Tree 是 Graph 中一種特例，也是一個非常重要的資料結構。由於節點在樹中的位置和分部方式會影響樹的整體深度，而不同的深度又會嚴重影響樹的效能，因此「維持樹的平衡」就是成為一個重要的課題。

對於一株 Binary Searching Tree 來說，當它有一個節點時，只有一種排法，兩個節點時有兩種，三個節點時有 1+2+2。

當有 N 個節點時，共有$\frac{(2N)!}{(N+1)!N!}$個節點。

Adelson, Velskii, Landis 三位學者提出一個能夠維持二元樹平衡的演算法→AVL Tree

當一株二元樹沒有受到任何規則限制任其生長時，往往會長出歪斜的樹。

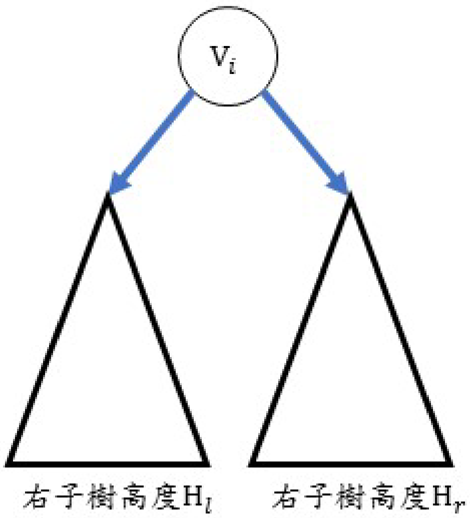
→當一株二元樹中有 n 個節點且$2^{n-1} \leq N < 2^n$

→$n = \lceil \log_2 N \rceil$

一株深度為 n 的二元樹是能夠容納 N 個節點的最小深度的二元樹。

AVL tree 的策略是不斷的對二元樹做檢查→檢查所有節點的「balance factor」（平衡要素），當 bf 只是樹不在平衡時便對樹的結構進行調整，以維持其平衡。

balance factor 定義如下：



$$\mathrm{b.f\ of}\ V_i = H_r - H_l$$

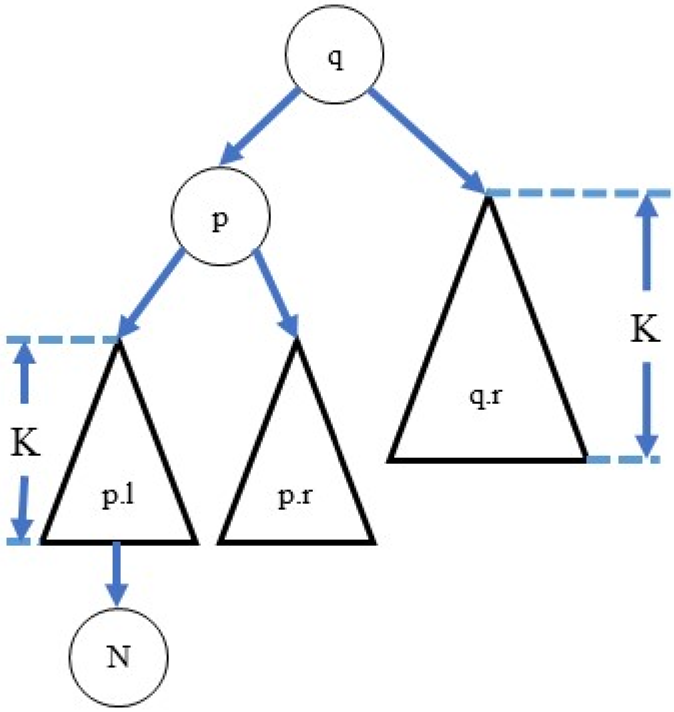
當一株二元數中所有節點的 bf 皆在±1之間，我們稱該數在平衡的狀態。

如何調整 tree 的結構？

AVL tree 提供了 4 種調整二元數的方式

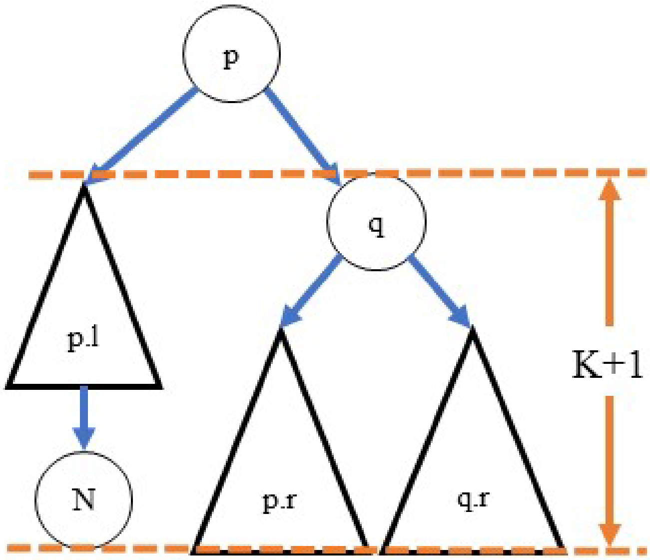
$$\begin{cases} LL \\ RR \\ LR \\ RL \end{cases} \text{rotation}$$

- **LL retation**

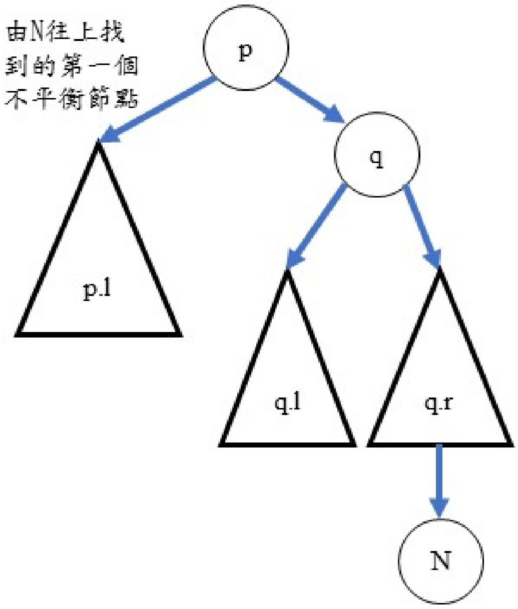


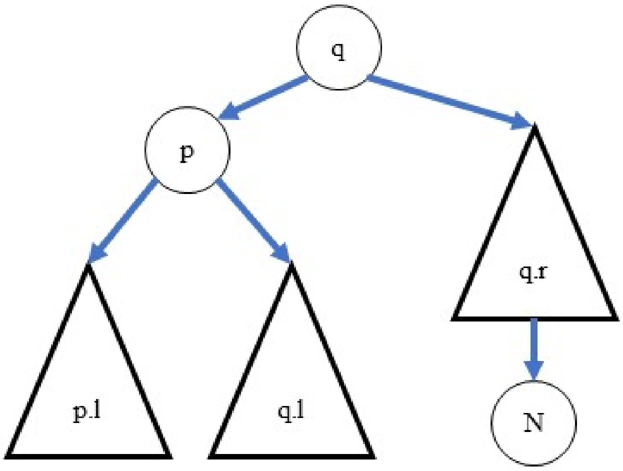
N 進入之前所有節點的$|bf| \leq 1$；因此調整前$\mathrm{p.l} < \mathrm{p} < \mathrm{p.r} < \mathrm{q} < \mathrm{q.r}$。

調整之後則$\mathrm{p.l} < \mathrm{p} < \mathrm{p.r} < \mathrm{q} < \mathrm{q.r}$維持原來大小關係。



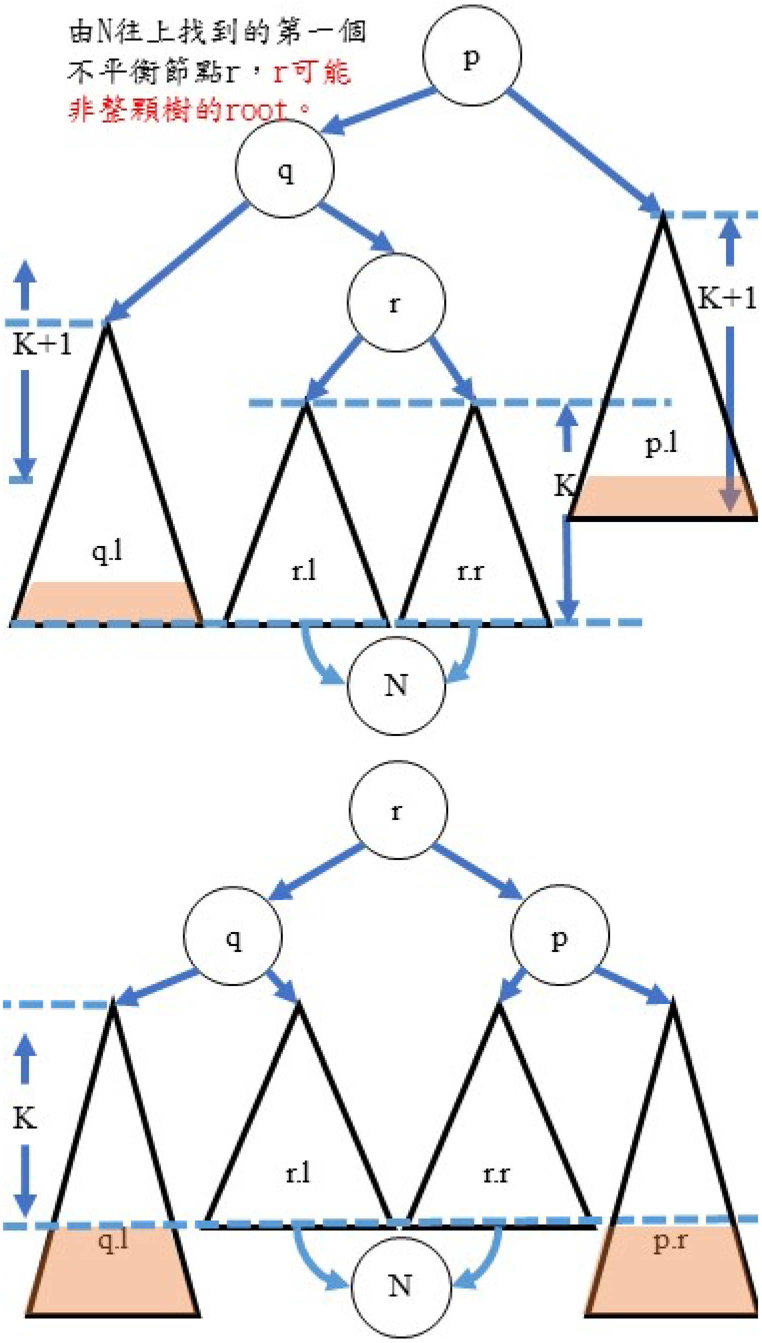
- **RR rotation**





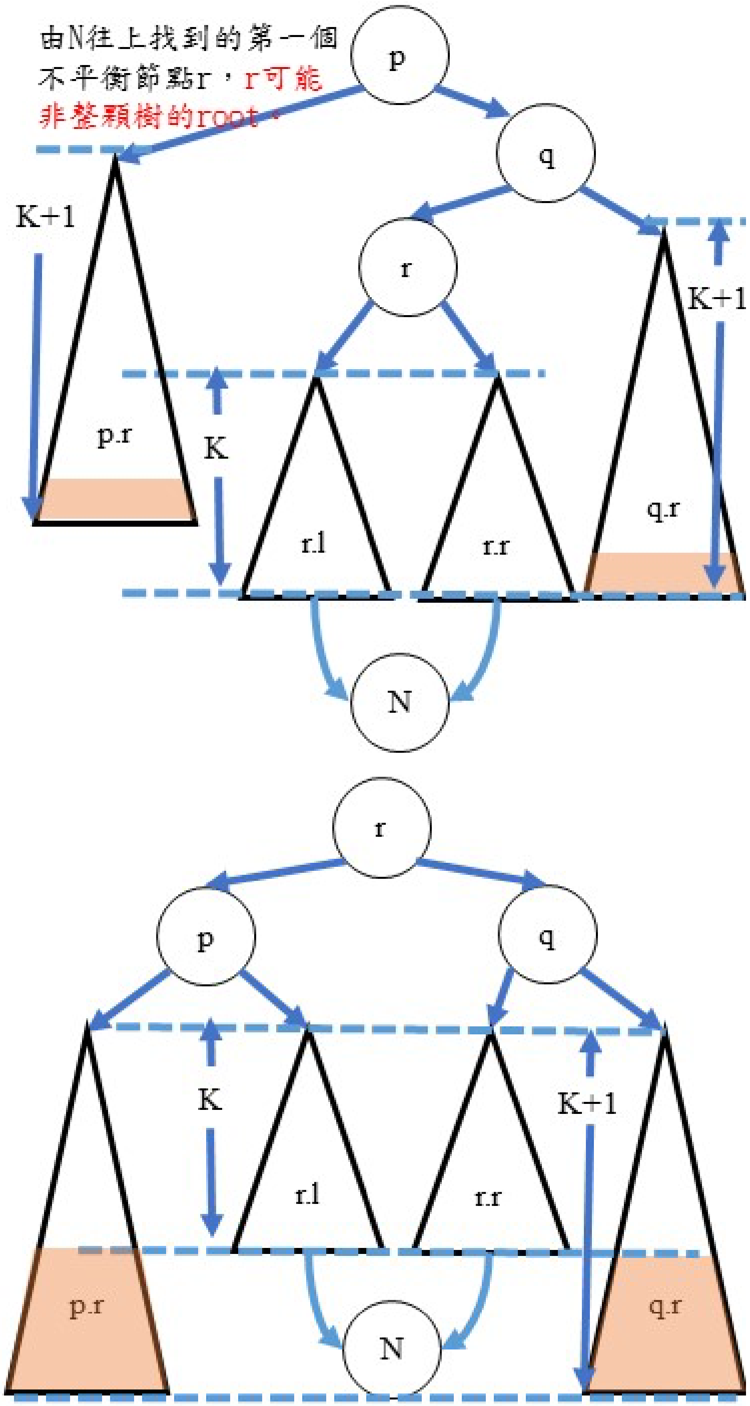
高度差，下一個如果是+1 的話表示右子樹有問題→使用 LR rotation；如果是-1 的話表示佐子樹有問題，使用 LL rotation。

- **LR rotation**



LR rotation 及 LL rotation 的判斷方式：第一個遇到的有問題的 node 為-2 的高度差，下一個如果是+1 的話表示右子樹有問題→使用 LR rotation；如果是-1 的話表示佐子樹有問題，使用 LL rotation。

- **RL Rotaiton**



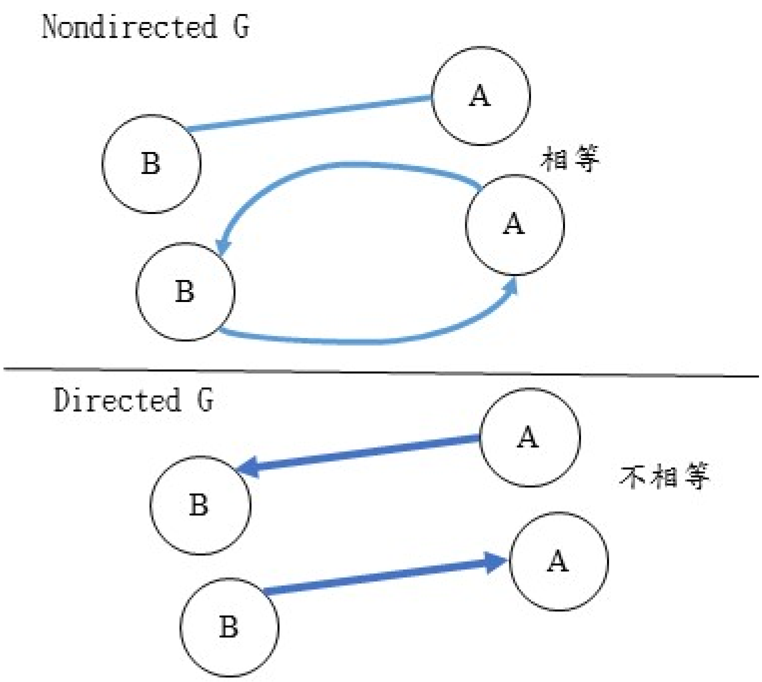
由N往上找到的第一個
不平衡節點r，r可能
非整顆樹的root。
p
q
r
K+1
p.r
K
r.l
r.r
K+1
q.r
N
r
p
q
K
r.l
r.r
K+1
p.r
N
q.r

➢ Graph 圖形結構

Tree 是圖形中的一個特例。如同 Tree 一般 Graph 也具有以下的結構：

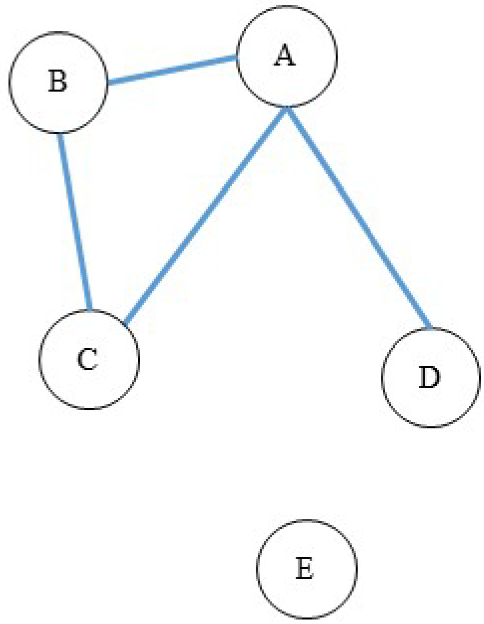
G = (V, E)；V 為頂點的集合；E 為邊的集合

與 Tree 不同的是 Graph 又可分為：$\begin{cases} \textit{Directed graph 方向性} \\ \textit{Nondirected graph 非方向性} \end{cases}$

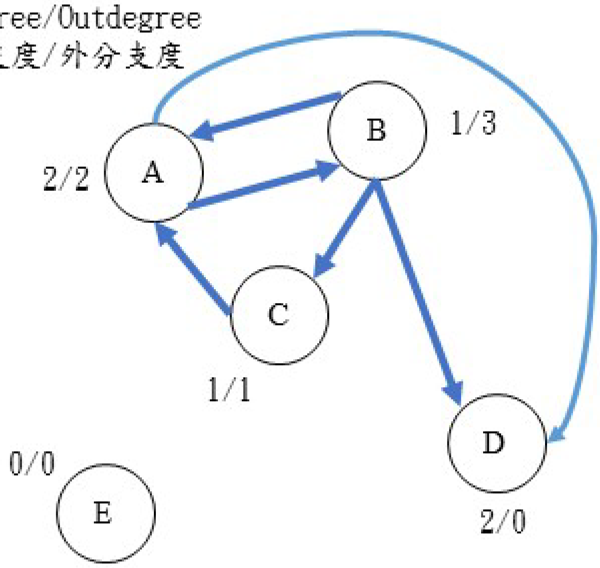


在下圖中

- Ⓐ與 Ⓑ，Ⓒ為相鄰（adjacent）
  - 直接有 e（A,B）與 e（A,C）將 Ⓐ與 Ⓑ，Ⓒ相連
- Ⓐ與 Ⓓ為相連（connected）
- Ⓐ到 Ⓓ如果走 e（A,B），e（B,D）因為該路徑經過這兩條 edges，則 Length of path 為 2
- 承上，e（A,B），e（B,D）構成一條由 Ⓐ到 Ⓓ的路徑（path）
- 另外 Ⓔ與 G 中其他節點不相連（disconnected）



對於一個Directed G
Indegree/Outdegree
內分支度/外分支度



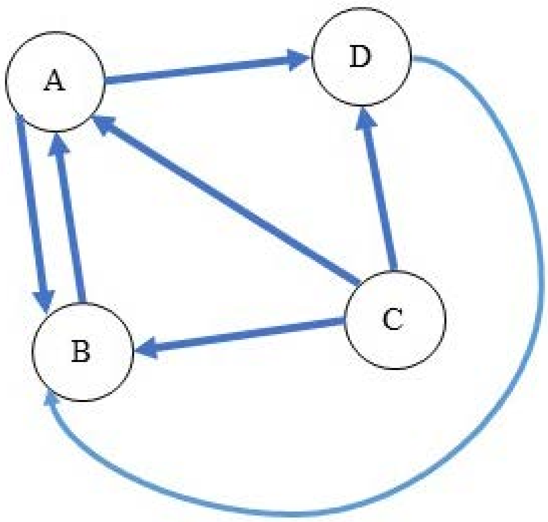
Graph 的資料表示法（representation of Graph）

1. Adjacency Matrix Representation

對於一個具有 n 個 nodes 的 G，可以使用一個 n*n 的矩陣（Matrix）M 去表示他，這時，矩陣某個元素（element）$a(i,j)$為位在第 i 行（row）第 j 列（column）的元素，$a(i,j)=1$代表$v_i \rightarrow v_j$的關係成立，也就是是$e(v_i, v_j)$說存在於$\{E\}$中。

反之，當$a(i,j)=0$代表v_i不能直接到達v_j。

$$\begin{cases} 當G為方向性圖形時a(i,j)不必等於a(j,i) \\ 當G為非方向性圖形時a(i,j)=a(j,i) \end{cases}$$



	A	B	C	D
A	0	1	0	1
B	1	0	0	0
C	1	1	0	1
D	0	1	0	0

$$M^2 = \begin{vmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{vmatrix} \begin{vmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 2 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{vmatrix}$$

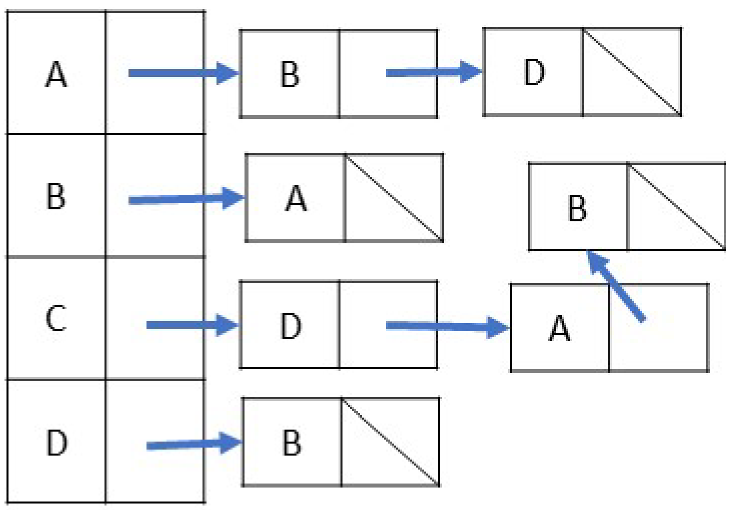
在M^2中 a（3,2）代表由 Ⓒ到 Ⓑ長度為 2 的路徑有兩條。

相鄰矩陣的優點在於可以直接M^x中$a(y,z)$這個元素的值，判斷v_y到v_z長度為 x 的路徑數量。

缺點在於，當 G 中 E 的 size 遠小於V^2時，也就是 G 是一個連結性相對地弱/疏的圖形時，size 為n^2的矩陣 M 中，就會有大量的元素值為 0→形成空間的浪費。(當元素$a(i,j)$值為 1 時，才會有一條邊e(i,j)存在。)

2. Adjacency List Representation

相對於相鄰矩陣，相鄰串列則眼著在空間的使用效率上，因此不使用 2D array，而使用 linked list。



利用 linked list 的好處在於可以自由地增/刪節點，僅只在需要時進行配置，因此節省使用空間。

缺點是總是必須對 list 依序拜訪（traversal），才能完全掌握資料的內容。

Graph 非常適合用來表現具有複雜結構的事象。原本複雜的文字、數據資料經過 Graph 表示之後，常常就能一目了然資料間相對關係，甚至令人容易發現某

些 pattern（模式）。

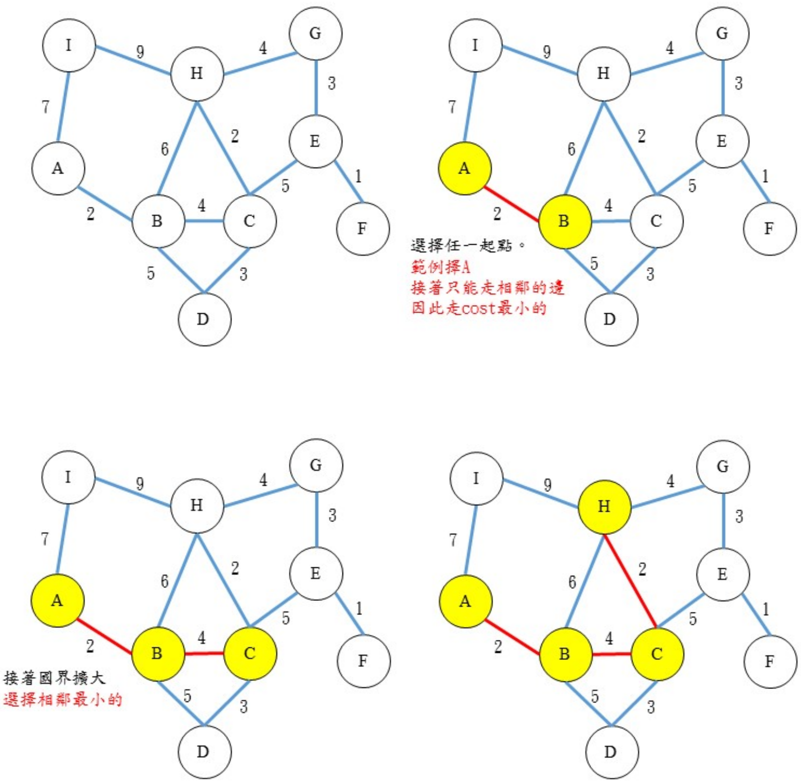
如果再 edge 上附加上數據，這時 Graph 會帶有更多的資訊，這些 edge 上的數字可能代表兩端點間的 cost、移動需要的時間……。

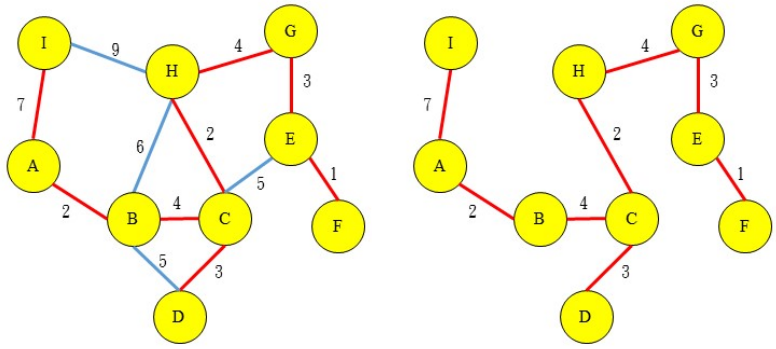
有時，為了再簡化 Graph 的複雜度，並且移除掉可能的循環性，對於具有 N 個節點的 G，會保留 N-1 條 edges，並將所有節點連結起來，稱為擴張樹（Spanning tree）。

再將 edge 上的 cost 納入 spanning tree 的考量中→設法找出會留下的 N-1 條 edge 上 cost 的總和為最小的問題，稱之為最小花費擴張樹（Minimum cost spanning tree）的問題。

常見的解法有$\begin{cases} Prim's\ algorithm \\ Kruskal'salgorithm \end{cases}$。

➢ *Prim's algorithm*





從 A 出發找到最小的（2）→B。以此類推。

	A	B	C	D	E	F	G	H	I
A	0	2	∞	∞	∞	∞	∞	∞	7
B	2	0	4	5	∞	∞	∞	6	∞
C	∞	4	0	3	5	∞	∞	2	∞
D	∞	5	3	0	∞	∞	∞	∞	∞
E	∞	∞	5	∞	0	1	3	∞	∞
F	∞	∞	∞	∞	1	0	∞	∞	∞
G	∞	∞	∞	∞	3	∞	0	4	∞
H	∞	6	2	∞	∞	∞	4	0	9
I	7	∞	∞	∞	∞	∞	∞	9	0

1. 選責任一點作為起點，假設為v_i刪除第i行、標記第i列。
2. 由目前已標記的列中，如果其中尚有未行被刪除的元素，比較其值選擇只有具有最小 cost 者，假設為$e(j,k)$，則v_j將為下一個起點，回到 1.。否則作完。

刪行、留列

j 為行（橫）、k 為列

➢ Kruskal's algorithm

Kruskal 使用了 edge list 來建立 Graph 的最小擴張術。

首先將所有的 edge 根據其 weighting value（加權值）由小到大做 sorting。這也成為演算法中決定其 performance 的關鍵。

接下來依序檢查各 edge 並且建立一個由原點子集合所構成的集合如下：

一開始時，$\mathcal{T} = \emptyset,\ \mathcal{V} = \mathcal{u} = \{\{v_1\},\{v_2\},\{v_3\},\ldots,[v_n]\}$

→$\mathcal{T}$代表已加入擴張樹的頂點集合；$\mathcal{u}$代表尚未被加入$\mathcal{T}$（＝尚未被拜訪過的）的頂點集合。

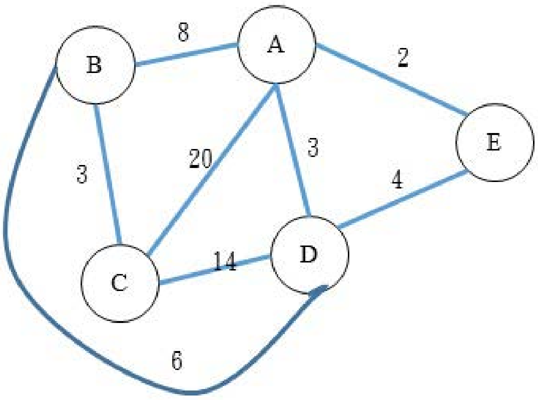
檢查 edge 如下，假設目前 edge 為e_{ij}

如果 $i,j \in \mathcal{T}$ $\begin{cases} \text{如果}i,j \in \mathcal{T}\text{中同一子集，do nothing} \\ \text{如果}i,j\text{分} \in \mathcal{T}\text{中兩個子集}\mathbb{y},\mathbb{z}\text{，則將}\mathbb{y},\mathbb{z}\text{聯集} \end{cases}$

如果i,j 中一者$\in \mathcal{T}$一者$\in\ \mathcal{u}$ （假設$i \in \mathcal{T}, j \in\ \mathcal{u} \rightarrow \mathcal{u} = \mathcal{u} - \{j\}$並將$j$加入到$\mathcal{T}$中$i$所屬的子集，然後將$e_{ij}$加入到$E_{\mathcal{T}}$中。

如果$i,j \notin \mathcal{T}$則$\mathcal{T} = \mathcal{T} + \{i,j\}, \mathcal{u} = \mathcal{u} - \{i\} - \{j\}$

重複以上步驟，直到$\begin{cases} E = \emptyset \\ E_{\mathcal{T}}\ \text{中還有}N-1\ \text{條}edge \end{cases}$



初始值

$E = \{\overline{AE}, \overline{AD}, \overline{BC}, \overline{DE}, \overline{BD}, \overline{AB}, \overline{CD}, \overline{AC}\}$

$\mathcal{U} = \{\{A\}, \{B\}, \{C\}, \{D\}, \{E\}\}$

$\mathcal{T} = \emptyset$

$E_{\mathcal{T}} = \emptyset$

第一步：檢查$\overline{AE}$，

$\mathcal{T} = \{\{A, E\}\}$

$\mathcal{U} = \{\{B\}, \{C\}, \{D\}\}$

$E_{\mathcal{T}} = \{\overline{AE}\}$

第二步：檢查$\overline{AD}$

$\mathcal{T} = \{\{A, D, E\}\}$

$\mathcal{U} = \{\{B\}, \{C\}\}$

$E_{\mathcal{T}} = \{\overline{AE}, \overline{AD}\}$

第三步：檢查$\overline{BC}$

$\mathcal{T} = \{\{A, D, E\}, \{B, C\}\}$

$\mathcal{U} = \emptyset$

$E_{\mathcal{T}} = \{\overline{AE}, \overline{AD}, \overline{BC}\}$

第四步：檢查$\overline{DE}$，Do nothing（因為 D，E 同屬$\mathcal{T}$中同一子集）

第五步：檢查$\overline{BD}$，

（B，D 屬$\mathcal{T}$中不同子集

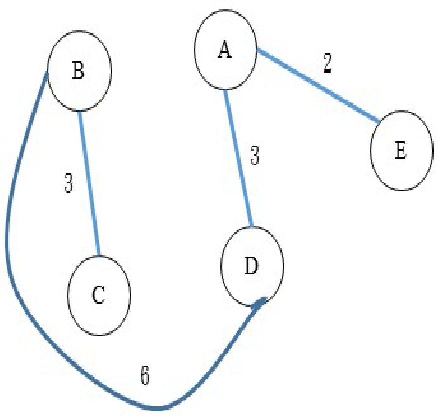
$\mathcal{T} = \{A, B, C, D, E\}$

$E_{\mathcal{T}} = \{\overline{AE}, \overline{AD}, \overline{BC}, \overline{BD}\}$

最後檢查

G 共有 5 個 V

$E_{\mathcal{T}}$中已找到 4 條 edge 故作完。



Kruskal 與 Prim 兩個演算法都選擇演算法執行過程中「目前」所能找到的最佳解（local optimal solution）最後產生最終的最佳解→皆屬於 Greedy Method（貪婪演算法）的一種。

Kruskal 的執行時間與$E\log E$成正比，而 Prim 由於是對 N*N 的 2D array 做刪行加列的操作，因此時間會與N^2成正比。

➢ **Kruskal v.s. Prim**

當 G 中 E 的數目接近 N。（對於一個 connected G，$\frac{N(N-1)}{2} \geq E \geq N-1$）

→當 E~N 則$E\log E \sim N\log N < N^2$Kruskal 較好

當 G 中邊的數量相對地多，G 被相對密集地連結在一起時$E \sim N^2$

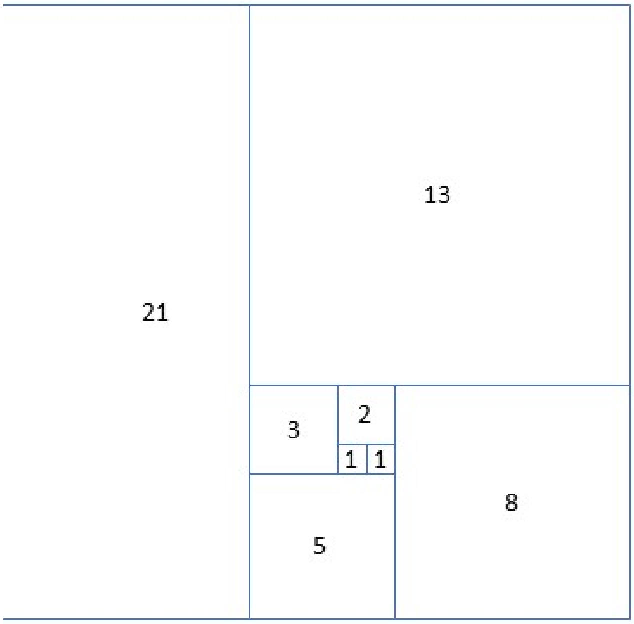
則$E\log E \sim N^2\log N^2 = 2N^2\log N > N^2$

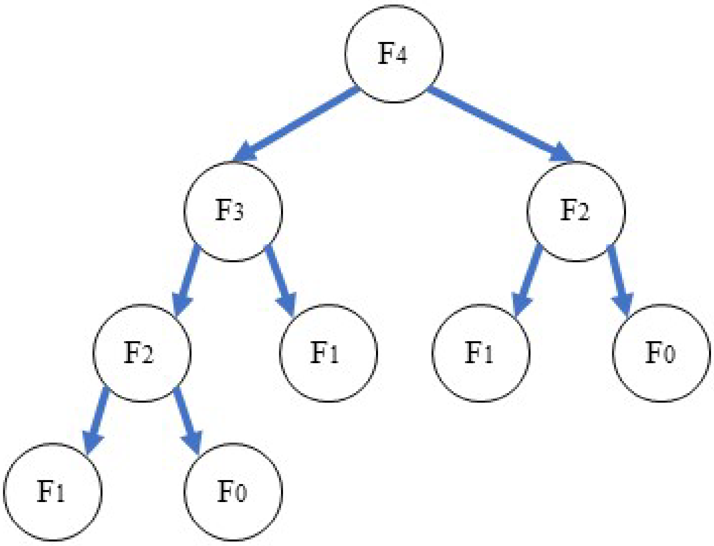
➢ **Fibonacci**

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2}, when\ n \geq 2 \end{cases}$$

寫程式去求費氏數列的方法？

1. Iteration 疊代
2. Recursion 遞迴






```
recursion_Fib(n)
{
    if(n==0)
        return 0;
    else
    {
        if(n==1)
            return 1;
        else
        {
            x=recursion_Fib(n-1)+recuension_Fib(n-2)
            return x;
        }/*else if n==1*/
    }
}
```

用 recursion 求F_n需要 n 次加法？

求F_2→1

F_3→2

F_4→4

$F_n = F_{n-1} + F_{n-2} + 1$。

- **Dynamic Programming**

分割各個擊破分割各個擊破

- →發揮 Divide and Conquer 的精神，給予一個系統化的架構。

→給予一個隨運算進行而改變的限制條件，因此縮減運算量。

Divide

對於 Dynamic Programming（以下簡稱 D.P.）來說，所謂「發揮 Divide and Conquer」的精神，代表的是：極端地、窮舉地求出所有可能的較小子問題的解，並利用這些解去組合出較大母問題的解。

Conquer

為求F_n需知F_{n-1}與F_{n-2}（上排為費氏數列，下排為需要加法數）

→宣告一個 array 存取過去的解。

F_n	F_{n-1}	F_{n-2}	F_{n-3}	……	F_4	F_3	F_2	F_1	F_0
n-1	n-2	n-3	n-4		3	2	1	0	0

例如求F_{14}時需要加法數用 D.P.需要 13 次。

若使用遞迴的方法則需要 609 次加法。

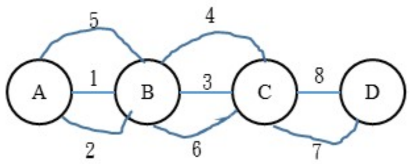
Problem：

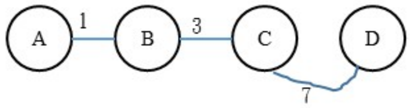
Find the “best” way to do something.

例如找最短路徑=找最好走法

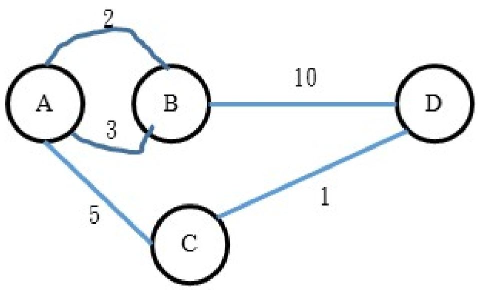
在某些情況下，例如 Graph 中利用 Greed Method 的 Prim’s and Kruskal’s algorithms 去求最小擴張樹。

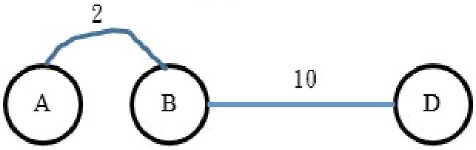
Greed Method 直觀又容易理解，例 1

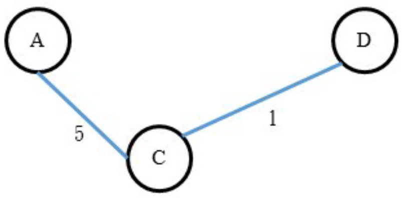




但也常常失敗，例 2，上為題目中為失敗例子，下圖為成功例子。







- 何時可以用 D.P.？
  - 當問題可以拆解
  - 並且拆法有限
  - 母問題的解可以由子問題的解拼湊出來

➢ Ex 1 Knapsack Problem

Capacit

一個小偷帶著長度為 n 的 one-dimensional array 作為他的包包。面前有 M 種無限多個，共有其不同 size 與 value 的 items→如何選擇達到最大收益？

Item	A	D	C	D	E
Size	3	4	7	8	9
Value	4	5	10	11	13

```
for(j=1 ; j<=M ; j++)//M為物件種類
{
    for(i=1 ; i<=N ; i++)//N為包包大小
    {
        if(i-size[j] >= 0)//表示包包目前的容量還放得下 一個j物件
        {
            if(cost[i-size[j]]+value[j]>=cost[i])
            //cost[i-size[j]]存放入j之前包包內的總價值
            //value[j]為一個j的價值
            //cost[i]沒放j前包包的總價值
            {
                best[i] = j;
                cost[i] = cost[i-size[j]+value[j];
            }/*end of if i*/
        }
    }
}
```

紅框處為演算法重點所在，稱為三相操作 triple operation。

以下數值代表目前包包的 best，英文字為放的 item，算完每一格的數量後，從第 13 格往回推算要放那些東西。

<table>
<tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td></tr>
<tr><td>0</td><td>0</td><td>4A</td><td>4A</td><td>4A</td><td>8A</td><td>8A</td><td>8A</td><td>12A</td><td>12A</td><td>12A</td><td>16 A</td><td>16A</td></tr>
<tr><td></td><td colspan="3">A</td><td colspan="3">A</td><td colspan="3">A</td><td colspan="3">A</td></tr>
</table>

看 B

<table>
<tr><td>0</td><td>0</td><td>4A</td><td>5B</td><td>5B</td><td>8A</td><td>9B</td><td>10B</td><td>12A</td><td>13B</td><td>14B</td><td>16A</td><td>17B</td></tr>
<tr><td colspan="3">A</td><td colspan="3">A</td><td colspan="3">A</td><td colspan="4">B</td></tr>
</table>

看 C

<table>
<tr><td>0</td><td>0</td><td>4A</td><td>5B</td><td>5B</td><td>8A</td><td>10C</td><td>10C</td><td>12A</td><td>14C</td><td>15C</td><td>16A</td><td>18C</td></tr>
<tr><td colspan="4">B</td><td colspan="9">C</td></tr>
</table>

看 D

<table>
<tr><td>0</td><td>0</td><td>4A</td><td>5B</td><td>5B</td><td>8A</td><td>10C</td><td>11D</td><td>12A</td><td>14C</td><td>15D</td><td>16D</td><td>18C</td></tr>
<tr><td colspan="4">B</td><td colspan="9">C</td></tr>
</table>

看 E

<table>
<tr><td>0</td><td>0</td><td>4A</td><td>5B</td><td>5B</td><td>8A</td><td>10C</td><td>11D</td><td>13E</td><td>14C</td><td>15D</td><td>17E</td><td>18E</td></tr>
<tr><td colspan="4">B</td><td colspan="9">E</td></tr>
</table>

以上為結果。

➢ Ex 2 Matrix Chain Production

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{vmatrix}_{4*2} \begin{vmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{vmatrix}_{2*3} |C|_{3*1}|D|_{1*2}|E|_{2*2}|F|_{2*3}$$

→求出最後結果共需做 n 次乘法？

|A||B|共需 4*2*3=24 次乘法。

若 AB 矩陣分別為 x*y 及 y*z 個元素，得到 C 矩陣（x*z）。共需 x*y*z 次乘法。

Q：面對一連串矩陣連乘時，如何找出使得所做乘法次數為最少的結合順序？

```
for( i=1 ; i<=N ; i++)//對角線又上半所有元素皆為MAX
    for( j=i+1 ; j<=N ; j++)
        cout[i][j]=Max;
for( i=1 ; i<=N ; i++)//對角線皆為0
    cost[i][i]=0;
for( j=1 ; j<N ; j++)//j控制檢查範圍
{
    for( i=1 ; i<=N-j ; i++)//i控制從哪個矩陣開始檢查
    {
        for( k=i+1 ; k<=i+j ; k++)//k在控制切在何處
        {   //r[i]代表第i個陣列的row數
            t=  cost[i][k-i]+//從i開始至k-1最少的乘法次數
                cost[k][i+j]+//從k開始至i+j最少的乘法次數
                r[i]*r[k]*r[i+j+1];//左邊的矩陣乘右邊矩陣需要乘法次數

            if(t<cost[i][i+j])//如果t比目前已知好
            {
                cost[i][i+j]=t;//triple operation置換
                best[i][i+j]=k;
                //對於第i個矩陣，連乘列第i+j個矩陣，目前已知切在k-1與k之間最好
            }/*end of if*/
        }/*end of for k*/
    }
}
```

r 陣列

A	B	C	D	E	F	
4	2	3	1	2	2	2

第一次、 僅有兩矩陣鄉城所需之次數（EX：AB、BC、CD......）

	A	B	C	D	E	F
A	0	24/B				
B		0	6/C			
C			0	6/D		
D				0	4/E	
E					0	12/F
F						0

第二次、 對黃色格子來說的計算方式為

Min（0+6+4*2*1, 24+0+4*3*1）

	A	B	C	D	E	F
A	0	24/B	14/B			
B		0	6/C	10/D		
C			0	6/D	10/D	
D				0	4/E	10/F
E					0	12/F
F						0

第三次、 Min（0+10+4*2*2,24+6+4*3*2+14+0*4*2*1）

	A	B	C	D	E	F
A	0	24/B	14/B	22/D		
B		0	6/C	10/D	14/D	
C			0	6/D	10/D	19/D
D				0	4/E	10/F
E					0	12/F
F						0

第四次、Min（0+14+4*2*2,24+10+4*3*2,14+10+4*1*2,22+0+4*2*2）

	A	B	C	D	E	F
A	0	24/B	14/B	22/D	30/B	
B		0	6/C	10/D	14/D	22/D
C			0	6/D	10/D	19/D
D				0	4/E	10/F
E					0	12/F
F						0

第五次、

	A	B	C	D	E	F
A	0	24/B	14/B	22/D	26/D	36/D
B		0	6/C	10/D	14/D	22/D
C			0	6/D	10/D	19/D
D				0	4/E	10/F
E					0	12/F
F						0

Min（0+22+4*2*3,24+19+4*3*3,14+10+4*1*3,22+12+4*2*3,26+0+4*2*3）

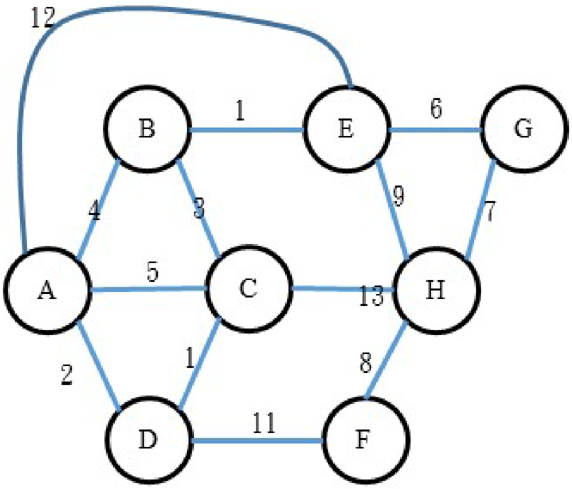
=Min（46,79,36,58,50）

結果：最少的乘法次數為 36 次，並且切在 D。

→（ABC）（DEF）→查表可知（ABC）要切在 B、DEF 要切在 F→（A（BC））（(DE）F）

➢ Shortest Path（最短路徑）

在邊上具有加權值的圖形中，常需要求取兩節點間的最短路徑。或更進一步求某一特定節點到 G 中其他所有節點的最短路徑。這時可以使用：Dijkstra's algorithm 去求取上述問題的解。



首先需要將 G 轉成 Adjacency Matrix 的形式$a[i][j]$。

	A	B	C	D	E	F	G	H
A	0	4	5	2	12	∞	∞	∞
B	4	0	3	∞	1	∞	∞	∞
C	5	3	0	1	∞	∞	∞	13
D	2	∞	1	0	∞	11	∞	∞
E	12	1	∞	∞	0	∞	6	9
F	∞	∞	∞	11	∞	0	∞	8
G	∞	∞	∞	∞	6	∞	0	7
H	∞	∞	13	∞	9	8	7	0

∞：表示對應兩點間沒有 edge

有值：表示對應兩端點i,j間只有 edge 直接連結，且其加權值被記錄在$a[i][j]$與$a[j][i]$的元素中。

初始化：

```
for(i=1 ; i<=N ; i++)
{
    visited[i] = 0;//代表該節點是否已被拜訪過的flag
    length[i] = a[start][i];
    pass_by[i] = start;
}/*start代表特定出發節點的index*/
visited[start] = 1;
```

實際運算：

```
如果visited中尚有節點未被拜訪過
{
    由未被拜訪過的節點中，選擇節點i使得length[i]為其最小者。
    visited[i] = 1;
    for(j=1; j<=N ; j++)
    {
        if(visited[j] && ((length[i]+a[i][j])<length[j]))//三項操作
        {//length[j]：目前已知最佳解；length[j]+a[i][j]：經由i的走法
            length[j] = length[i]+a[i][j];
            pass_by[j] = i;
        }
    }
}/*end of 如果*/
```

Example：以 C 為 start

	A	B	C	D	E	F	G	H
visited：	0	0	1	0	0	0	0	0
length：	5	3	0	1	∞	∞	∞	13
pass_by：	C	C	C	C	C	C	C	C

找到第一個最小的為 D，所以 C、D 拜訪過不考慮。若 length[i]+a[i][j]小於原本 length 的值則替換。

①+	2		1			11		∞

		B	C	D	E		G	H
visited：		0	1	1	0		0	0
length：		3	0	1	∞	12	∞	13
pass_by：		C	C	C	C		C	C

找到最小的為 A，所以 A、C、D 拜訪過不考慮。若 length[i]+a[i][j]小於原本 length 的值則替換。因此面對 C→E 路徑時，原來的∞代表沒有路徑相連，現在如果經由 A，則 AE 可連到 E。至於到 A 則要經由 D（寫在 pass_by 中），最後 C 可直達連到 D。C→D→A→E

③+	0						∞	∞

	A	B	C	D		F	G	H
visited：	1	0	1	1		0	0	0
length：	3	3	0	1		12	∞	13
pass_by：	D	C	C	C		D	C	C

找到最小的為 B，所以 A、B、C、D 拜訪過不考慮。若 length[i]+a[i][j]小於原本 length 的值則替換。

	A	B	C	D		F	G	H
③+					1		∞	∞
	A	B	C	D		F	G	H
visited：	1	1	1	1		0	0	0
length：	3	3	0	1		12	∞	13
pass_by：	D	C	C	C		D	C	C

找到最小的為 E，所以 A、B、C、D、E 拜訪過不考慮。若 length[i]+a[i][j]小於原本 length 的值則替換。

	A	B	C	D	E	F		H
④+	12			∞	0		6	
	A	B	C	D	E	F		H
visited：	1	1	1	1	1	0		0
length：	3	3	0	1	4	12		13
pass_by：	D	C	C	C	B	D		C

找到最小的為 G，所以 A、B、C、D、E、G 拜訪過不考慮。若 length[i]+a[i][j]小於原本 length 的值則替換。

	A	B	C	D	E	F	G	H
⑩+		∞	∞	∞	6		0	
	A	B	C	D	E	F	G	H
visited：	1	1	1	1	1	0	1	0
length：	3	3	0	1	4	12	10	13
pass_by：	D	C	C	C	B	D	E	C

找到最小的為 F，所以 A、B、C、D、E、F、G 拜訪過不考慮。若 length[i]+a[i][j]小於原本 length 的值則替換。

⑫+

⑫+		~~∞~~	~~∞~~	~~11~~		~~0~~		8

	A	B	C	D	E	F	G	H
visited：	1	1	1	1	1	1	1	0
length：	3	3	0	1	4	12	10	13
pass_by：	D	C	C	C	B	D	E	C

找到最小的為 H，因為所有節點皆被拜訪過，因此結束。

	A	B	C	D	E	F	G	H
visited：	1	1	1	1	1	1	1	1
length：	3	3	0	1	4	12	10	13
pass_by：	D	C	C	C	B	D	E	C

以 C 點為出發點：

A：C→D→A

B：C→B

C：

D：C→D

E：C→B→E

F：C→D→F

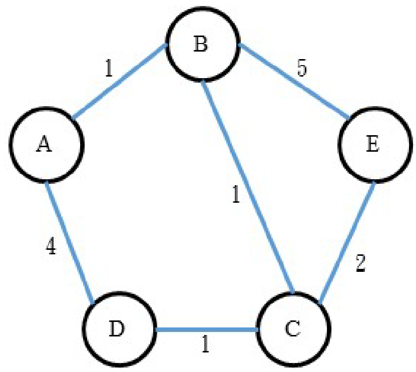
G：C→B→E→G

H：C→H

- **Floyd-Warshall's algorithm （Multi-terminal shortest paths problem）**

多端點最短路徑問題。

```
for(j=1 ; j<=N ; j++)//控制用來相加做比較的行
{
    for(i=1 ; i<=N ; i++)//由上向下，決定
    {
        for(k=1 ; k<=N ; k++)//由左到右，控制    的第k個元素
        {
            if((i!=j) && (j!=K))
            {//三項操作i到k如果經由j會不會比較好
                tmp = a[i][j] + a[j][k];
                if(a[i][k] > tmp)
                {
                    a[i][k] = tmp;
                    p[i][k] = p[i][j];
                }
            }
        }
    }
}
```



	A	B	C	D	E
A	0A	1B	∞C	∞D	4E
B	1A	0B	5C	1D	∞E
C	∞A	5B	0C	2D	∞E
D	∞A	1B	2C	0D	1E
E	4A	∞B	∞C	1D	0E

第一步：填色為被更動的格子。

j=1（藍色圈圈）；i=2（紅色圈圈）；k=1~n（綠色框框）

j=1（藍色圈圈）；i=3（紅色圈圈）；k=1~n（綠色框框）

j=1（藍色圈圈）；i=4（紅色圈圈）；k=1~n（綠色框框）

j=1（藍色圈圈）；i=5（紅色圈圈）；k=1~n（綠色框框）

＋ 和 做比較，選小的！

	A	B	C	D	E
A	0A	1B	∞C	∞D	4E
B	1A	0B	5C	1D	
C	∞A	5B	0C	2D	∞E
D	∞A	1B	2C	0D	1E
E	4A		∞C	1D	0E

第二步：j=2

	A	B	C	D	E
A	0A	1B			4E
B	1A	0B	5C	1D	5A
C		5B	0C	2D	
D		1B	2C	0D	1E
E	4A	5A		1D	0E

第三步：j=3

	A	B	C	D	E
A	0A	1B	6B	2B	4E
B	1A	0B	5C	1D	5A
C	6B	5B	0C	2D	10B
D	2B	1B	2C	0D	1E
E	4A	5A	10B	1D	0E

第四步：j=4

	A	B	C	D	E
A	0A	1B		2B	
B	1A	0B		1D	
C			0C	2D	
D	2B	1B	2C	0D	1E
E				1D	0E

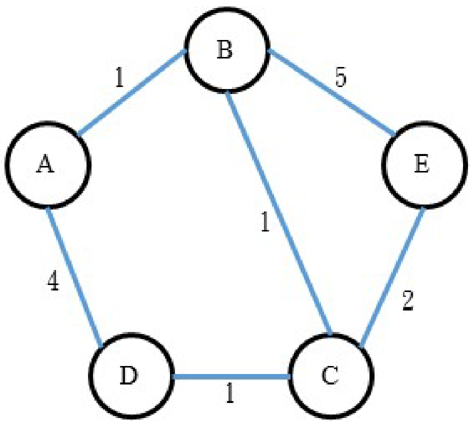
第五步：j=5

	A	B	C	D	E
A	0A	1B	4B	2B	3B
B	1A	0B	3D	1D	2D
C	4D	3D	0C	2D	3D
D	2B	1B	2C	0D	1E
E	3D	2D	3D	1D	0E

結果：

	A	B	C	D	E
A	0A	1B	4B	2B	3B
B	1A	0B	3D	1D	2D
C	4D	3D	0C	2D	3D
D	2B	1B	2C	0D	1E
E	3D	2D	3D	1D	0E

在原本的演算法的最後一行，p[i][k]=j 會有問題，例如從 C 至 A 就無法往回推算，僅可以從 A 推回 C。因此演算法將改成 p[i][k]=p[i][j]，代表某一點至另一點的下一站。例如 C 要到 A，則下一站為 D，接著 D 要到 A 則下一站為 B，最後 B 即可直接到達 A，路徑為 C→D→B→A。



➢ Binary Searching Tree

與一般二元樹不同的是，二元搜尋樹為了搜尋的效率，因此特別要求：樹中任一節點的 key value 都大於左子樹中任一節點的 key 值，並小於右子樹中任一節點的 key 值。這樣的約定，使得來到某一個節點 i 上時，就一定會發生以下三者中的任一種情形：

$$\begin{cases} i.key == target.key \rightarrow \text{找到，結束搜尋。} \\ i.key > target.key \rightarrow \text{尚未找到；接下來只需要往左子樹繼續搜尋} \\ i.key < target.key \rightarrow \text{尚未找到；接下來只需要網柚子樹繼續搜尋} \end{cases}$$

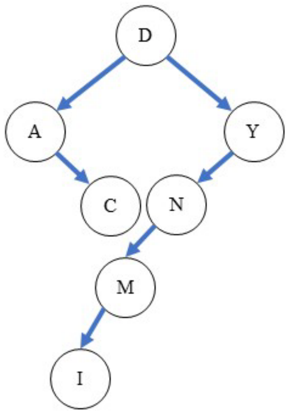
直到來到樹中某一個終端節點，這時如果仍然$i.key! = target.key$就表示 target 不存在於 tree 中，搜尋失敗。

在進一步思考，同樣內容的兩株二元搜尋樹，在搜尋時的效率會因為各節點排列順序的差異而有不同嗎？

在某些狀況下，我們可能知道，或可以預估樹中各節點發生的頻率或機率→直觀地想，為了提高搜尋的效率，發生頻率越高的節點自然是越靠近 root，深度越淺越快被找到越好。

D	Y	N	A	M	I	C
1	1	2	4	5	3	2

如果依照 input sequence 來建立二元搜尋樹，會得到：



weighted internal length of path

什麼是加權內部路徑長？

$$P_i = \sum_{i=1}^{N} freq(i) * depth(i)$$

（N 為 tree 中節點數）

$P_i = 1*1+4*2+1*2+2*3+2*3*5*4+3*5 = 58$

有更好的放法嗎？最好的嗎？

→一株具有最小加權內部路徑長的二元搜尋樹被稱為 Optimal Binary Searching Tree（最佳化二元搜尋樹）。

```
for(j=1 ; j<N ; j++)
{
    for(i=1 ; i<=N-j ; i++)
    {
        for(k=i ; k<i+j ; k++)
        {
            tmp = cost[i][k-1]+cost[k+1][i+j];
            if(tmp < cost[i][i+j])
            {
                cost[i][i+j] = tmp;
                best[i][i+j] = k;
            }
        }/*end of for k*/
        tmpfq = 0;//只是在計算i到j所構成的二元樹的內部路徑長
        for(k=i ; k<i+j ; k++)
        {
            tmpfq = tmpfq + freq[k];
        }
        cost[i][i+j] = cost[i][i+j] + tmpfq;
    }/*end of for i*/
}/*end of for j*/
```

將樹提高一層

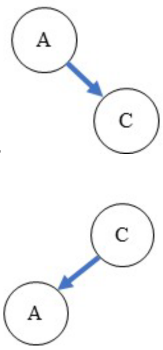
深度增加一層

初始化：

	A	C	D	I	M	N	Y
freq	4	2	1	3	5	2	1

	A	C	D	I	M	N	Y
A	4A						
C		2C					
D			1D				
I				3I			
M					5M		
N						2N	
Y							1Y

第一步：A 0+2



C 4+0

	A	C	D	I	M	N	Y
A	4A	8A					
C		2C	4C				
D			1D	5I			
I				3I	11M		
M					5M	9M	
N						2N	4N
Y							1Y

第二步：A　0+4 最佳→4（以 A 為 root 右子樹的值）+（4+2+1）(全部加一層）=11

C　4+1

D　8+0

	A	C	D	I	M	N	Y
A	4A	8A	11A				
C		2C	4C	10I			
D			1D	5I	14M		
I				3I	11M	15M	
M					5M	9M	12M
N						2N	4N
Y							1Y

第三步：C　0+14

D　2+11

I　4+5→9（以 I 為 root；左子樹 CD；右子樹 M 的值）+（2+1+3+5）(加深一層）

M　10+0

	A	C	D	I	M	N	Y
A	4A	8A	11A	19C			
C		2C	4C	10I	20I		
D			1D	5I	14M	18M	
I				3I	11M	15M	18M
M					5M	9M	12M
N						2N	4N
Y							1Y

第四步：C　0+18

D　2+15

I　4+9

M　10+2→12（M 為 root；左子樹 CDI；右子樹:N）+2+1+3+5+2（加深一層）

N　20+0

	A	C	D	I	M	N	Y
A	4A	8A	11A	19C	31I		
C		2C	4C	10I	20I	25M	
D			1D	5I	14M	18M	21M
I				3I	11M	15M	18M
M					5M	9M	12M
N						2N	4N
Y							1Y

第四步：

	A	C	D	I	M	N	Y
A	4A	8A	11A	19C	31I	37I	
C		2C	4C	10I	20I	25M	28M
D			1D	5I	14M	18M	21M
I				3I	11M	15M	18M
M					5M	9M	12M
N						2N	4N
Y							1Y

第五步：A　0+28

C　4+21

D　8+18

I　11+12→23（I 為 root；左子樹 ACD；右子樹 MNY）+4+2+1+3+5+2+1=41

M　19+4→23（M 為 root；左子樹 ACDI；右子樹 NY）+4+2+1+3+5+2+1=41

N　31+1

Y　37+0

	A	C	D	I	M	N	Y
A	4A	8A	11A	19C	31I	37I	41M 41I
C		2C	4C	10I	20I	25M	28M
D			1D	5I	14M	18M	21M
I				3I	11M	15M	18M
M					5M	9M	12M
N						2N	4N
Y							1Y

考試規定：等號都放左邊！

結果（等號放左邊，以 I 為 root）：