Q1.

T [i..n] 主要的陣列   
B [i..n] 紀錄某個位置的元素是第 i 個被賦值的   
P [i..n] 紀錄被賦值元素的 index   
number 目前有多少的元素被賦值   
init\_val 初始值

虛擬碼：

procedure init{   
 number = 0  
 init\_val = -1 (自訂預設值)  
}

procedure store (i, v){   
 if not(B[i] < number and P[B[i]] = i) // i 元素未被賦值過

B[i] <- number, P[number] <- i, number++

T[i] <- v  
}

function val (i){   
 if (B[i] < number and P[B[i]] = i)   
 return T[i]   
 else   
 return init\_val   
}

Q2.

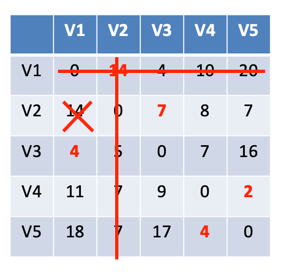
根據slide Ch12-1 P.19，畫出該頁TSP問題的三個版本(BFS、DFS、Best First Search)。



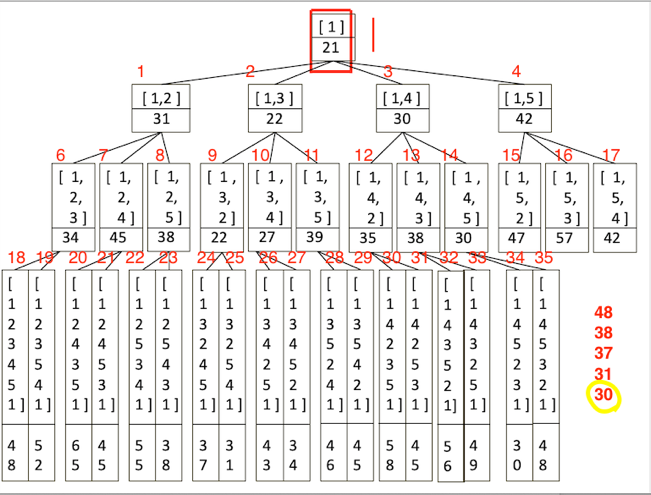
假設起點為[1]，首先將每個row的做小值相加，4+7+4+2+4=21，21就是lower bound。

BFS

再來考慮[1,2] [1,3] [1,4] [1,5]的lower bound。



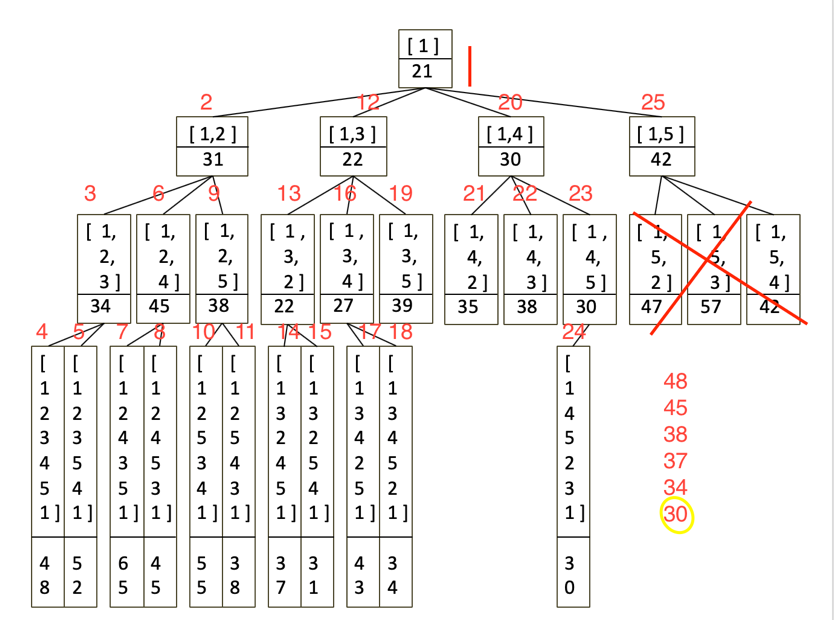
Lower bound 分別為[1,2]-31 [1,3]-22 [1,4]-30 [1,5]-42，再來考慮[1,2,3] [1,2,4] [1,2,5]的lower bound，利用上述類似的方式以BFS的走法做下去，可以得到lower bound為[1,2,3]-34 [1,2,4]-45 [1,2,5]-38，如此可以繼續做下去。



每個節點旁邊的數字代表走到幾步，右邊一排數字則是依序找到的最小值，因此30為最小值。

DFS

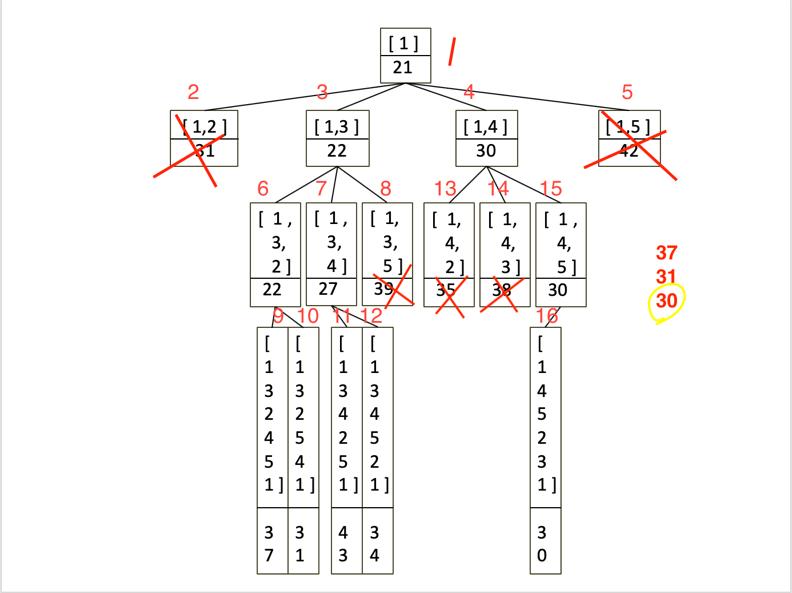
Lower bound的找法與上述呈現的方法一樣，只是利用DFS的步驟找lower bound。



每個節點旁邊的數字代表走到幾步，右邊一排數字則是依序找到的最小值，因此30為最小值。

Best First Search

Lower bound的找法與上述呈現的方法一樣，不過best first 是以最小lower bound依序去找。



每個節點旁邊的數字代表走到幾步，右邊一排數字則是依序找到的最小值，因此30為最小值。

Q3.

SOS with DP 原先表格為二維大小 T[i, k]

我們的演算法 SOS(i, k) 試圖解原先 T[i, k] 的問題

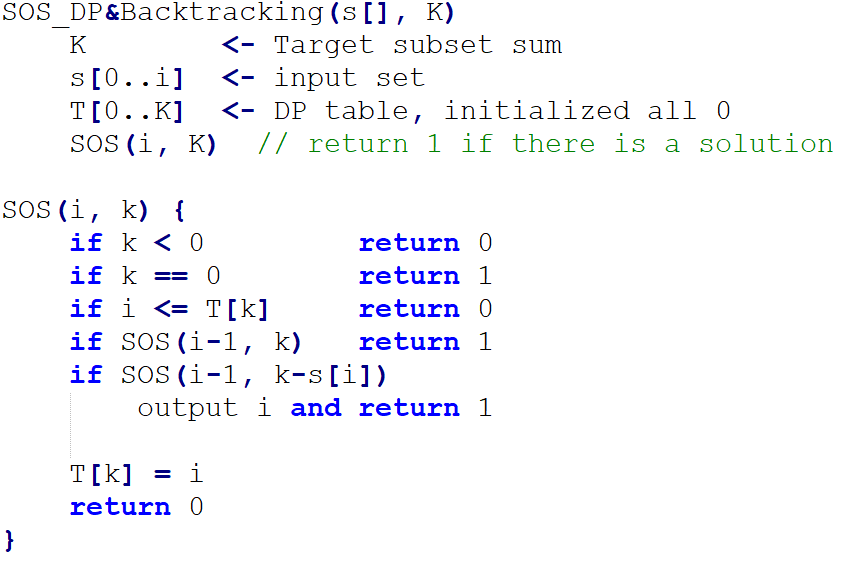
回傳值 1 表示有解，0 則無解

表格壓縮為一維大小 T[0..k]，初始值全為 0

執行中當 T[k] = j > 0

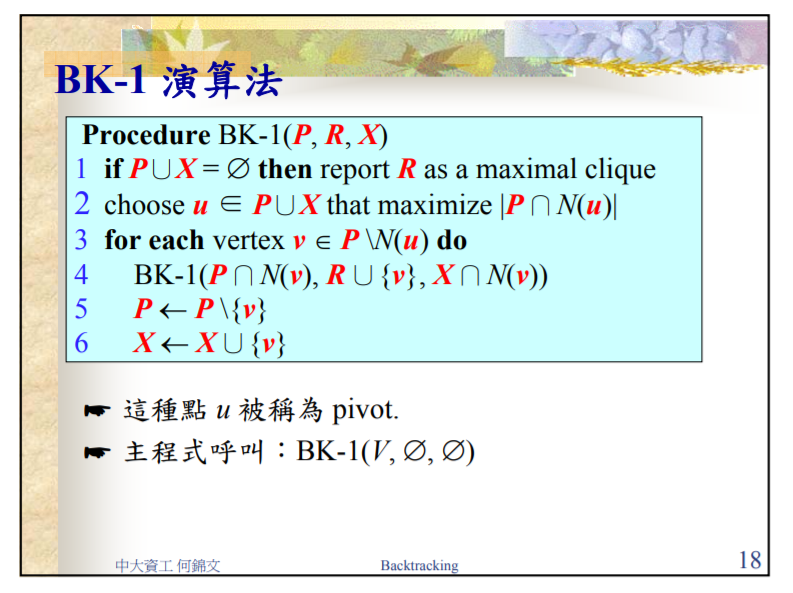
表示子問題 SOS(1, k)、SOS(2, k) … SOS(j, k) 無解

虛擬碼：



Q4.

BK-1 演算法：



Step1：計算upper bound of maximum clique，用Greedy Coloring，算完後顏色個數即為 upper bound

Step2：在 |( **P**∩N(**v**) ) ∪ ( **R**∪{**v**} )| **≧ upper bound** 的情況下才繼續執行，可刪去某些情況

Greedy Coloring

每個相鄰的點顏色須不一樣計算各點degree 依degree由大到小排序依順序替各點塗色

**Procedure** BK-1\_modif **(P, R, X, UB)**

**If** **P**∪**X** = Ø **then** report **R** as a **maximum** clique  
Choose **u** ∈ **P**∪**X** that maximize |**P**∩N(**u**)|  
**for each** vertex **v** ∈ **P** \ N(**u**) **do**   
 **if** | (**P**∩N(**v**)) ∪ (**R**∪{**v**}) | **≧ UB** //與原本唯一不同的地方  
 BK-1\_modif (**P**∩N(**v**), **R**∪{**v**}, **X**∩N(**v**))  
 **P** <- **P** \ {**v**}  
 **X** <- **X**∪{**v**}