始會先判斷這個 subproblem 是否解過，若是曾接解過就直接 return 存在 table 中的那個值了。接下來會分成三個 case(let k = endpoint of i)：

Case 1：if i < k < j, MPS(i, j) = MAX{MPS(i + 1, k) + 1 + MPS(k + 1, j), MPS(i + 1, j)}

Case 2：if k == j, MPS(i, j) = MPS(i + 1, j) + 1

Case 3: otherwise, MPS(i, j) = MPS(i + 1, j)

最後記憶這次解過的 subproblem，return subproblem MPS 數量。

Bottom-up：

用雙層 for 迴圈來實做，依序計算把 table 一個一個填上，也是分成三個 case(j,i 顛倒是因為 table 是下三角的形式)：

Case 1：if i < k < j, M[j,i] = MAX{M[k,i+1] + 1 + M[j,k+1], M[j,i+1]}

Case 2：if k == j, M[j,i] = MPS[j,i+1] + 1

Case 3: otherwise, MPS[j,i] = M[j,i+1]

(4) **找到 maximum planar subset for each resulting chords**：這部分 top-down 與 bottom-up 是一樣的，都是去 check table M 的最後一列，用一個 for 迴圈依序 go through M[N-1][0], M[N-1][1] … M[N-1][N-1]，找 MPS 數量少 1 的位置。因為找(i, N-1)，i from 0 to N – 1，的 MPS 數時，當 MPS 數少 1 表示在那時少掉的那個 chord 是我們所需要的，就把它 fout 在 output file 中，這樣一來用 linear time 就可以找完我們所需的 MPS chords 了。

2. MPS 實做時的發現

(1) **Top-down 與 bottom-up 的差別**：

Top-down

	12.in	1000.in	10000.in	100000.in
Memory(MB)	**12.420**	**14.400**	**208.848**	**19678.788**
Run time(ms)	**0**	**3.316**	**319.419**	**56048.4**

Bottom-up

	12.in	1000.in	10000.in	100000.in
Memory(MB)	**14.596**	**16.576**	**210.160**	**19678.788**
Run time(ms)	**0**	**5.0**	**560.915**	**377868.9**

由表格中的數據可知，top-down 其實比 bottom-up 快非常多倍，尤其越大的 case 越明顯，像 maximum planar subset 這種不需要解出所有 subproblem 的 DP problem 比較適合用 top-down 的方式來解，這驗證教授上課時所說的 top-down 與 bottom-up 的取捨。我一開始是採用 bottom-up 的寫法，但後來想 improve 程式的 runtime 所以改乘用 top-down 的方式來寫，經過優化之後的 run time 果然快非常多！另外，記憶體空間消耗量也是一個不容小覷的問題，因此 memory 的優化我採用了下三角 table M 的形式，M 用 double pointer 去處理，相較於原本直接

開一個$N \times N$的 table 整整少了約一半的記憶體消耗量，算是十分有效的 memory 優化。

(2) **記憶體連續**：這是我最後一次做的優化，當一個二維 array 在做運算時，在同一個橫列中做運算會比在同一個直行中做運算還快，因為在同一個橫列中記憶體是連續的。當時本來是寫的與 HW2 MPS 那題的方法相同，k = endpoint of j，然後是利用 M(i, j) =M(i, j-1)往下掃直到 i == j，這樣一來 top-down 的方式會大量的在 M 的直行中做運算。後來我改成 k = endpoint of i，然後是利用 M(i, j) =M(i+1, j)往上掃直到 i == j，如果是這樣的話就會是在 M 的橫列中做運算了。經過實測發現，確實運算速度快了 2~3 倍，效果顯著！