1. **Flow Chart**
   1. Recursive

no

yes

yes

yes

yes

largest yield = yield

return largest yield

yield > largest yield?

Call Recursive  
(where area = area – crop area[i])

no

no

is crop plantable ?

(crop area[i] <= area)

return 0

no

* 1. Memoized

no

yes

update record

yes

no

return largest yield

yes

no

return 0

is crop plantable ?

(crop area[i] <= area)

no

Call Memoize  
(where area = area – crop area[i])

yield > largest yield?

largest yield = yield

yes

record[area – crop area[i]] != 0

yield =   
record[area – crop\_area[i]]

yes

no

* 1. Dynamic Programming

yes

yes

no

no

no

yes

is crop plantable ?

(crop\_area[i] <= area)

subyield =   
crop\_value[i] + yield[area-crop\_area[i]]

subyield > yield[area]?

yield[area] = subyield

yes

no

return yield[total area]

return 0

1. **Design Concept of Algorithm**
   1. Recursive
      1. 依序分別假設(在面積允許的情況下)每種作物必種下一株
      2. 對剩下的面積做遞迴呼叫尋找最佳種植方式
      3. 最後比大小回傳最大值

在設計recursive的演算法要把握最大的原則就是遞迴的呼叫，把問題變成子問題然後在找到解答後層層回傳得到答案，是一個top-down的approach

* 1. Memoized
     1. 依序分別假設(在面積允許的情況下)每種作物必種下一株
     2. 檢查剩下的面積在先前遞迴呼叫是否計算過，有則跳到iv.
     3. 對剩下的面積做遞迴呼叫尋找最佳種植方式，並將回傳值記錄下來
     4. 最後比大小回傳最大值

Memoize的做法和recursive很類似，一樣也是按照top-down approach，唯一的差別在於會把運算過的recursion答案記錄下來，在做呼叫前先檢查之前有沒有做過一樣的運算，有的話就利用記錄下的答案，沒有的話就依樣遞迴呼叫並更新紀錄

* 1. Dynamic Programming
     1. 從面積為一開始，到總面積，分別對每個面積算出最佳解如下
     2. 每算一總面積，就把扣掉當下作物面積的最佳解加上作物價值
     3. 把得到的答案和種植其他作物的最大值做比對，如果大於則取代之
     4. 完成一個面積所有的作物種植比對就將最大值更新到該面積最佳解的紀錄裏面

Dynamic Programming用的是bottom-up的方式去解最佳解，先解面積較小的最佳解，而每次計算面積最佳解的時候就是利用先前已經運算過較小面積的最佳解去求得

1. **Implementation Comparison**

在實際跑的執行時間比較表如下：

|  |  |  |
| --- | --- | --- |
|  | Runtime (case1) | Runtime (case2) |
| Recursive | 0.842 sec | 2.441 |
| Memoize | 0.001 sec | 0.001 sec |
| Dynamic Programming | 0.001 sec | 0.001 sec |

很明顯能看到在時間上Memoize和Dynamic Programming都比recursive快許多，主要原因是因為recursive花費了很多時間在做已經做過的運算，top-down approach無法保證不會再遇到先前算過一模一樣的子問題，所以導致很常都多花了時間在做不必要的運算，而memoize和DP分別用紀錄和bottom-up的方式去避免這樣的問題發生。DP用的bottom-up approach是先從小問題開始算，然後再依照子問題去解母問題，這樣就不會有重複運算的問題，因為不同面積最佳解只會算一次；而memoize用查表的方式去紀錄recursive已經算過的面積最佳解，雖然有可能多一點size的overhead去建立array，但整體效率還是高於recurisve很多，甚至理論上會比DP快一些，原因是因為memoize不見得對每種面積都會去做運算，而是只做「有需要」的面積最佳解，所以在運算時間上又小了DP一些。

1. **Problems in Implement Time and Discussion**

這次因為作業要求case1, case2要在五分鐘內完成，用memoize和dynamic programming都可以輕易過，但是在recursive的部分就遇到了一些問題，程式跑了很久都沒辦法跑完，經過進一步的分析，我發覺問題出在於很多遞迴呼叫其實呼叫的參數都一樣只是順序組合不一樣，例如：a→b→c、c→b→a、a→c→b等呼叫順序其實都是一樣的。為了解決這樣的問題，我在function多加了一個參數，確保當下進行呼叫的crop index不能小於已經使用過的crop index，這樣就可以排除重複的問題，進行這項修改後果然在幾秒內就能跑完。

1. **Bonus Part**

在auxiliary cases中，會發現和original case的作物種類和cost都一樣，只是每一項面積（包括總面積）都乘以了一個相同倍數k，在DP中原先的space complexity就會從W變成Wk（因為要計算的面積種類數多了k倍，存各面積最佳解的array也會放大k倍），time complexity會從O(nW)變成O(nkW)。

要讓time complexity和原先的original case相同的辦法就是在進行DP前找出所有作物面積以及總面積的**最大公因數**，並把他們做resize的動作，這樣在進行DP的時候一樣可以在space complexity W和time complexity O(nW+n) = O(nW)（+n因為需要掃過每種作物一次做resize）下完成，雖然兩者big O看似一樣，但是其實execution time和overhead上還是有差異，後者不用浪費那麼多空間和時間。