一開始用以下的資料結構：

Ver. 1

Planar: 包含 `MPS`, `chords`, `vertices` 和 `memo`，

Vertices: `int`，儲存vertices的數量

Chords: `map<int, int>` 儲存所有的chord，因為是以當前的j來尋找chord，所以以 chord 的 `end` 作為 key，`start` 作為 value，在查詢上會比較方便

Memo: `vector< vector< map<int, int> > >` 用 N * N 的 `vector`，每個element中都是 `map<int, int>`，也就是 `memo[i][j]` 中儲存其中的 MPS chords，如上一樣是以 chord 的 end 作為 key，start 作為 value。用 map 也可以用 `map::size` 來比較 element 間的大小。相較之下用 array 在取值和比較大小的部分可能就沒有這麼方便

MPS: `map<int, int> Planar::MPS(int i, int j)` 做法和 HW2 一樣是 Bottom-up，用2個 Loop 從小的 Subproblem 跑到大的 Subproblem，共跑C N取2個 Subproblem

但是memo使用的空間太大，100000的case沒辦法跑完，所以換成如下的資料結構：

Ver. 2

Planar: 包含 `MPS`, `chords`, `vertices`, `maxChords` 和 `memo`，在construct的時候就以 bottom-up 的做法計算每組的最大數量並儲存在 `memo` 中。

Vertices: `int`，儲存vertices的數量

Chords: `int[vertices]` 儲存所有的chord，`chords[start] = end` 和 `chords[end] = start`，兩端都寫進 `chords` 裡。測試之後用 array 的方式儲存比起 map 快約 4 倍。

Memo: `vector< vector<int> >` 用 N * N 的 `vector`，每個element中改成儲存 `int`，即該組的最大chord的數量。

Max Chords: `map<int, int>` 用 end 當 key， start 當 value。

MPS: `map<int, int> Planar::MPS(int i, int j)` 做法改成用 top-down的方式找 chord，用constructor裡計算好的 `memo` 來判斷，並將找到的chord存到 `maxChord` 裡。

將資料結構修改後，測試100000的case成功跑進4分內。