程式設計（105-2）
作業六

作業設計：孔令傑
國立臺灣大學資訊管理學系

第一題

（40 分；每小題 10 分）請回憶一下作業五第二題，昱賢提出的以遞迴方式求解「C 幾取幾」的演算法。那個方法有個好處，就是比較能避免溢位（overflow），但缺點是需要比較長的執行時間。傑夫仔細地思考了一下這個遞迴解法，並且觀察了敬傑寫的範例程式碼（在作業五的解答裡），發現之所以需要比較長的執行時間，是因為「有些問題被重複地求解了」。比如說要算 $C(10,5)$ 時，我們需要算 $C(9,5)$ 和 $C(9,4)$，而要算 $C(9,5)$ 需要算 $C(8,5)$ 和 $C(8,4)$，要算 $C(9,4)$ 則需要算 $C(8,4)$ 和 $C(8,3)$。我們可以發現 $C(8,4)$ 被重複求解了，而在敬傑的程式碼裡面，是貨真價實地要花兩份時間來做這一份工作。整個求解 $C(10,5)$ 的過程當然就有很多時間被浪費在求解這些一樣的題目上了。

有鑑於此，傑夫想到了兩個加速（不要浪費時間）的作法：

(a) 傑夫打算在使用者輸入 n 和 m，並且要求程式計算 $C(n,m)$ 之後，宣告一個 $n \times m$ 的二維矩陣 A，將每個元素都初始化為 -1。傑夫預計要在 A_{ij} 儲存 $C(i,j)$ 的值，如此一來，每當想要求解 $C(i,j)$ 之前，就先檢查一下 A_{ij} 是否為 -1，如果是 -1 就去求解，並用求解後的結果取代 -1，否則就直接回傳 A_{ij} 即可。

由於 n 和 m 是使用者輸入的值，因此傑夫打算宣告一個二維的動態陣列 `com`，並且修改遞迴函數 `combiRec` 為

```
int combiRec(int n, int m, int** com)
{
  if(n < m)
    return -1;
  else if(n == m)
    return 1;
  else if(m == 1)
    return n;
  else
  {
    if(com[n - 1][m - 1] == -1)
    {
      int res = combiRec(n - 1, m, com) + combiRec(n - 1, m - 1, com);
      com[n - 1][m - 1] = res;
      return res;
    }
    else
      return com[n - 1][m - 1];
  }
```

```
}
```

請用你自己的話描述上面這段程式碼，並說明它如何實現傑夫的設計。

Sol. 我們比較傳入的 n,m 的關係，根據傳入值的關係，來計算值，而當 n,m 不是前三個 special case 時，會去檢查前面的值有沒有算過，因為如果有算過的話，就會記載com 這個陣列之中，所以如果值為-1(初始值)，代表沒有算過，必須計算，然後再將計算的結果存入陣列裡。

(b) 接著 main function 也需要做相對應的改寫，以便宣告並且初始化 com。傑夫寫好了 main function 的架構如下：

```
int main()
{
  int n = 0, m = 0;
  cin >> n >> m;

  // declare an n by m dynamic array
  // and initialize all elements in it to -1

  cout << combiRec(n, m, com);
  return 0;
}
```

請幫傑夫完成註解那部份的宣告和初始化。請注意 int** 和二維靜態陣列是不一樣的，把 com 宣告成一個靜態二維陣列丟進 combiRec 是會產生 syntax error 的！完成之後，請試著輸入 $n = 35$ 和 $m = 15$，雖然結果是錯的（因為溢位），但是確實會比之前快多了。

Sol.

```
int** com = new int*[n]
for (int i = 0; i < n; i++){
  com[i] = new int[m];
  for (int j = 0; j < m; j++){
    com[i][j] = -1;
  }
}
```

(c) 我們在上面使用的方法，說到底就是「解過的題目不要再解一次」，但本質上還是一個 top-down 的設計：「我要解一個大問題，就拆成一些小問題去分別解決」。在同一種思維模式下，其實有另一種 bottom-up 的設計常常是更好的：「我從最小的問題開始往上解，以便每當我需要解一個比較大的問題時，它底下的小問題都已經被解好了，我只要查一下表即可」。這類型演算法被通稱為**動態程式規劃**（dynamic programming）。

我們用 Fibonacci 數列來當例子。還記得在之前上課時，我們曾寫過程式根據給定的 n，找第 n 個 Fibonacci 數，而我們利用那個例子說明了遞迴的作法可能會浪費很多時間。當時的實作基本上是這樣：

```
int fib(int n)
{
  if(n <= 2)
    return 1;
  else
    return fib(n - 1) + fib(n - 2);
}
```

如果我們改採用上面介紹的 top-down 作法，可以把程式改寫成：

```
int fibRec(int n, int* fibValue)
{
  if(n <= 2)
    return 1;
  else
  {
    if(fibValue[n - 1] == -1)
    {
      int res = fibRec(n - 1, fibValue) + fibRec(n - 2, fibValue);
      fibValue[n - 1] = res;
      return res;
    }
    else
      return fibValue[n - 1];
  }
}
```

其中 `fibValue` 是儲存前 n 個 Fibonacci 數的一維動態陣列。如果我們改採用 bottom-up 的動態程式規劃，可以把程式改寫成：

```
int fibDP(int n, int* fibValue)
{
  fibValue[0] = fibValue[1] = 1;
  for(int i = 2; i < n; i++)
    fibValue[i] = fibValue[i - 1] + fibValue[i - 2];
  return fibValue[n - 1];
}
```

可以看到 `fibDP` 的 header 和 `fibRec` 的 header 是一模一樣的，亦即 `fibValue` 依然是用來儲存前 n 個 Fibonacci 數的一維動態陣列。然而現在我們是從最小的子問題（$k = 1$ 和 2）開始往比較大的子問題（$k = 3, 4, \ldots$）去解，直到我們解出原本要解的問題（$k = n$）為止。

現在，請改寫傑夫在 (a) 小題寫的 `combiRec` 函數，將之以動態程式規劃的方式實作出來。

Sol.

```
int combiRec(int n, int m, int** com)
{
  if (n < m)
    return 1;
  for (int i = 0; i < n; i++){
    com[i][0] = 1;
  }
  for (int i = 0; i < m; i++){
    com[i][i] = i+1;
  }
  for (int i = 1; i < n; i++){
    for (int j = 1; j < n; j++){
      com[i][j] = com[i - 1][j] + com[i - 1][j - 1];
    }
  }
  return com[n - 1][m - 1];
}
```

(d) 請比較 (a) 和 (c) 小題的兩種實作方式，並且用自己的話說明兩者各有什麼「在時間複雜度上的」好處跟壞處。

Sol. 比較 (a)(c) 兩題的作法時，我們可以在執行完後將com 陣列印出來，這時發現到當如果使用 (c) 的做法時，我們會完全填滿com 陣列，時間複雜度為 O(n^2) 但在 (a) 小題時，只需要計算需要的格子，計算複雜度為 O(min(n,m))

第二題

（20 分；每小題 10 分）請回答以下兩個彼此之間沒有任何關聯的問題：

(a) 請針對本次作業的第四題所指定的演算法，撰寫一個 pseudocode。請盡量讓你的 pseudocode 詳細且精確，接近可以被直接翻譯成程式碼（但還不是程式碼）的樣子，像作業三第六頁的第二個 pseudocode 那樣。你的 pseudocode 愈詳細且精確，就會得到愈高分。

注意：第四題是加分題，但這題是基本題。

Sol.

```
宣告一個空的(零售商,物流中心)清單
length = 0
for i 從 1 到 m:
  if 零售店 i 還有需求
    nearCenter = -1
    minDistance = INF
    for j 從 1 到 n
      distance = 零售店 i 及物流中心 j 的距離
```

```
      if distance 使得毛利為正
        if distance <  minDistance
          minDistance = distance
          nearCenter = j
        else if distance == minDistance:
          if 物流中心 j 的存貨 > nearCenter的存貨
            nearCenter = j
    將 i j 存入清單中
    length++

if length == 0
  代表沒有零售店可以被滿足需求了
  跳出迴圈
for i 從 1 到 length(清單的長度)
  minDistance = INF
  optimalStore = 0;
  if 清單的第 i 組 pair 的距離 < minDistance
    minDistance = 清單的第 i 組 pair 的距離
    optimalStore = 清單第 i 組的零售店編號
  else if 清單的第 i 組 pair 的距離 == minDistance
    if 清單第 i 組的零售店需求量 > optimalStore的需求量
      optimalStore = 清單第 i 組的零售店編號

供給量 = min(optimalStore的需求量, 相對的物流中心存貨)
optimalStore 的需求量 -= 供給量
相對的物流中心存貨 -= 供給量
```

(b) 請考慮第六講投影片第 61 頁的程式碼：

```
int main()
{
  int r = 3;
  int** array = new int*[r];
  for(int i = 0; i < r; i++)
  {
    array[i] = new int[i + 1];
    for(int j = 0; j <= i; j++)
      array[i][j] = j + 1;
  }
  print(array, r); // later
  // some delete statements
  return 0;
}
```

佩蓉在 `// some delete statements` 那邊嘗試寫了一句 `delete` 敘述

```
  delete [] array;
```

但是昱賢跟她說這樣還是會發生 memory leak，因為那三條一維陣列的空間都沒有被釋放。他說，得要搭配一個迴圈去釋放那三條一維陣列，才是這裡需要的完整釋放動態記憶體空間的作法。請幫佩蓉寫出完整且正確的程式碼，來釋放所有被動態配置的記憶體空間。你的答案可能包含佩蓉原本寫的，也可能不包含；若你覺得應該包含，請把佩蓉原本寫的那句包含在你的答案裡。

Sol.

```
for (int i = 0; i < r; r++){
  delete [] array[i];
}
delete [] array;
```