1. (10%) Suppose you are asked to implement the n-th Fibanacci number using recursion only (i.e., no loop implementation)? How to avoid stack overflow for large n?

使用遞迴實現第 n 個費波那契數，可定義一個遞迴函數，該函數返回第 n 個費波那契數。遞迴過程中，每次調用函數時，我們會對第 n-1 和 n-2 個費波那契數再次調用同一函數。但是，對於大的數 n，使用純遞迴的方式容易導致溢出問題，因為每一次函數調用都會佔用堆疊空間。

為了避免這個問題，可以採用**記憶化遞迴**（Memoization）的方法。具體來說，可以使用一個數組或哈希表來保存之前計算過的費波那契數值，當需要計算某個費波那契數時，首先查找是否已經計算過，如果已經計算過，直接返回結果，否則進行遞迴計算。這樣可以避免重複計算，同時減少遞迴深度，從而降低堆疊空間的使用量。

下面是使用記憶化遞迴的 C 實作：

```
#include <stdio.h>

#define MAX 100
int memory[MAX];

int fibonacci(int n) {
   if (n <= 1) {
      return n;
   }
   if (memo[n] != -1) {
      return memory[n];
   }
   return memory[n] = fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
   int n = 50; // 要計算的費波那契數的索引
   for (int i = 0; i < MAX; i++) {
      memory[i] = -1;
   }
   printf ("第 %d 個費波那契數為 %d\n", n, fibonacci(n));
   return 0;
```

}

使用記憶化遞迴，我們僅需要對每個 n 只計算一次，因此在計算大數 n 的情況下可以避免堆疊溢出，同時提高計算效率。

2. What is the Big O complexity of following recurrence? Justify your answer.
(a). T(n) = 2T(n / 4) + T((2 / 3)n) + cn, where c is a constant.
(b). T(n) = 2T($\sqrt{n}$)+lgn. (hint: you may replace n with other form)

(a)
這個遞迴式的 Big O 複雜度是 O(nlogn)。
可以使用主定理 (Master Theorem) 來證明這一點。
遞迴式的形式為 T(n) = 2T(n/4) + T((2/3)n) + cn。
根據主定理，如果存在常數 k > 0 和 d >= 0，使得：

對於所有的 n 大於某一個 n0，有 T(n) <= kn^d，那麼 T(n) 的複雜度是 O(n^d)。
如果 T(n) <= kn^d 對於某一個常數 d > 0 以及一些正數 ε > 0 成立，同時對於充分大的 n，有 T(n) >= kn^d，那麼 T(n) 的複雜度是 O(n^d * log n)。

對於遞迴 T(n) = 2T(n/4) + T((2/3)n) + cn，可以看到第二項 T((2/3)n) 的規模比較大，所以可以忽略掉其他項。因此，將遞迴式簡化為 T(n) = T((2/3)n) + cn。這個遞迴式符合主定理的第二種情況，其中 d = 1，因此複雜度為 O(n^d * log n) = O(n * log n)。

(b)
將 n 替換為 2^k，其中 k = log2(n)，這樣可以將遞迴式表示為 T(2^k) = 2T($2^{(k/2)}$) + k。這就變成了形式類似合併排序的遞迴式。因此，我們可以使用主定理來解這個遞迴式。

根據主定理，如果遞迴式具有形式 T(n) = aT(n/b) + f(n)，其中 a >= 1，b > 1，且 f(n) 是漸進正的，那麼：

如果 f(n) = O($n^{logb(a-ε)}$)，對於某個 ε > 0，那麼 T(n) = Θ($n^{logb(a)}$)。
如果 f(n) = Θ($n^{logb(a)}$)，那麼 T(n) = Θ($n^{logb(a)}$ * log n)。
如果 f(n) = Ω($n^{logb(a+ε)}$)，對於某個 ε > 0，且 af(n/b) <= cf(n) 對於某個 c < 1 和所有足夠大的 n，那麼 T(n) = Θ(f(n))。

對於遞迴式 T(2^k) = 2T($2^{(k/2)}$) + k, a = 2, b = 2 , f(n) = k。因為 k = log2(n)，所以 f(n)

= log2(n)。我們可以看到 f(n) = Θ(log n)，同時也符合主定理的第三種情況。因此，複雜度為 Θ(log n)。

3. Prove that an n-element heap has at most $\boldsymbol{n}/(\boldsymbol{2}^{(\boldsymbol{h}+\boldsymbol{1})})$ nodes at height h.
使用數學歸納法來證明這一點。

基礎情況：當 h = 0 時，這意味著在堆的最底層，即葉子節點所在的層級。在這種情況下，所有的葉子節點都是在同一個層級，每個葉子節點都是一個高度為 0 的節點。一個 n 元素的堆有 n 個葉子節點，因此在高度 h = 0 的情況下，有 $n/(2^{(0+1)}) = n/2$ 個節點。這是基礎情況。

假設對於某個正整數 k，當 h = k 時，一個 n 元素的堆最多有 $n/(2^{(k+1)})$ 個節點。

接下來，假設高度為 k+1 的節點數量是 $n/(2^{(k+2)})$。可以通過觀察堆的結構來進行證明。在高度為 k+1 的層級上，每個節點都有兩個子節點，這意味著在高度為 k 的層級上，每個節點都有 2 個父節點。
因此，在高度為 k 的層級上，每個節點都負責其下層的兩倍節點的數量。由於假設高度為 k 的層級上有 $n/(2^{(k+1)})$ 個節點，所以高度為 k+1 的層級上的每個節點都負責其下層的 $n/(2^{(k+1)}) * 2 = n/(2^{(k+2)})$ 個節點的數量。

因此，通過數學歸納法，證明了對於所有的正整數 h，一個 n 元素的堆最多有 $n/(2^{(h+1)})$ 個節點。

4. Mr. Stupid claims we can sort n real numbers in linear time by multiplying a large integer to each real number such that all of them become integers. Then， the counting sort can be used to sort these integers in linear time. What is the problem of this method?
將每個實數都乘以一個大整數以轉換成整數的方法可能會導致數據溢出。如果原始的實數數值非常大，將其乘以一個大整數後，可能會超出表示範圍，進而導致溢出錯誤。

其次，即使假設數據不會溢出，將每個實數轉換成整數也可能會導致精度有問題。因為將實數轉換為整數會截斷小數部分，這樣可能會導致排序結果不準確。

最後，即使忽略溢出和精度丟失的問題，使用計數排序來排序這些大整數也不一定會在線性時間內完成。計數排序通常用於整數排序，當整數的範圍很小且比較集中時才能夠達到線性時間的排序。但是在這種情況下，由於我們將實數轉換為整數後，整數的範圍可能會變得非常大，使得計數排序的效率大大降低。

綜上所述，這種方法存在著數據溢出、精度丟失以及計數排序效率不高的問題，因此並不適用於在線性時間內對實數進行排序。

5. Longest common subsequence.
(a) (10%) Write the optimal substructure (recurrence) of computing LCS of k sequences, where k = 3.
What is the time complexity of computing LCS for k sequences of length n.
(b) Mr. Smart claimed that the LCS of three sequences can be obtained by first computing LCSs of any two sequences (say LCS12), and then compute LCS of LCS12 and the remaining sequence. What's the bug of this method? Give an example.
(c) Given a string, find the longest subsequence occurring at least twice in the string, requiring their indices must not overlap. e.g., Given ATACTCGAG, the answer is 4 since ATCG occurs twice and their indices (i.e., (1,2,4,7) and (3,5,6,9)) do not overlap. Describe a dynamic programming (recurrence) for this problem. Illustrate a bottom-up DP using the string ATACTCGAG.
(d) Compute the Longest Palindrome Subsequence (LPS) in any sequence using dynamic programming. Given a string "character," the LPS is "carac." You should write down the recurrence and bottom-up computation.

(a)
計算 k 個序列的最長公共子序列（LCS）的最佳子結構（recurrence）可描述為以下形式，當 k = 3 時：

假設我們有三個序列 S1、S2 和 S3，它們的長度分別為 n1、n2 和 n3。

我們定義 dp[i][j][k] 為三個序列 S1[0..i-1]、S2[0..j-1] 和 S3[0..k-1] 的 LCS 長度。

則我們可以使用以下遞迴關係來填充 dp 數組：

如果 S1[i-1] = S2[j-1] = S3[k-1]，則 dp[i][j][k] = dp[i-1][j-1][k-1] + 1。
否則，dp[i][j][k] = max(dp[i-1][j][k]，dp[i][j-1][k]，dp[i][j][k-1])。

這個遞迴關係表示，在考慮第 i、j 和 k 個字符時，我們比較它們是否相等。如果相等，則 LCS 長度增加 1，並且我們考慮前一個字符時的 LCS 長度；如果不相等，則我們將 LCS 長度更新為前一個字符的 LCS 長度中的最大值。
對於計算 k 個序列的長度為 n 的最長公共子序列，其時間複雜度為 O(n^3)。

(b)
Mr. Smart 的方法存在一個錯誤，這是因為 LCS12 和剩餘序列之間的 LCS 不一定是整體的 LCS。考慮以下示例：序列 A 為 "ABCD"，序列 B 為 "ACBD"，序列 C 為 "ADBC"。根據 Mr. Smart 的方法，我們首先計算序列 A 和 B 的 LCS，得到 "ABD"。然後計算 "ABD" 和序列 C 的 LCS，但是 "ABD" 和序列 C 的 LCS 是 "AD"，與序列 A、B、C 的 LCS "ABD" 不同。因此，Mr. Smart 的方法不適用於找到 k 個序列的整體 LCS。

(c)
計算一個給定字符串中至少出現兩次的最長子序列，且要求它們的索引不能重疊的問題，可以通過動態規劃遞迴來解決：
下面描述了動態規劃的遞迴關係，以及使用 bottom-up 的方法對字符串 "ATACTCGAG" 進行動態規劃。

遞迴關係：
假設 dp[i][j] 表示字符串中以索引 i 和 j 結尾的最長重複子序列的長度。
我們可以使用以下遞迴關係來填充 dp 數組：

如果字符 s[i] 等於字符 s[j] 且 i 不等於 j，則 dp[i][j] = dp[i-1][j-1] + 1。
否則，dp[i][j] = max(dp[i-1][j], dp[i][j-1])。

bottom-up 的動態規劃：
我們可以使用 bottom-up 的方法來填充 dp 數組。首先初始化 dp 數組為零。然後遍歷字符串中的每對字符，根據上述遞迴關係來更新 dp 數組。

下面是對字符串 "ATACTCGAG" 的 bottom-up 的動態規劃過程：
初始時，dp 數組為：

		1	2	3	4	5	6	7	8	9
		A	T	A	C	T	C	G	A	G
1	A	0	0	0	0	0	0	0	0	0
2	T	0	0	0	0	0	0	0	0	0
3	A	0	0	0	0	0	0	0	0	0
4	C	0	0	0	0	0	0	0	0	0
5	T	0	0	0	0	0	0	0	0	0
6	C	0	0	0	0	0	0	0	0	0
7	G	0	0	0	0	0	0	0	0	0
8	A	0	0	0	0	0	0	0	0	0
0	G	0	0	0	0	0	0	0	0	0

接著，遍歷字符串中的每對字符：

字符 'A' 與字符 'T' 不相等，因此 dp[1][0] = max(dp[0][0], dp[1][1]) = 0。
字符 'T' 與字符 'A' 不相等，因此 dp[2][1] = max(dp[1][1], dp[2][2]) = 0。
字符 'A' 與字符 'C' 不相等，因此 dp[3][2] = max(dp[2][2], dp[3][3]) = 0。
...
依此類推，我們填充整個 dp 數組，最後找到 dp 數組中的最大值，即為最長重複子序列的長度。在這個例子中，最終 dp[1][7] = dp[3][9] = 4 (ATCG 出現第二次)，最長重複子序列的長度為 4。

(d)
要計算任何序列的最長回文子序列（Longest Palindrome Subsequence，LPS），可以使用動態規劃。
以下是動態規劃的遞迴關係和 bottom-up 的計算過程：

遞迴關係：
假設 dp[i][j] 表示字符串中從索引 i 到索引 j 的子串的最長回文子序列的長度。我們可以使用以下遞迴關係來填充 dp 數組：

如果字符 s[i] 等於字符 s[j]，則 dp[i][j] = dp[i+1][j-1] + 2。
如果字符 s[i] 不等於字符 s[j]，則 dp[i][j] = max(dp[i+1][j], dp[i][j-1])。
bottom-up 的動態規劃：
使用 bottom-up 的方法來填充 dp 數組。首先初始化 dp 數組為單個字符的長度為 1，即 dp[i][i] = 1。然後遍歷字符串中的每個子串的長度，根據上述遞迴關係來更新 dp 數組。

下面是對字符串 "character" 的 bottom-up 的動態規劃過程：

初始時，dp 數組為：（一個長度為 1 的字串本身就是迴文字串。）

		1	2	3	4	5	6	7	8	9
		c	h	a	r	a	c	t	e	r
1	c	1								
2	h		1							
3	a			1						
4	r				1					
5	a					1				
6	c						1			
7	t							1		
8	e								1	
0	r									1

接著，遍歷字符串中的每個子串的長度：

對於長度為 2 的子串，我們比較相應的字符：

'c' 不等於 'h'，因此 dp[0][1] = max(dp[0][0], dp[1][1]) = 1。
'h' 不等於 'a'，因此 dp[1][2] = max(dp[1][1], dp[2][2]) = 1。
'a' 不等於 'r'，因此 dp[2][3] = max(dp[2][2], dp[3][3]) = 1。
...
對於長度為 3 的子串，我們比較相應的字符：

'c' 不等於 'a'，因此 dp[0][2] = max(dp[0][1], dp[1][2]) = 1。
'h' 不等於 'r'，因此 dp[1][3] = max(dp[1][2], dp[2][3]) = 1。
...
依此類推，我們填充整個 dp 數組，最後找到 dp[0][8]，即為最長回文子序列的長度。在這個例子中，最長迴文子序列為"carac"，最長回文子序列的長度為 5。

6. Consider the knapsack problem of n items and W pack size. Suppose the pack/item sizes are very large and the item values are very small. Give a dynamic programming for solving this problem. Illustrate your algorithm using the following example (W=500).

Item	Weight	Value
1	100	1
2	200	2
3	250	4
4	300	5

在背包容量為 W 的情況下，解決 n 個物品的背包問題，可以使用動態規劃算法。

令 dp[i][w] 表示在前 i 個物品中選擇使得重量不超過 w 的情況下的最大價值。

則我們可以使用以下遞迴關係來計算 dp[i][w]：

如果物品 i 的重量超過 w，則 dp[i][w] = dp[i-1][w]，即不選擇物品 i。
否則，dp[i][w] = max(dp[i-1][w]， dp[i-1][w-weight[i]] + value[i])，即在選擇和不選擇物品 i 中取最大值。
接下來我們可以使用這個遞迴關係填充一個二維數組 dp。最後，答案即為 dp[n][W]。

使用例子中的數據 (W=500，物品數 n=4)填充 dp：

dp = [

[0， 0， 0， ...， 0]，
[0， 0， 0， ...， 0]，
[0， 0， 0， ...， 0]，
[0， 0， 0， ...， 0]，
[0， 0， 0， ...， 0]
]
現在填充 dp[4][500]：

對於物品 1，重量為 100，價值為 1。因為 100 <= 500，所以 dp[1][100] = 1，其它 dp[1][w] = 0。
對於物品 2，重量為 200，價值為 2。因為 200 <= 500，所以 dp[2][200] = 2，其它 dp[2][w] = 0。
對於物品 3，重量為 250，價值為 4。因為 250 <= 500，所以 dp[3][250] = 4，其它 dp[3][w] = 0。
對於物品 4，重量為 300，價值為 5。因為 300 <= 500，所以 dp[4][300] = 5，其它 dp[4][w] = 0。
現在我們使用上面的遞迴關係來計算 dp[4][500]。例如，對於 dp[4][500]，我們可以通過比較 dp[3][500] 和 dp[3][500-300] + 5 來得到最大值。

這樣填充 dp 數組直到得到 dp[4][500]，我們就能得到最大價值。
以下是範例：

```
#include <stdio.h>

#define max(a， b) ((a) > (b) ? (a) : (b))

// 定義最大物品數量和最大背包容量
#define MAX_ITEMS 100
#define MAX_WEIGHT 1000

// 物品的重量和價值
int weight[MAX_ITEMS] = {100， 200， 250， 300};
int value[MAX_ITEMS] = {1， 2， 4， 5};

// 動態規劃函數
int knapsack(int n， int W) {
    int dp[MAX_ITEMS + 1][MAX_WEIGHT + 1];
```

```
    // 初始化 dp 數組
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0) {
                dp[i][w] = 0;
            } else if (weight[i - 1] <= w) {
                dp[i][w] = max(value[i - 1] + dp[i - 1][w - weight[i - 1]]， dp[i - 1][w]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    return dp[n][W];
}

int main() {
    int n = 4; // 物品數量
    int W = 500; // 背包容量

    int max_value = knapsack(n， W);
    printf("最大價值為：%d\n"， max_value);

    return 0;
}
```

首先定義了物品的重量和價值。然後使用一個二維數組 dp 來保存動態規劃的結果。填充 dp 數組，最後返回 dp[n][W]，即最大價值。

7. Consider the following six activities with (start time, finish time, and value): (2, 4, 3), (5, 5, 5), (3, 4, 2), (1, 4, 3), (1, 3, 1), (3, 5, 4). Illustrate a dynamic programming algorithm for computing the mutually exclusive subset of activities of maximum total values using the above example.

使用動態規劃算法來計算具有最大總價值的互斥活動子集。
以下是對於給定活動的動態規劃算法：

首先，將活動按照結束時間進行排序，確保每次選擇的活動是不衝突的。

然後，定義一個 dp 數組，其中 dp[i] 表示在考慮前 i 個活動時的最大總價值。

使用遞迴關係來填充這個數組。

遞迴關係如下：

如果第 i 個活動是第一個活動（即沒有其他活動與之相互排斥），則 dp[i] 等於第 i 個活動的價值。
否則，對於每個結束時間早於第 i 個活動的活動 j，我們可以計算 dp[i] = max(dp[i], dp[j] + value[i])。
最後，dp 數組中的最大值就是所求的相互排斥的活動子集的最大總價值。

下面是範例：

```
#include <stdio.h>

// 定義活動結構
struct Activity {
    int start;
    int finish;
    int value;
};

// 按結束時間遞增排序
void sortActivities(struct Activity activities[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (activities[j].finish > activities[j+1].finish) {
                // 交換活動
                struct Activity temp = activities[j];
                activities[j] = activities[j+1];
                activities[j+1] = temp;
            }
        }
    }
}

// 動態規劃算法計算最大總價值
int dynamicProgramming(struct Activity activities[], int n) {
    // 將活動按結束時間遞增排序
    sortActivities(activities, n);
```

```
    // 初始化 dp 數組
    int dp[n];
    for (int i = 0; i < n; i++) {
        dp[i] = activities[i].value;
    }

    // 填充 dp 數組
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (activities[j].finish <= activities[i].start) {
                dp[i] = (dp[i] > dp[j] + activities[i].value) ? dp[i] : dp[j] + activities[i].value;
            }
        }
    }

    // 找到最大總價值
    int maxTotalValue = 0;
    for (int i = 0; i < n; i++) {
        if (dp[i] > maxTotalValue) {
            maxTotalValue = dp[i];
        }
    }

    return maxTotalValue;
}

int main() {
    // 活動列表
    struct Activity activities[] = {
        {2, 4, 3},
        {5, 5, 5},
        {3, 4, 2},
        {1, 4, 3},
        {1, 3, 1},
        {3, 5, 4}
    };
```

```
    // 活動數量
    int n = sizeof(activities) / sizeof(activities[0]);

    // 計算最大總價值
    int maxTotalValue = dynamicProgramming(activities, n);
    printf("最大總價值: %d\n", maxTotalValue);

    return 0;
}
```

這個動態規劃算法的時間複雜度取決於排序活動的時間複雜度和填充 dp 數組的時間複雜度。

排序活動：對活動按結束時間進行排序的時間複雜度 O(n log n)（快速排序），其中 n 是活動的數量。

填充 dp 數組：填充 dp 數組的時間複雜度為 $O(n^2)$，因為我們需要對每個活動 j 找到結束時間早於當前活動的所有活動 j 進行比較。

因此，整個算法的時間複雜度為 $O(n^2 + n \log n)$。在最壞的情況下，它可以簡化為 $O(n^2)$，因為排序的時間複雜度優於填充 dp 數組的時間複雜度。