# Pseudo code

```
Memorization:
Mem(i, j)
    If c[i][j] != -1
        then return c[i][j]
    if i = 0 or j = 0
        then c[i][j] = 0
            return 0
        else
            then c[i][] = max(Mem(i-1,j), Mem(i,j-1))
                return c[i][j]
PRINT(i, j)
    If i = 0 or j = 0
        then return
    if x[i-1] =y[j-1]
        then PRINT(i-1, j-1);
        output << x[i-1];
        else if Mem(i-1, j) >= Mem(i, j-1)
            then PRINT(i-1, j);
            else PRINT(i, j-1);
main
    string x.length() = n
    string y.length() = m
    PRINT(n, m)
```

# Analysis

1. R E C 超過 3 分鐘時的數列長度

   我每個數列長度都試了 30 次，並扣掉最極端大和極端小的 5 個值在做平均，如此計算下來的結果，大約會在長度在 44, 45 的時候超過 3 分鐘，雖然這樣的計算方式沒辦法精確地計算出平均時間，因為各種 case 的機率比應該是一樣的，而 best case 和 worst case 就應該乘上他們的比例加進平均時間，因此 worst case 應該要佔很大的比例，但由於 recursive 時間浮動的實在太劇烈，我算出平均下來只有不到 1 秒時間的 30 位數列也有可能超過 10 幾分鐘，由於花費的時間實在太久，因此我只能用這種較不精確的算法

   但由這種算法還是能看出他的趨勢，雖說我在 20 分鐘的時候讓 code 強制

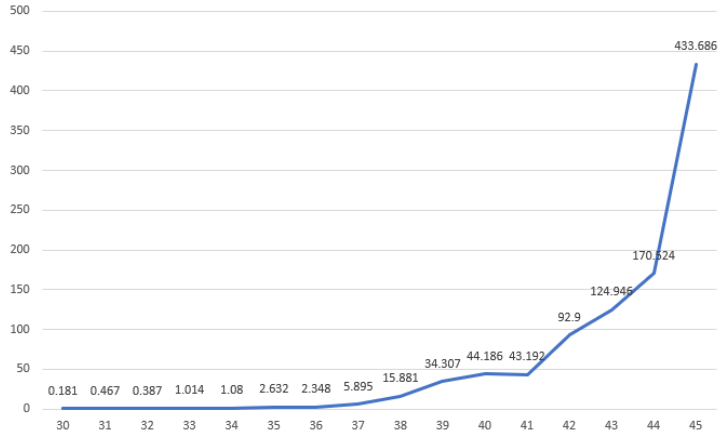
停止很有可能會影響花費時間較長的幾個 case，但我們還是能明顯的看出整個 recursive 的時間成指數成長

而 worse case 不像平均的 case 會有不同 case 的情形，所以指數的趨勢更精確且又更明顯

但 best case 就只是不管數列多長多短，看斜對角有幾個格子就做幾次加法，因此在執行時間上幾乎沒有差多少

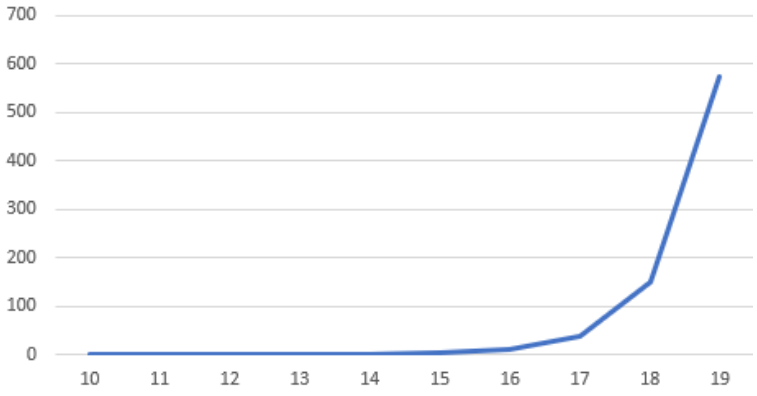
Random case avg.:

| length | time(s) |
|---|---|
| 30 | 0.181 |
| 31 | 0.467 |
| 32 | 0.387 |
| 33 | 1.014 |
| 34 | 1.08 |
| 35 | 2.632 |
| 36 | 2.348 |
| 37 | 5.895 |
| 38 | 15.881 |
| 39 | 34.307 |
| 40 | 44.186 |
| 41 | 43.192 |
| 42 | 92.9 |
| 43 | 124.946 |
| 44 | 170.524 |
| 45 | 433.686 |



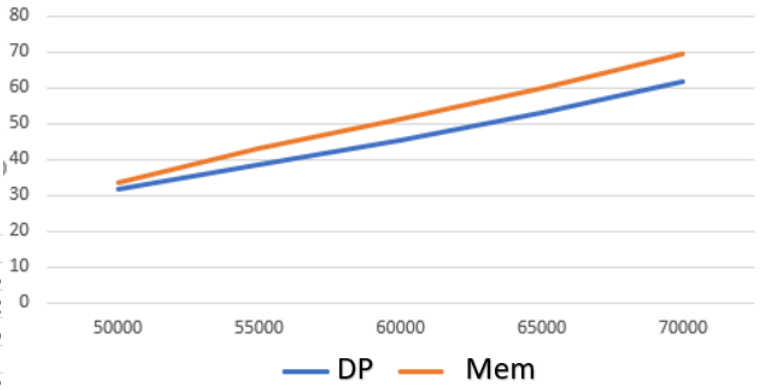

Worst case:

| length | time(s) |
|---|---|
| 10 | 0.011 |
| 11 | 0.025 |
| 12 | 0.076 |
| 13 | 0.12 |
| 14 | 0.76 |
| 15 | 2.76 |
| 16 | 9.85 |
| 17 | 38.08 |
| 18 | 150.51 |
| 19 | 575.46 |




2. D P 慢於 M E M 時的 數列長度
由於我電腦以數列 40000 長度輸入的時候就會以 bad allocate 的 error 跳出 code，所以我 40000 以上的數列長度都直接在工作站上面試，而在工作站上我試到超過 75000 工作站 allocation 也壞了，而到壞掉之前，我的 memorization 還是比 dynamic programming 還要慢

| | mem(s) | dp(s) |
|---|---|---|
| 50000 | 31.718 | 33.53 |
| 55000 | 38.336 | 43.02 |
| 60000 | 45.52 | 51.46 |
| 65000 | 53.3 | 59.91 |
| 70000 | 61.9 | 69.7 |

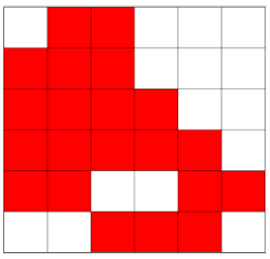


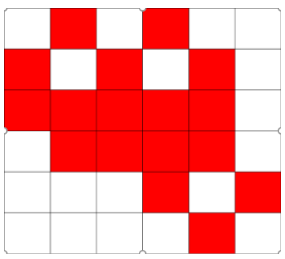

3. 承 上 M E M 相較於 DP/ R E C 省去多少計算

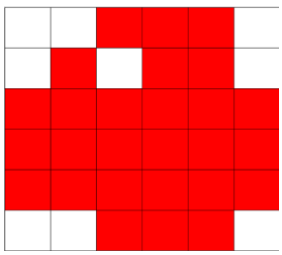
由於要同時把 REC 放進比較裡，所以不能選數列長度太長的例子，所以我就以範例提供的長度(6 位)分析，而計算的次數我以進入那個 function(ex, recursive, mem) count++的方式計算，而 DP 不管 case 如何計算的次數都一樣，也就是把表格全填滿，6*6 = 36。

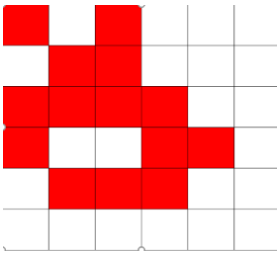
Memorization 和 recursive 就比較麻煩了，特別是 recursive，在 worst case 的時候需要計算 60282 次，而在 best case 的時候一次都不用計算，只要沿著斜角一直+1 就好

而 memorization 雖然跟 recursive 在 best case 的時候一樣只需要一直沿著斜角+1，但在 worst case 的時候只需要計算 117 次，連 recursive 的 1%都不到，但明顯還是比 DP 慢，下面我列了幾個 RANDOM CASE，紅色的格子表示的是 memorization 有進入並填上值的格子，平均下來大概每次有 16 個格子不需要算，雖說比 DP 少算了接近一半的格子，但呼叫 function 的時間與計算的時間相比仍舊長了一點，所以理論上來說，當整張表的格子數很多到算完每一格要很久的時間，以至於呼叫 function 的時間可以被忽略時，memorization 就應該超過 DP，但系統無法負荷我的 code 到可以測出那個超過 DP 的交叉點，甚至在數列長度變長得時候差距還拉大了









4. 其他

由於這次是對時間的測試，所以我和同學們就互相比對了 run time, 結果發現我 recursive 跑的時間比其他人的整整慢了好幾個位數(ex, 我 worst case 大概數列有 13 位就會超過 3 分鐘, 但他們都跑到 19 位才超過)，對於答案都正確卻有這麼大的時間差我們都覺得很奇怪，彼此互相研究了各自的 code 發現我們幾乎概念都一樣，又更覺得奇怪

最後我發現我在數列內容不同時(“1””0” or “0””1”), 我在比較的時候會呼叫一次 rec 來看誰的值比較大，回傳的時候在 call 一次 rec 回傳值，所以每次都會比他們多做一次 recursive, 難怪會有這麼大的時間差異

由於是 worst case 得測試，所以以那張表格來看，每格都會去找它右邊跟上面的值，再加上我呼叫了兩次，所以每層(算出單一一格)都會重複呼叫，所以 time complexity 就會比原本多乘以一個 O(2^n)

由於我只有修改 recursive 的程式，所以 memorization 的 code 還是在比較的時候呼叫了一次 function，在回傳的時候又叫了一次 function，這有可能就是我第二題 memorization 的結果一直無法超越 DP 的原因了