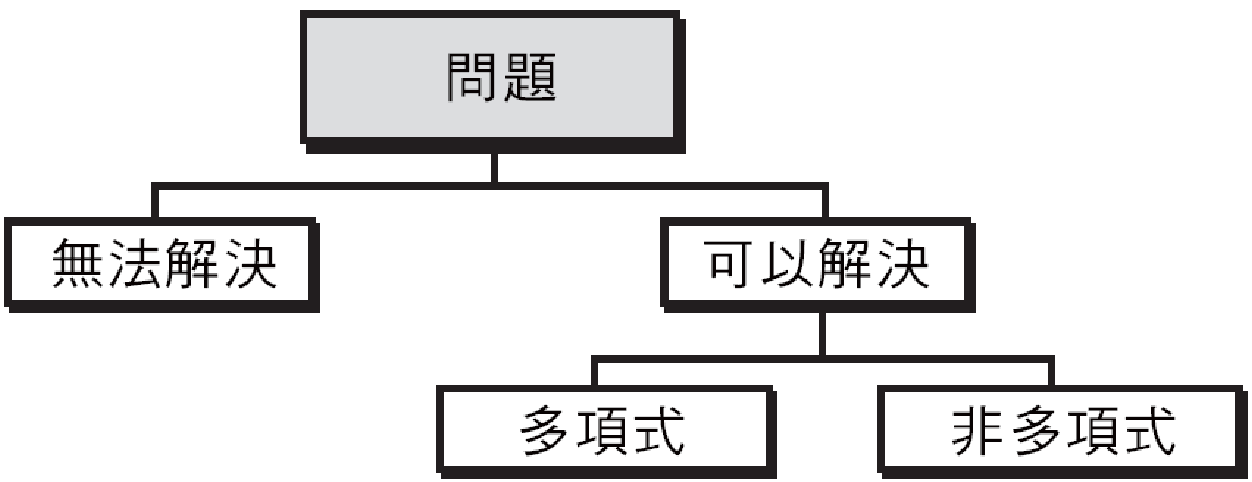
問題解決與演算設計

國立臺北科技大學資訊工程系
郭忠義

問題複雜度

- 一般在電腦科學中，問題可分成兩大類：
  - 無法解決的問題（unsolvable problems）
  - 可以解決的問題（solvable problems）：電腦花多少時間，某程式複雜度如何？執行所需時間、記憶體。
    - 多項式問題（polynomial problems）
    - 非多項式問題（non-polynomial problems）。

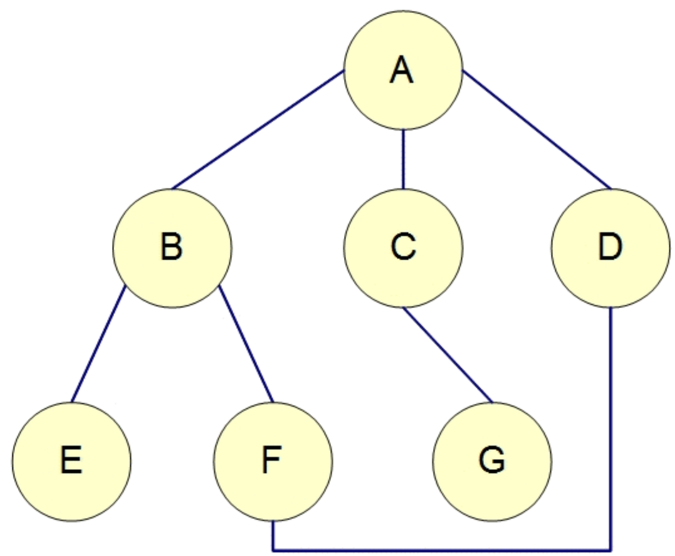


問題複雜度

- 大寫 O 符號
  - 不在意確切數量，在意數量等級。這種效率簡化稱大寫 O 符號（big-O notation）。指令敘述的數量是輸入數量的函數。
- 多項式問題
  - 若程式具 O(log n)、O(n)、O(n2)、O(n3)、O(n4)、 或O(nk)（k 為常數）的複雜度，稱多項式（polynomial problems）。
- 非多項式問題
  - 若程式複雜度大於多項式，例如：O(10^n) 或 O(n!)，則只有在輸入數量很小（小於 100）時才能夠解決問題。
  - 若輸入數量很大，則需數月才可得到非多項式問題（non-polynomial problems）的解決結果。

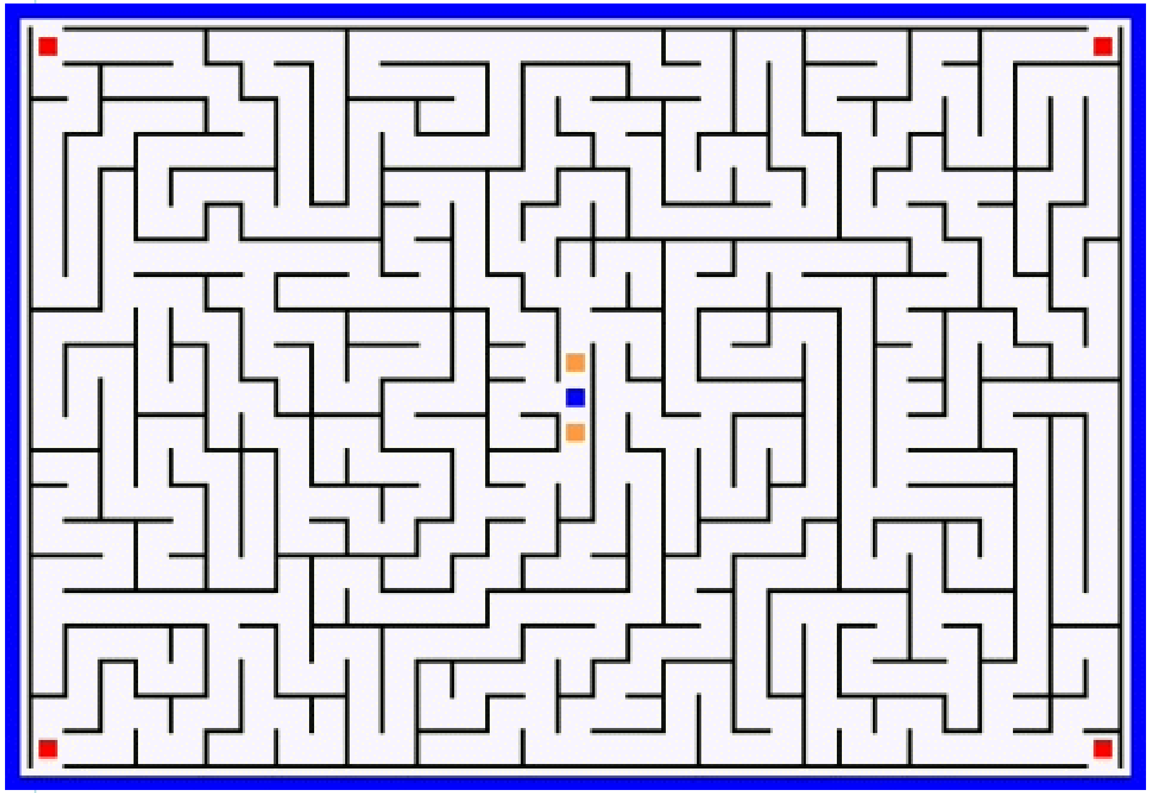
圖論－廣度優先搜尋法

- 廣度優先搜尋法 (Breadth-first Search)
  - 從圖某節點(vertex, node)開始走訪，接著走訪此節點所有相鄰且未拜訪過的節點，
  - 由走訪過節點繼續進行先廣後深搜尋。把同一深度(level)節點走訪完，再繼續向下個深度搜尋，直到找到目的節點或遍尋全部節點。
  - 廣度優先搜尋法屬於盲目搜索(uninformed search)，可利用佇列(Queue)處理。



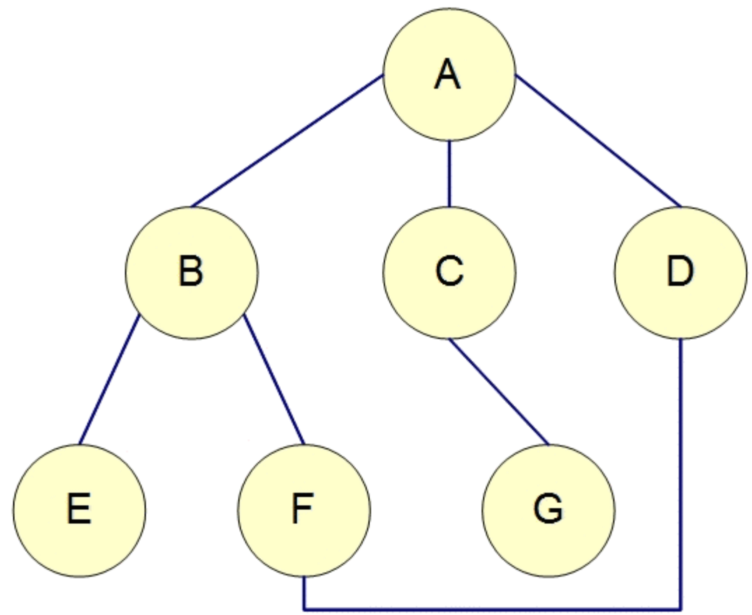
圖論－廣度優先搜尋法

```
procedure BFS(vertex s) {
    create a queue Q
    enqueue s onto Q
    mark s as visited
    while Q is not empty {
        dequeue a vertex from Q into v
        for each w adjacent to v {
            if w unvisited  {
                mark w as visited
                enqueue w onto Q
            }
        }
    }
}
```



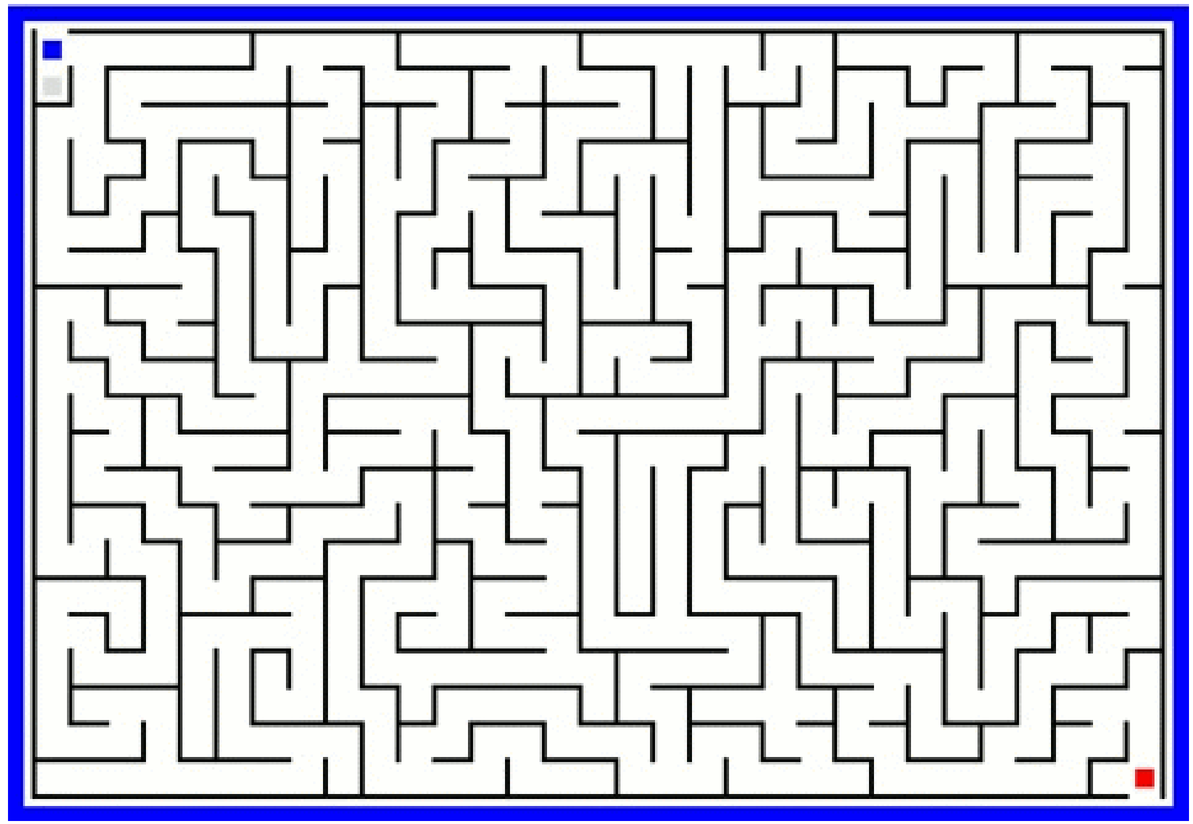
圖論－深度優先搜尋法

- 深度優先搜尋法 (Depth-first Search)
  - 從圖某節點開始走訪，先探尋邊(edge)上未搜尋的一節點，並儘可能深的搜索，直到該節點的所有邊上節點都已探尋；
  - 回溯(backtracking)到前一節點，重覆探尋未搜尋節點，直到找到目的節點或遍尋全部節點。
  - 屬盲目搜索，利用堆疊(Stack)處理。



圖論－深度優先搜尋法

```
procedure dfs(vertex v) {
    mark v as visited
    for each w adjacent to v {
        if w unvisited {
            dfs(w)
        }
    }
}
```

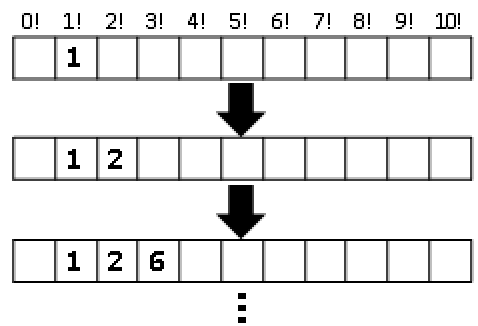


動態程式規劃

- 動態程式規劃(Dynamic Programming)
  - 階乘（ Factorial ）

```
void factorial(int f[10], int N) {
    int i=0;
    f[0] = 0;    f[1] = 1;
    for (i=2; i<=N; ++i) {
        f[i] = f[i-1] * i;
    }
}
```

```
void factorial(int N) {
    int i=0, f=1;
    for (i=2; i<=N; ++i) {
        f = f * i;
    }
}
```



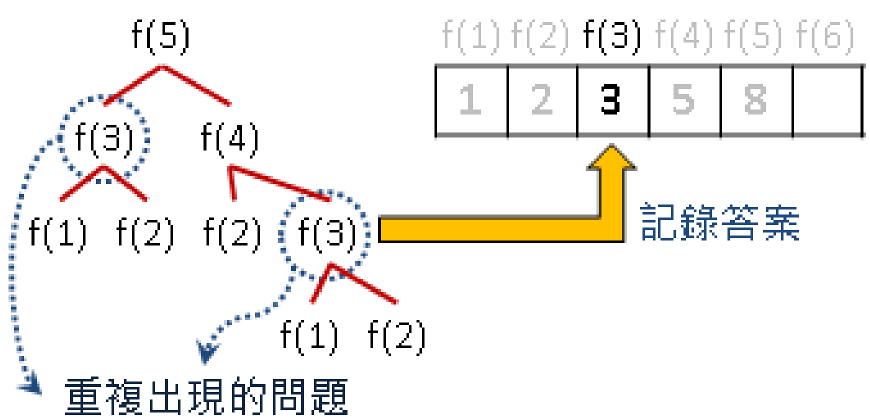
動態程式規劃

- 動態規劃是分治法的延伸。
  - 當遞迴分割出來的問題，一而再、再而三出現，就運用記憶法儲存這些問題的答案，避免重複求解，以空間換取時間。
  - 規劃的過程是反覆讀取資料、計算、儲存資料。
  - 時間複雜度O(N)
  - 空間複雜度O(N)

Recurrence

$$f(n) = \begin{cases} 1 & , \text{if } n = 1 \\ 2 & , \text{if } n = 2 \\ f(n-2) + f(n-1) & , \text{if } n \geq 3 \end{cases}$$

Divide and Conquer　　Memoization




```
int f(int n) {
    if (n == 0 || n == 1)        return 1;
    else            return f(n-1) + f(n-2);
}
```

動態程式規劃

- 動態規劃是分治法的延伸。

```
int f(int n, int s[]) {
    if (n == 0 || n == 1) return 1;
     // 用 0 代表該問題還未計算答案
     if (s[n]) return s[n];
     return s[n] = f(n-1, s) + f(n-2, s);
}
void stairs_climbing(){
    int stairs[20];
    for (int i=0; i<=20; i++) {
        stairs[i] = 0;
    }
    printf("%d", f(15, stairs));
}
```

貪婪演算法

- 貪婪演算法 (Greedy Algorithm) - 換零錢遊戲
  - 有 71個 1元，幣值分別為 29元、22元、5元、1元，請用最少零錢個數兌換零錢。
  - 局部解：29元 2 個，22元 0 個，5元2 個， 1元3個，2+2+3=7
- 動態程式規劃
  - 最佳解：29元 0 個，22元 3 個，5元1 個，1元0個，3+1=4
- 貪婪演算法不一定是最佳解，但效率高
  - 一種短視/近利/貪婪的想法，每一步都不管大局，只求局部解決
  - 透過一步步的選擇局部最佳解來得到問題解答。
  - 每個選擇是根據某種準則決定，前次決定不會影響後次決定。
- 動態規劃演算法可以求出最佳解，但效率略差

動態程式規劃

- 71元，最佳解：29元0個，22元3個，5元1個，1元0個，3+1=4
  - f(n) = min(1+f(n-29), 1+f(n-22), 1+f(n-5), 1+f(n-1))
  - 71元若以29元兌換，剩71-29=42元，總兌換數=42元可兌換個數+1
  - f(0)= 0, f(n) = 1 + min (f(n-c_1), f(n-c_2), …, f(n-c_k))
  - n>c_i, 1<i<k, c_i 是硬幣面額，k是總共有幾種面額
  - c_1=29, c_2=22, c_3=5, c_4=1
  - f(1)=1, f(2)=1+f(1)=2, f(3)=1+min(f(2))=3, f(4)=1+min(f(3))=4
  - f(5) = 1+min(f(5-5), f(5-1)) = 1+min(f(0), f(4)) = 1
    - 用一個1元換；或用一個5元換；之後可以如何換最少。
  - f(6) = 1+min(f(6-5), f(6-1)) = 1+min(f(1), f(5)) = 1+min(1, 1) = 2
  - f(7) = 1+min(f(7-5), f(7-1)) = 1+min((f2), f(6)) = 1+min(2, 2) = 3
  - f(8) = …
  - 要宣告 int f[n] 空間，換取計算時間，並計算各種可能性。

動態程式規劃

```
#include <stdio.h>
int f(int n, int coinType[], int k) {
    int p=0, i=0, coin=0, min_coin=0;
    int min_number[100]={0}, min_first_element[100];
    for (p=0; p<100; p++) min_number[p]=0;
    for (p=1;p<=n;p++){
        min_coin = n;
        for(i=0;i<k;i++){
            coin = coinType[i];
            if (((p-coin)>=0)&&((1+min_number[p-coin])<min_coin))
                min_coin = 1 + min_number[p-coin];
        }
        min_number[p] = min_coin;
        min_first_element[p] = coin;
    }
    for (p=1; p<=n; p++) {
        printf("(%d, %d)\n", p, min_number[p]);
    }
}
```

```
int main(){
    int coinType[10]={29, 22, 5, 1};
    int k=4;
    //int coinType[10]={5, 4, 2, 1};
    int n =71;  //8
    f(n, coinType, k);
    return 0;
}
```