第一章 基本觀念

一、 演算法的基本定義

Input 輸入	由外界輸入「零」個以上的資料(沒有外界輸入，自己產生的也可以)
Output 輸出	至少有一個以上的輸出
Effectiveness 有效性	凡是電腦能算的，人都能算
Definiteness 明確性	每個指令要有明確的定義，像除以零的運算，連數學上的定義都沒有，就是不符合明確性
Finiteness 有限性	演算法在執行過一定的步驟後會終止，這點跟「procedure」不同，像 OS 就不能停下來

二、 Data Structure

Data types	例如 int 是一種資料型態，它是 16bit 的 2 補數
Data object	因此 int 包含了-32768~32767
Data structure	

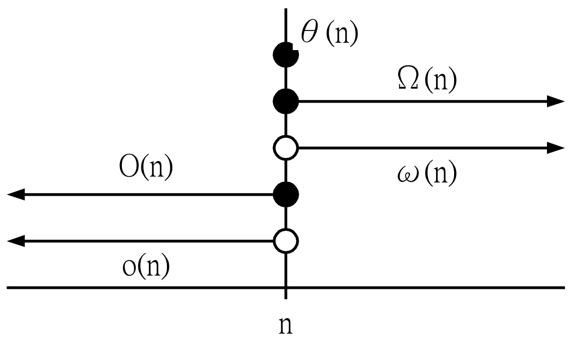
三、 何謂 ADT(Abstract data type)

ADT 是一種資料型態，ADT 的表示方式使得物件的規格與物件上的運算與該物件的內部表示法無關。

例如我們定義：Stack 是一種在同一端進行 insert、delete 操作的資料型態，但並沒有說明 Stack 如何用 Array 或 linked list 實作。

四、 Asymptotic notations(漸近式表示法)

O	f(n)=O(g(n))，若且唯若存在著正數常數 c_1、n_0 使得當 $n>=n_0$ 時，$0<=f(n)<=c_1xg(n)$
o(untightly upper bound)	f(n)=o(g(n))，若且唯若存在著任一常數 $c_1>0$、n_0 使得當 $n>=n_0$ 時，$0<=f(n)<c_1xg(n)$ (注意，沒有等號)
Ω	f(n)=Ω(g(n))，若且唯若存在著正數常數 c_1、n_0 使得當 $n>=n_0$ 時，$0<=c_1xg(n)<=f(n)$
ω (untightly lower bound)	f(n)=ω(g(n))，若且唯若存在著任一常數 $c_1>0$、n_0 使得當 $n>=n_0$ 時，$0<=c_1xg(n)<f(n)$ (注意，沒有等號)
θ	f(n)=θ(g(n))，若且唯若存在著正數常數 c_1、c_2、n_0 使得當 $n>=n_0$ 時，$0<=c_1xg(n)<=f(n)<=c_2xg(n)$



▲$\Omega(n)+\theta(n^2)=[n,\infty)+[n^2,n^2]$對應項取大的→$[n^2,\infty)=\Omega(n^2)$

▲$\Omega(n)+O(n^2)=[n,\infty)+[1,n^2]$對應項取大的→$[n,\infty)=\Omega(n)$

▲$\theta(n)+o(n^2)=[n,n]+[1,n^2]$對應項取大的→$[n,n^2]=\Omega(n)$或 $o(n^2)$

五、 常用級數公式

$$\sum_{i=1}^{n} i=\frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2=\frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i^3=\left(\frac{n(n+1)}{2}\right)^2$$

$$\sum_{i=1}^{n} i^k=\frac{n^{k+1}}{|k+1|},k\neq -1$$

$$\sum_{i=1}^{n}\frac{1}{i}=\theta(\log n)$$

$$\sum_{i=1}^{n}\log i=\theta(n\log n)$$

六、 常見的等級比較：

▲$1<\log n<n<\log(n!)<n^{常數}<常數^{n}<n!<n^n$

▲$n^{\varepsilon}>\ln(n)$，其中 $0<\varepsilon<1$

七、 Matrix Product Chain

如果使用窮舉法，則 n 個矩陣會有 $\frac{1}{n+1}\times C_n^{2n}$ 種相乘方式，以下是使用 dynamic programming 的方式，時間複雜度為 $O(n^3)$

A[i,j]=0, i=j

$A[i,j]=\min_{i\le k<j}\{A[i,k]+A[k+1,j]+d[i-1]\times d[j]\times d[k]\}$ 其中 i<j

A[i,j]的意思是矩陣的第 i 個乘到第 j 個，最少的相乘次數

以 A_{4*2}、B_{2*3}、C_{3*5}，則 $d_0=4,d_1=2,d_1=2,d_2=3,d_2=3,d_3=5$

第二章 陣列

算算 column major、row major、元素位置(盡量畫圖)，帶狀矩陣(Band Matrix)比較麻煩一點

值得注意的是，如果是多維矩陣，
A[L1…U1][L2…U2][L3…U3]…[Ln…Un]…算位置的方法->

令 Wi = Ui – L i + 1

列優先(row-major ordering)

Loc(A[I1,I2,I3….,In]) = a+c[(I1 – L1)W2W3W4…Wn
+ (I2 – L2)W3W4…Wn
+(I3 – L3)W4W5…Wn+…+(In – Ln)]

行優先（column-major ordering ）

Loc(A[I1,I2,I3….,In]) = a + c[(I1 – L1)+
(I2 – L2)W1+
(I3 – L3)W1W2+….
(In – Ln)W1W2W3…Wn-1]

用在二維也可以這樣算…這樣記 ->

列優先從右邊開始乘 Wi…往左邊減…越乘越少
行優先從左邊開始乘 Wi …我右邊加…用乘闊多

第三章 堆疊與佇列(Stacks and Queues)

	Stack	Queue	Tree	Graph
定義	一種有限的 ordered list，其插入與刪除運算皆在同一端進行，具有後進先出的特性	一種有限的 ordered list，其插入運算在一端進行，而刪除運算則在另一端進行，具有先進先出的特性		
應用	1.圖形的 DFS 2.副程式呼叫 3.運算式中序式轉後序式 4.回溯法	1.圖形的 BFS 2.OS 的 scheduling 3.multiple buffer	1.搜尋 2.索引 3.huffman code	1.最短路徑的計算 2.網路的 routing 3.topological sort

一、 Mazing problem 的處理原則(p66)

八個方向：

二、 Stack 的應用問題：

P62 車廂問題 （畫個圖）

車廂放進去 · 做 Push 和 Pop 的動作，如果車廂沒有了，還要 pop 就是不合法的動作，push 用 E 表示 pop 用 X 表示，那麼 EEEXXX 就是合法串列

組共有 種合法串列

合法串列問題總共跟四種問題類似

車廂還有順序問題：

123456 車廂進去，有一些順序無法 pop…大 小 中
像是 154236 中 423 就不行…

三、 四大類型的 Queue

以 C 語言來討論，陣列由 q[0~n-1]

Linear Queue	Front = -1,rear = -1
	Empty : front=rear
	Full : front = -1 and rear = n-1
	AddQ : rear++；q[rear]= item
	DelQ : front++；return q[front]

Circular Queue (未使用特殊方法時只能用 n-1 個)	Front= n-1 ,rear= n-1
	Empty : front = rear
	Full : (rear+1) mod n = front
	AddQ : q[rear]=item；rear=(rear+1) mod n
	DelQ : return q[front]；front=(front+1) mod n
Linear Deque	Right = n/2,Left=Right+1
	Empty : Left>Right
	Full:left = 0 and right=n-1
	左邊 AddQ:left---；q[left]=item 左邊 DelQ:return q[left]；left++
	右邊 AddQ:right++；q[right]=item 右邊 DelQ:return q[right]；right—
	Linear 有一個問題，就是可能左邊頂到 或是右邊頂到 這時候就要整個 Move
Circular Deque	Left=1 right=0
	Empty : (right+1) mod n = left
	Full:(right+2)mod n=left 留一個位置來分辨，不然如果又重疊…又分不清楚 Emptyc 還是 Full 了
	左邊 AddQ : left = (left+n-1) mod n；q[left]=item 左邊 DelQ : return q[left]；left = (left+1) mod n 幹麻 left + n – 1？ 研判是怕從正變成負的 另一邊從負的變成正的 沒差…
	右邊 AddQ:right=(right+1) mod n；q[right]=item 右邊 DelQ:return q[right]；right = (right+n-1) mod n

DeQue -> 兩端皆可 刪除 或 插入 的 Queue
DeQue 實做的技巧->函數傳的時候用傳一個 Left_End 判斷 True or False 這樣用一個函式就可以解決了

因為 Circular 都只能存 n – 1 筆資料，雖然 circular queue 的目的在於可以循環放，而不是放很多，不過還是有三個解決方法：

1. Tagger Circular Queue

用一個 tag 來分辨是 empty 還是 full ->

empty : QFull False and front = rear
Full: QFull True and front = rear

2. Circular queue with Special Value

一開始設 front = 0 rear = n (其實只到 n - 1
add 時 因為會做 (n+1) % n 所以其實也是 在 1 的位置

但是 rear = n 那是初值設定…那怎麼半，就是 Del 的時候判斷 ，如果 rear = front (空了) 那趕快去把 fonrt = 0 rear = n

3. 設一個 Length empty : Length = 0 full:Length = n

Front 可以是 0 ~ n 任何值 queue 的大小可以到 n
Add 的時候用 (Front + Length) 連 rear 都省了@@

但如果考試考…可以放幾個 還是達 n – 1 (那我打那麼多幹麻？)

四、 Multiple Stacks and Queues

▲two stacks

	→							←	

(有三種　相對的　同向的　背向的，後兩者碰到底都要在整個移動所以還是相對的好)

▲multi stacks、multi queues

	→			→			→		

用 b[i]　和　t[i] 紀錄每個的開頭跟尾巴 b[i] 在整個陣列前面的一個　t[i]在陣列最後面

所以有陣列滿的話就是 t[i] = b[i + 1]

滿了去移動 -> 先往右找 stack j　~　stack j +1　中空的位置
然後般 stack i+1 到 stack j 往右邊一格

再往左找　找到 stack j 和 stack j + 1　，在搬 stack j +1　到 stack I 往左搬一格　（自己也要搬的意思）

五、 運算式的處理

運算式轉換的優先等級設定表

Operator 或 operand	ISP	ICP
operand	6	6
(	0	6
)	6	0
^(指數較特別，它是右結合性)	4	5
*/	4	3
+-	2	1
#	-1	-1

```
void infix_to_postfix()
{
  push('#');
  infix=infix + '#';
  while (stack is not empty)
  {
        u=getnext(infix);
        if (x is an operand)
              output(x);
```

虛線內這段程式碼其實可以不加，因為上表優先順序編號規則已經包含左右括號的處理，不過有的老師「習慣看到」括號的處理，所以還是加上比較保險

```
        else if (x = '(')
              push(x);
        else if (x = ')')
        {
              p=pop();
              while(p!='(')
              {
                    output(pop());
                    p=pop();
              }
              discard(x);
              discard(p);
        }
```

```
        else
        {
              while(ISP()>ICP(x))
                    output(pop());
              if (ISP()<ICP(x))
                    push(x);
              else
              {
                    discard(pop());
                    discard(x);
              }
        }

  }
}
```

上面這個簡單來說就是把運算子都 push 進去看到運算元就直接 pop…如果看到運算子優先順序 stack 裡面比較高的，就 pop 出來，把 input 這一個給 push 進去。

▲please design an algorithm to evaluate an infix expression without converting it to postfix one.

六、 運算式的檢查

1. 後序式的檢查
   由左而右檢查，初值為 0，遇到 operand 就+1，遇到 operator 就-1，過程中不能出現 0，最後檢查完一定是 1
2. 前序式的檢查
   同上，只是改成由右往左
3. 中序式的檢查
   i. 檢查運算子的位置是否都正確
      從 0 開始，由左而右，遇到運算元+1，運算子-1，如果遇到括號則值不變，因此除了括號以外，資料會 01 相間，而且左括號的值都應該是 0，右括號都應該是 1

A	+	B	*	(	C	-	(	D	/	E	)	)
1	0	1	0	0	1	0	0	1	0	1	1	1

上式為正確資料

A	+	B	*	(	C	-	(	D	/	E	)
1	0	1	0	0	1	0	0	1	0	1	1

上式的資料符合規則 1，但實際上少了個括號，所以要再做第二個檢查

   ii. 檢查括號是否對稱
      由左往右，遇到括號就放入堆疊中，要放入堆疊中的括號如果與目前堆疊頂端的括號對稱，則一起 pop 掉(因為這一組確定合法)，最後如果 stack 不為空，則表示括號沒有成對，運算式不正確

七、 前序式轉二元樹

前序式：-*^+ABCD+E*FG
中序式：(A+B)^C*D-E+F*G

```
char c[14]={"-*^+ABCD+E*FG"};
//其實只有 13 個字元，但陣列要多宣告一個長度放'/0'結尾符號
void main()
{
  node *n=(node *)malloc(sizeof(node));
  n->c=c[0];
  node *p;
  p=build(n);
}

node * build(node *p)
{
  if (p==NULL) return NULL;
  else
  {
        node *n=(node *)malloc(sizeof(node));
        n->c=Get_Next();
        if (IsOperand(n->c))
        {
              n->left=NULL;
              n->right=NULL;
        }
        else
        {
              n->left=build(n);
              n->right=build(n);
        }
        return n;
  }
}
```

八、前　中　後　之間的轉換：

前後轉中 -> ok…用 stack　看到運算子就 pop 兩個出來用再 push 回去
中轉前後 -> 這個要參考前面的符號順序表
前後互轉 -> 這個要用畫圖的　框起來那樣轉

九、一個題型　infix 要直接算 ->
轉成後序在 output 的時候直接當成 postfix 的輸入

第四章 遞迴

▲所有的 for loop 都能用遞迴改寫

一、 著名的遞迴程式

▲注意一下它們的「邊界條件」

1. Factorial
   n!=1，當 n<=1
   n!=n×(n-1)!，當 n>1
2. 最大公因數 GCD
   gcd(a,b)=a，當 b=0
   gcd(a,b)=gcd(b,a mod b)，當 b>0
3. Fibonacci Numbers
   Fn=n，當 n<=1
   Fn=Fn-1+Fn-2，當 n>1
   時間複雜度：

Recursive	$\theta(\varphi^n)$
Iterative	θ(n)
公式法 $F_{2n}=F_n(F_{n+1}+F_{n-1})$	O($\log_2$n)
公式法 $F_n=\frac{1}{\sqrt{5}}(\phi^n-\hat{\phi}^n)$	O($\log_2$n)

   最底下的公式，因為 φ^n 即使使用 Horner's Rule，也要 $\log_2$n 的相乘次數
4. Combination(組合公式)
   $$C_k^n=1當n=k或k=0$$
   $$C_k^n=C_k^{n-1}+C_{k-1}^{n-1}otherwise$$
   遞迴版：O(2^n)
   Dynamic programming：O(nk)
   遞迴關係式：C[i][j]=c[i+j-1][j]+c[i+j-1][j-1]
   程式：
```
for (i=1;i<=k;i++)
  for (j=1;j<=n-k;j++)
        C[i+j][j]=C[i+j-1][j-1]+ C[i+j-1][j];
```
5. Ackermann's Function
   A(m,n)=n+1，當 m=0
   A(m,n)=A(m-1,n)，當 m<>0 and n=0
   A(m,n)=A(m-1,A(m,n-1))，otherwise
   它也有 dynamic programming 的解法
6. Tower of Hanoi
   搬動次數：
   T(n)=1，當 n=1
   T(n)=2T(n-1)+1，當 n>1
```
Void move(int n, char source, char temp, char dest)
{
  if (n= =1)
        move disk 1 from source to dest;
  else
  {
        move(n-1,source,dest,temp);
        move disk n from source to dest;
        move(n-1,temp,source,dest);
  }
}
```
   時間複雜度：O(2^n)
7. Permutation(排列組合)
   時間複雜度：
   T(n)=c，當 n=1
   T(n)=n×T(n-1)+cn，當 n>1→T(n)=O(n!)
   //陣列由 1 到 n
```
void perm(int a[],int k,int n)
{
  if (k==n) output();
  else
  {
        for (i=k;i<=n;i++)
        {
              swap(a[i],a[k]);
              perm(a,k+1,n);
              swap(a[i],a[k]);//換回來
        }
  }
}
```
8. 集合 S 的所有子集合
   時間複雜度：
   T(n)=c，當 n=0
   T(n)=2T(n-1)+c，當 n>0→T(n)=O(n×2^n)
```
void subset(char s[],int k,int i,int n)
//參數 k 是記錄已有幾個元素被選入 subset 中，參數 i 則是記錄目前在考慮第幾個元素是否要加入
{
  if (i>n) output();
  else
  {
        subset(s,k,i+1,n);
        s[k+1]=atom[i];
        subset(s,k+1,i+1,n);
  }
}
```
   △subset(s,k,i+1,n);//第 i 個元素不加入，因此子集合中還是只有 k 個元素，並往下考慮第 i+1 個元素
   △s[k+1]=atom[i];//第 i 個元素加入子集合中為第 k+1 個元素
   △subset(s,k+1,i+1,n);//第 i 個元素已加入，並往下考慮第 i+1 個元素
   △此演算法的時間複雜度為 O(2^n)
9. selection of kth-small element

▲求解時間複雜度的方法

1. Master theorem
2. 暴力法(brute force)
3. 遞迴關係式

二、 遞迴關係式

▲如果特徵方程式的根為 1,2,3

則通解為 $a_k=c_1 3^k+c_2 2^k+c_3 1^k$

▲如果特徵方程式的根為 2(三重根),3(二重根)

則通解為

$$a_k=(c_1k^2+c_2k^1+c_3)\times 2^k+(c_4k^1+c_5)\times 3^k$$

▲型一：齊次方程式
   ex：F(n)=F(n-1)+F(n-2)

▲型二：非齊次方程式
   ex：T(n)=T(n-1)+T(n-2)+1

▲型三：Domain Transform
   ex：T(n)=2T(n/2)+n-1
   令 n=2^k 去解

▲型四：Range Transform
   ex：$a_n=3a_{n-1}^2$
   左右取 $\log_2$→
   $$\log_2(a_n)=\log_2(3a_{n-1}^2)=\log_2 3+2\log_2 a_{n-1}$$
   $$令b_n=\log_2(a_n)$$

三、 Master Theorem

$$T(n)=a\times T(\frac{n}{b})+g(n)$$

$$g(n)=O(n^{\log_b a-\varepsilon})\Rightarrow T(n)=\theta(n^{\log_b a})$$

$$g(n)=\theta(n^{\log_b a}\times\log^k n)\Rightarrow T(n)=\theta(n^{\log_b a}\times\log^{k+1} n)$$

$$g(n)=\Omega(n^{\log_b a+\varepsilon})\Rightarrow T(n)=\theta(g(n))$$

如果是這種型式 $T(n)=T(pn)+T(qn)+cn$，則

$$若p+q<1\Rightarrow T(n)=\frac{cn}{1-(p+q)}$$

$$若p+q=1\Rightarrow T(n)=O(n\log n)$$

第五章 Linked Lists

▲定義：Linked lists 是一個 ordered list，其各項資料可儲存於記憶體中分散之位置。Linked lists 可用 array 或 dynamic allocation 實作

▲Linked List 分為 Single Linked List、Bi-directional/Double Linked List

▲Linked Stacks and Queues
Linked List 的結構選擇

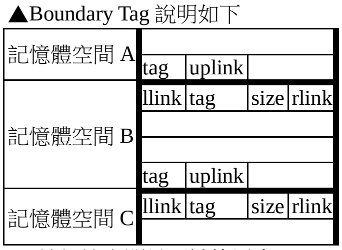
單鏈 雙鏈	非循環 循環	使用首節點 使用尾指標

Linked Stack	單鏈+top 指標實作 Linked Stack，而且是以 list 的前端作為 stack 的 top，因為使用最簡單的結構時在 linked list 的前端加入、刪除資料最容易
Linked Queue	使用單鏈+front、rear 指標實作
Linked Deque	使用雙鏈+left、right 指標實作

第六章 Linked Lists 的應用

一、 等價關係(Equivalence Relations)
等價關係具有 reflexive、symmetric、transitive 三種特性

二、 Dynamic Memory Management
用 linked list 來連接可用空間，但隨著記憶體空間越來越零碎，要將可用空間進行合併的運算需線性搜尋串列較為耗時，因此採用 Boundary Tag(邊界標籤)的方式來合併可用空間
▲Boundary Tag 說明如下

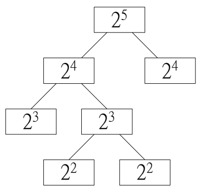


tag 註記該空間是否被使用中
llink 指向上一個空間的尾端
rlink 指向下一個空間的開端
size 說明配置空間的大小
uplink 指向本身區塊的開頭
△當空間 B 被釋放時，只要讀第一列的資料，就可以決定是否要跟上下區塊合併，例如由 llink 讀取空間 A 的 tag，得知空間 A 也是 free 的，由 uplink 找到空間 A 的開頭進行合併，由 rlink 找到空間 C 的開頭，判斷其 tag 欄位，若是 free，則可進行合併

三、 Buddy System(夥伴系統)
△也是一種動態記憶體的管理辦法，其配置空間皆為 2^k，其中 0<=k<=m，而記憶體總空間為 2^m
△每塊空間的結構為

llink	tag	kval	Rlink

示意圖如右
一旦空間被釋放，則只有原自同一個父親的節點才能合併，兄弟不能合併



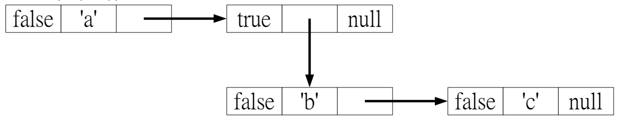
四、 廣義串列(Generalized List)
△節點結構

tag	data 或 dlink	link

```
Struct ListNode
{
        bool tag;
        union
        {
                int data;
                ListNode *dlink;
        } u;
        ListNode *dlink;
}
```

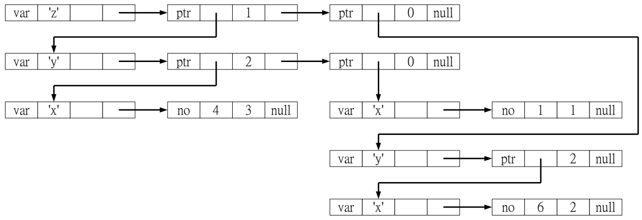
tag=false，表示此節點用來記錄 data
tag=true，表示此節點用來記錄指標

▲A=(a,(b,c))的廣義串列表示法



▲$4x^3y^2z+6x^2y^2+xz=(4x^3y^2+x)z^1+6x^2y^2z^0=((4x^3)y^2+x^1y^0)z^1+6x^2y^2z^0$
節點結構

a	b	exp	link
Var：表示欄位 b 記錄變數 Ptr：表示欄位 b 記錄指標 No：表示欄位 b 記錄係數		記錄指數	記錄 link



五、 廣義串列多項式表示法
1. 用單一陣列表示 $2x^4-x^3+5x+3$

0	1	2	3	4	5
3	5	0	-1	2	0

2. 用二個陣列表示 $2x^4-x^3+5x+3$

coef	2	-1	5	3
exp	4	3	1	0

3. **Horner's Rule**
減少多項式乘法的次數

$$p(x)=a_nx^n+a_{n-1}x^{n-1}+..........+a_1x^1+a_0$$
$$=(..............((a_n)x+a_{n-1})x...............+a_1)x+a_0$$

```
float peval(float coef[],int n,float x)
{
        float p;int i;
        p=coef[n];
        for (i=n-1;i>=0;i--)
                p=p*x+coef[i];
        return p;
}
```

4. sparse matrices(稀疏矩陣)
5. 矩陣的鏈結串列表示法
見手寫筆記

六、 字型比對(Pattern Matching)
1. Rabin-Karp algorithm
▲一般的字串比對方式，是用 pattern 每次移動一個位置跟 source 比對，因為假設 pattern 長度 m，source 長度 n，則比對次數約 O(mn)；(每次最多比 m 次，一共要比 n-m 次)
▲Rabin-Karp 的方法可以在 O(m+n)時間內比完
方法如下：
假設 source 字串為 ababacabacabad................
pattern 為{baadc}
則以 pattern 中的字元自行定義編碼，例如：

a	b	c	d
0	1	2	3

△(編碼要包含所有會出現的字元嗎?還是以 patter 裡的來編即可，pattern 以外的字元用特殊的編碼??不知，沒講)
△將 pattern{baadc}換算成此編碼{baadc}={10032}$_4$四進位
→pattern=270
△將 source 字串也依此編碼進行計算，除了第一次要全部重算之外，之後每移動一個位置，只要加算新進來的字元即可，不必全部重算，原理如下：

$$(S_iS_{i+1}...S_{i+m-1})_d=S_i*d^{m-1}+S_{i+1}*d^{m-2}+...+S_{i+m-2}*d+S_{i+m-1}$$
往右 shift 一位

$$(S_{i+1}S_{i+2}...S_{i+m})_d = S_{i+1}*d^{m-1} + S_{i+2}*d^{m-2} + ... + S_{i+m-1}*d + S_{i+m}$$
$$= ((S_i S_{i+1}...S_{i+m-1})_d - S_i*d^{m-1})*d + S_{i+m}$$

例如：{ababa}=$(01010)_4$=68
68<>270，往右移一位→{babac}
可用(68-0*4^4)*4+2=274(此即為往右移一位後，新的編碼，不用重算)

2. KMP(Knuth-Morris-Pratt algorithm)
使用 prefix function 或稱 failure function 來加速 pattern 的移動
failure function 的定義：

$p = p_0 p_1 ....p_{n-1}$ is a pattern

f(j)=最大的 k 值，當 k<j 且 k>=0 使得

$p_0 p_1 ... p_k = p_{j-k} p_{j-k+1} ... p_j$

f(j)=-1，在其它的情形下

-1	-1	0	0	1	2	3	4	-1
a	b	a	a	b	a	a	b	b

此演算法的時間複雜度為 O(m+n)

3. Boyer-Moore Algorithm
採用類似 KMP 的方法加速 pattern 的移動,當字串中出現一個完全未在 pattern 中出現的字元，可以進行所謂的 bad character shift，一次將 pattern 移動較長的距離，因此此演算法的 worst case 為 O(mn)，best case 為 O(n/m)，average case 為 O(m+n)

計算 failure function 程式碼：

```
void fail()
{
	p[0]=-1;
	for (i=1;i<=n-1;i++)
	{
		j=p[i-1];
		while (a[i]!=a[j+1] && j>=0)
			j=p[j];
		if (a[i]==a[j+1])
			p[i]=j+1;
		else
			p[i]=-1;
	}
}
```

字串比對程式碼

```
int pmatch(char *s, char *p)
{
	int i=0,j=0;
	while (i<n && j<m)
	{
		if (s[i]==p[j])
		{
			i++;
			j++;
		}
		else
		{
			if (j==0) i++;
			else j=p[j-1]+1;
		}
	}
	return (j==m?i-m:-1);
}
```

第七章 Tree

一、 名詞

1. 節點 Degree，即節點的分支度 = branch
2. 樹的 Degree，整棵的所有節點中的最大分支度
3. leaf node(terminal node)，葉節點，分支度為 0 的節點
4. internal node(分支度不為 0 的節點)，??這樣加上 failure node 的 leaf node 算不算 internal node 呢?
5. external node，又稱為 failure node，協助運算用
6. 其它：subtree、children、parent、sibling(兄弟)、ancestor、descendant、level、height(=depth)
7. 樹：至少有一節點(跟二元樹的區隔，二元樹可為空)
8. forest(森林)：可為空集合，由 n>=0 棵樹組成

▲兩個在 Graph 出現的名詞

9. free tree(自由樹)：係指一個 connected、acyclic 的無向圖形
10. sink tree：算好 single source all destination 有向圖中的所有最短路徑後，把從終點往回走到起點的路徑方向都反轉所形成的路徑

二、 樹的表示法(p3)

11. 范氏圖
12. 廣義串列
13. 階層表示法
14. 連結串列表示法
15. Leftmost-Child-Right-Next-Sibling

三、 二元樹

★注意 p5 的 Binary Tree、Ordered Tree、Unordered Tree 的比較

二元樹與樹的三個主要不同特點：

1. 樹不可為空集合，至少存在一樹根，二元樹可以是空集合
2. 樹的每一節點分枝度(degree)>=0，二元樹每一節點的分支度 d 存在著 0<= d <=2
3. 樹的子樹之間沒有次序關係，而二元樹的子樹之間有次序關係

Binary Tree 的特性：

Level i 的節點數 = $2^{i-1}, i\ge 1$

若樹的高度 = h，則此 full binary tree 的總節點數 = $2^{h}-1$

如果是 complete binary tree，則總節點數 = 2^{h-1} <= n <= 2^{h}-1

若一二元樹有 n 個節點，B 代表樹的總分支數，n_i 代表分支度為 I 的節點數，則會有下列關係：

$B=n-1$

$n=n_0+n_1+n_2$

$n_0=n_2+1$

n 個節點所能排成不同二元樹個數為 $\frac{1}{n+1}\times C_n^{2n}=C_n^{2n}-C_{n-1}^{2n}$

▲遞迴關係式：

Bn=1，n=1

Bn = B_0B_{n-1}+ B_1B_{n-2}+ …… + $B_{n-2}B_1$+ $B_{n-1}B_0$，n>0

▲幾個要弄清楚的名詞

complete binary tree (1~k-1 層全滿，第 k 層只缺右側的 node，都不缺也可以→full binary tree 一定是 complete binary tree，complete binary tree 有可能是 full binary tree)

full binary tree (深度為 k 具有 2^k-1 個節點)

strictly binary tree (除樹葉外每個節點同時具有左右子樹)

extended binary tree (加入 external node 的二元樹稱之)

▲幾棵二元樹的計算

A、B、C 三個 node 可以建立幾棵二元樹：3!*(1/4*C(6,3))

A、B、C 三個 node 可以建立幾棵二元「搜尋」樹：1/4*C(6,3)

後者不用乘以 3!，是因為它是 BST，ABC 的順序會被固定，而前者 A、B、C 的次序可以任意調動

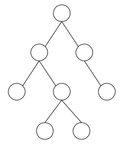
四、 二元樹的陣列表示法(p9)

注意陣列的起始是 0 或 1，會影響到左兒子是 a[2i+1]或 a[2i]

推導如果是 k-ary tree，則用陣列表示時如何推算第 x 個兒子，或由兒子推算父親

五、 二元樹追蹤(binary tree traversal)

遞迴的寫法好寫，非遞迴的寫法比較難



在進行程式碼的追蹤時，可應用此圖形去追蹤各種狀況

16. recursive program
17. stack(p16、17 範例 30、31、32)

    前序、中序的非遞迴寫法都比較直覺，後序的要去判斷節點的狀態，例如第二次從 stack 中出來才 output。考試時若一時想不到直接後序追蹤的方式，可以運用前序追蹤後，再反轉輸出結果也算後序輸出。(會被扣點分數，但總比寫不出來好)

18. parent pointer(p18 範例 32)
19. threaded binary tree
20. link inversion tree：當往節點的 left subtree 搜尋時，將 left child 指標改成指向 parent；搜尋 right child 時，則將 right child 指標改成指向 parent

▲(p13)題目只給前序，後序；要求找出所有可能的二元樹

前序：N LLLL RRRR

後序：LLLL RRRR N

六、 樹林與二元樹的轉換(p23)

為何要將 tree 或 forest 轉成 binary tree?

主要是指標浪費率的關係

n nodes，order k 的 tree，整棵樹的總指標數=nk，有用的指標數=B

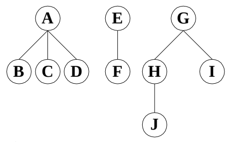
又 B=n-1→空指標個數=nk-(n-1)→指標浪費率

$$=\frac{nk-(n-1)}{nk}=\frac{n(k-1)+1}{nk}=\frac{k-1}{k}+\frac{1}{nk}\approx\frac{k-1}{k}$$

所以即使是二元樹也浪廢了 1/2=50%

所以有 threaded binary tree 的結構出現

▲樹的追蹤一般而言先轉二元樹再做追蹤，但也有直接追蹤的方法



(p23 有上圖直接對樹進行前、中、後序的追蹤方式)

七、 Threaded binary tree

引線有分前、中、後，考試若沒有指明，則通常是預設用中引線

(這裡相關的題型比較複雜，要多演練)(但這幾年較少考<~那可以不要唸嗎 = =?)

思考： (都用畫圖來思考各種狀況)

1.引線如何用來進行二元樹追蹤 (inorder -> 找中序後繼者)

(preorder 可以做…p28…而 postorder 就麻煩了 不過可以用 stack 反轉他 LRN -> NRL…就可用上面方法解)

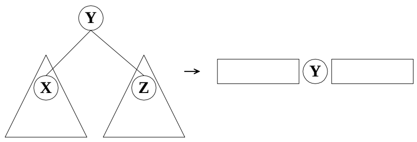
用引線作中序追蹤 -> 一直去找中序後繼者

用引線作前序追蹤->左邊沒兒子就一直往上到右有兒子

2.如何建立引線
3.刪除 threaded binary tree 中的節點時，相關的引線調整動作(p30 範例 55)

八、 Binary search tree
★二元搜尋樹的左子樹一定小於右子樹
★對 Binary search tree 進行中序追蹤，即可獲得由小到大的排序結果
★任意 BST，要如何重建使它有最小高度?
21. AVL Tree(太麻煩 <- 明明很好玩)
22. 用 in-order traversal 取出排序數列，再重建 BST(p34)
▲二元樹的資料搜尋平均比較次數 O(logn)的證明過程
▲二元樹的節點上加一個 size 欄位，記錄包含節點本身的子樹的總節點個數，這樣的資料結構有助於搜尋第 ith 小的元素。詳見 p38 範例 67

九、 算式樹(Expression trees)(p46)
對算式樹做前序追蹤可得前序式，做後序追蹤可得後序式
對算式樹做中序追蹤，則必需考慮下列狀況才能獲得正確的中序式

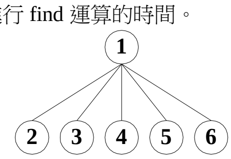


轉換成右式時，加不加括號與 operator 優先等級之間的關係 X > Y 不加，X < Y 要加；Y > Z 要加，Y < Z 不加

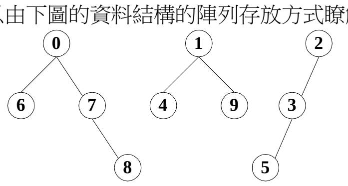
十、 Decision Trees
▲著名的問題，有 n 個 coin，其中一枚為假，則決策樹最低的高度是多少?
解題：n 個 coin → 最多有 2n 種結果 → 亦即失敗節點有 2n 個
如果題目是說「最多」有一枚為假，則可能結果有 2n+1 種
▲注意：決策樹每一節點的分支度最多為 3，所以是當成 ternary tree 在做節點個數的計算

十一、Disjoint sets(互斥集合)
Union 運算：O(1)
Find 運算：O(logn)(這是個證明題)
上述兩個運算在 minimum cost spanning tree 的 Kruskal's algorithm 有應用。

▲weighting rule：指互斥集合在進行 union 運算時，是將節點個數少的樹根指向節點個數多的樹根
▲collapsing rule(path compression)：在進行 Find 的過程中，每碰到一個元素就把它的 parent 指向根節點，這樣可以節省下次進行 find 運算的時間。



怪!?Find(6)=1 是很快，但是去找到 6 這個元素難道不用時間嗎?
Ans：不用，因為元素的位置都不動，只動它的 parent 指標，可以由下圖的資料結構的陣列存放方式瞭解



0	1	2	3	4	5	6	7	8	9
-4	-3	-3	2	1	3	0	0	7	1

負值表示此樹共有多少個節點
正值則是 parent 指標，用來指向父親
所以如果把子樹 1 Union 到子樹 0，則陣列如下

0	1	2	3	4	5	6	7	8	9
-7	0	-3	2	1	3	0	0	7	1

改兩個值就好，非常迅速

第八章 Heap

	Insert	Del-max	Del-min	Combine	Delete any one	Decrease-Key
Max heap	● O(logn)	● O(logn)				
Min heap	● O(logn)		● O(logn)			
Min-Max heap	● O(logn)	● O(logn)	● O(logn)			
Deap	● O(logn)	● O(logn)	● O(logn)			
Leftist Tree	● O(logn)		● O(logn)	● O(logn)		
Binomial heap	●O(1) *		● O(logn) *	●O(1) *		
Fibonacci heap	●O(1) *		● O(logn) *	●O(1) *	●O(1) *	●O(1) *

註：有＊是 armortized cost

一、 Max heap、Min heap

1. 定義(Max heap)：
   a.是一棵 complete binary tree
   b.每個 node 的 key value 皆大於 children 的 key value
2. Insert 的步驟 O(logn)：每插入一個節點就由下往上做調整(跟 parent 做比較)
3. Delete 的步驟 O(logn)：刪除根節點後，把最後一個元素拿出來點，然後由 root 往下做調整(跟左右 child 中大的或小的比較，小的往上調整，直到比 child 大/小，或是到了 leave，也就是 2i>n)
4. 資料結構：array

二、 Min-Max Heap

1. 定義：
   a.是一棵 **complete binary tree**
   b.min level 與 max level 間隔出現，且 root 為 min level
   c.設 x 為任一 subtree 的 root
   若 x 在 min level，則 x 是 subtree 中最小的 key
   若 x 在 max level，則 x 是 subtree 中最大的 key
2. 應用在 doubled ended priority queue
3. Insert 的步驟 O(logn)：

	x < parent	x > parent
x 插入在 min level	在 min level 中調整	與 parent 互換，在 max level 中調整
x 插入在 max level	與 parent 互換，在 min level 中調整	在 max level 中調整

找到要如何 Insert 之後，可以分兩種情況：在 MaxLevel 和在 MinLevel，這時候插入就分兩種，以 Max 為例（唸 Delete 不如唸這個）：

```
Void VerifyMax ( element h[], int I, element x )
{
      int gp;
      gp = i/ 4;     //  祖父
      while(gp !=){          //控制什麼時候要跳出來
            if( x > h[gp])
            {     h[i] = h[gp];//把祖父拉下來,自己往上繼續比
                  i = gp;
                  gp = i / 4;
            }
            else gp = 0;          //只是要控制不要繼續比了
   h[i] = x ;
}
```

4. Delete 的步驟 O(logn)：(有點繁瑣)(以下以 delete min 為例，有時間唸，沒時間算了啦)
   a.挑出最後一個元素 x 來做比較
   b.挑出最小的元素 y
   c.若 x < y，則用 x 做 root，動作完成
   d.若 x > y
   狀況 1：若 y 是兒子(max level)，則 x、y 互換，動作完成
   狀況 2：若 y 是孫子(min level)，則 x、y 互換，然後 x 在新的位置中與 parent(max level)做比較
   狀況 2-1：若 x < parent，則以 x 為 subtree 的 root 重覆步驟 b 以後的動作
   狀況 2-2：若 x > parent，x 與 parent z 互換，然後以剛換下來的 parent z 為 subtree 的 root 重覆步驟 b 以後的動作
5. 資料結構：array

三、 Deaps(Double-ended heaps)

1. 定義：
   a.是一棵 complete binary tree，可為 empty
   b.根節點不存放 key 值
   c. left sub-tree 是 min-heap，right sub-tree 是 max-heap
   d.當 right sub-tree 不是 empty tree 時，left sub-tree 中每個節點 i 在 right sub-tree 中皆可找到一個對應節點(partner) j，且符合 d[i]<d[j]之限制，若 right sub-tree 的對應節點不存在，則取對應節點的 parent 為 j
2. partner 的求法 $j=i+2^{\lfloor \log_2 i \rfloor -1}$，若$j>n \rightarrow j=\left\lfloor \frac{j}{2} \right\rfloor$
   公式原理：第 h 層節點 i 的右側 partner 一定是 i 加上 h 層(如果是滿的時候)的總節點數的一半，第 h 層的總節點數 $= 2^{\lfloor \log i \rfloor}$
3. Insert 步驟 O(logn)：

	x < partner	x > partner
x 插入在 min heap	在 min heap 調整	與 partner 互換，在 max heap 做調整
x 插入在 max heap	與 partner 互換，在 min heap 做調整	在 max heap 調整

4. Delete 步驟 O(logn)：(以 delete min 為例)
   先在 min heap 中由上往下做調整(做完會空一格出來)，然後把最後一個元素拿出來做 insert 的動作

四、 Leftist Trees(左撇子樹) – 可以說是 Min-Leftist Tree

1. 用途：priority queues 的合併時間 O(logn)，若使用 heap 結構要花 O(n)的時間
2. 定義：
   a.是一棵 binary tree，可為 empty
   b.每個內部節點皆滿足
   shortest(LeftChild(x))>=shortest(RightChild(x))
   根節點到左子樹的外部節點的最短路徑 >=
   根節點到右子樹的外部節點的最短路徑
3. Combine 步驟 O(logn)：

   有兩顆樹要 Combine 的 root a > b，那麼就把 a 的右子樹去跟 b 做合併，合併完在 a 的右子樹這樣子 recursively 下去做
   combine 之後要再由下往上檢查每一 subtree 的 root 是否滿足 shortest(LeftChild(x))>=shortest(RightChild(x))，若否則左右互換。
   P15 有證明這個由上而下的合併過程及由下而上的調整過程時間是 O(logn)
4. Insert 步驟 O(logn)：就建一個節點的 leftist tree 就好了，然後兩顆樹要 Combine
5. Delete min O(logn)步驟：就是把根節點殺掉，然後原本的左右子樹做 combine 的動作
6. 資料結構：每個節點都要有 shortest path 的記錄

五、 Skew Heaps

1. 出現在 Horowitz 的習題 p9-28，用途??
2. Insert 步驟：同 leftist tree
3. Combine 步驟：由上而下過程同 leftist tree，但由下而上的調整過程不同，它是不管如何，每 check 一個 subtree 的 root，就一定左右互換(why??太閒!?)
4. Delete min 步驟：同 leftist tree

六、 Binomial Heaps 二項式堆積

1. 定義：Binomial tree of degree k
   a.k=0，為一個單一節點的 tree，以 B_0 表示
   b.k>0，含有一個 root，且具有 $B_0,B_1,\ldots\ldots B_{k-1}$ 共 k 個子樹
   c.B_k 含有 2^k 個 nodes
2. 分為 min binomial heap、max binomial heap
3. 用途：(課本 p9-29)使用分攤成本法，我們可以獲得一連串的運算的複雜度更精密的上限值。
4. Insert 步驟 O(1)：
   a.將新的資料建立一個 degree=0 的 B-Heap
   b.將新節點加入原來 B-Heaps 的頂層串列中
   c.指標指向最小的 root
5. Combine 步驟 O(1)：degree 一樣的合成一棵，combine 動作在 delete min 時觸發
6. Delete min 步驟 O(logn)
   （媽呀上課的看不懂，但看講義懂了，你好厲害）
   刪除最小的 root，然後反覆把 degree 相同的 binomial trees 合併成一棵，直到所有的 tree 的 degree 都不同為止
   時間分析：詳見 p18、19
   O(MaxDegree + s)
   MaxDegree：是合併過程中用到的最大陣列
   MaxDegree <= $\lfloor \log_2 n \rfloor$
   s：若 delete min 後有 s 棵 tree 要合併，則最多合併 s-1 次
   其中 s 又可分析如下
   s = #insert + lastsize + u –1

#insert	上一次 delete min 到這一次 delete min 之間的 insert 次數
lastsize	上一次 delete min 後，trees 的個數
u	本次 delete min 刪除的 root 後會分裂出來的 trees 個數

   其中#insert 的成本分攤給每一次的 insert
   而 lastsize 的成本(O(logn))分攤給上一次的 delete min
   所以剩下 u 為本次的成本
   其中 u <= $\lfloor \log_2 n \rfloor$
   因此 delete min 的分攤成本為 O(logn)
7. 資料結構：頂層用 Circular Linked List 串連

七、 Fibonacci Heaps (應該是強在可以做 Decrease key)

1. 分為 min Fibonacci heap、max Fibonacci heap
2. B-Heaps 是 F-Heaps 的一個特例
3. Insert 步驟 O(1)：同 B-Heaps
4. Delete 步驟 O(logn)：同 B-Heaps，但不合併 degree 相同的 trees
5. Decrease key O(1)：任一節點的 key 值減少某些值，減少後若還是大於 parent，則留著，否則整個子樹獨立到最頂層
6. cascading cut：若 decrease key 發生子樹脫離的狀況時，檢查其父節點，若其父節點已失去過一個子樹，則本次再失去時就要跟著做 cascading cut
7. 資料結構：頂層用 double linked list 串連
8. 應用：
   在 Graph 的 Single source all destination shortest path algorithm 中
   使用 Dijkstra 的演算法是 $O(n^2)$
   如果應用 F-Heaps 來改良其演算法

```
for i =1 to n-2
{
  u=choose_min( );      //註一
  for every(u,w)
  {
        if (not visited[w])
        {
              if (dist[u]+cost[u][w]<dist[w])
                    dist[w]=dist[u]+cost[u,w];    //註二
        }
  }
}
```

▲註一：選出未拜訪的頂點中距離成本最小的，運用 F-Heaps 的 delete min 運算→時間 O(logn)
▲註二：相當於 decrease key 的運算，每次用掉 O(1)的時間
▲for every(u,w)這行一共有 e 個邊，所以會執行 O(e)次。因此，此演算法搭配 Adjacency list 的資料結構可以在 O(nlogn+e)的時間完成
★因為 decrease key 運算會使得下一次的 choose_min( )，可以在 O(logn)的時間內完成，否則若用一般的搜尋最小值的方法要花 O(n)的時間，則時間壓不下 $O(n^2)$

八、 Armotrized Cost 分析

1. 這個在講義上，用看的好了
2. 分三種，**總計法**（全部加起來然後/n）、**會計法**(做完要漂亮的分配)、**位能法**(這蝦米？)

第九章 Graphs

一、名詞

1. Digraph(即 directed graph 有向圖)，習慣上用<v1,v2>表示 v1→v2 的邊
2. Undirected Graph：無向圖，習慣上用(v1,v2)表示 v1、v2 之間的邊
3. Complete Graph：一無向圖中，任意兩頂點間皆有 edge 存在；無向圖的完全圖形總邊數是 n*(n+1)/2 而有向圖則是 n*(n+1)
4. Multi-Graph：一圖形中，相同的邊重覆多次，一般不討論這種圖形
5. Simple Path：在一路徑中除了起點與終點可以相同之外(不同亦可)，其餘頂點不可以重覆
6. Adjacent (相鄰)
7. Connected 存在有 path
8. Incident (連接，好像跟 Adjency 一樣捏)
9. Connected graph：一圖形中任兩頂點皆有路徑到對方；注意它跟 complete graph 的差別，complete graph 是指任兩點間皆相鄰
10. Connected component：同上，component 是指圖形的一部分
11. Free Tree：connected、acyclic 的無向圖亦稱為 Free Tree
12. Sink Tree：(p57)算好 single source all destination 有向圖中的所有最短路徑後，把從終點往回走到起點的路徑方向都反轉所形成的路徑
13. Strong Connected Graph：強連通圖形，用在有向圖，也是圖形中任兩頂點皆有路徑可以到達
14. Strong Connected Component：用在有向圖
15. Degree：無向圖是指所有與其 incident 的邊的總數；有向圖則區分「In-Degree」、「Out-Degree」
16. Eularian Cycle：尤拉循環；所有頂點的 degree 皆為偶數；因而從任一頂點開始，經過所有的邊僅一次，並回到原來的地方
17. Eularian Chain(Path)：尤拉路徑；除了起點跟終點的 degree 為奇數，其它皆為偶數；因而從任一頂點開始，經過所有的邊僅一次，不一定回到出發點
18. Hamiltonian Cycle：找出一個遊走圖形 G 中所有「頂點」各一次並回到起點的路徑。
19. Cycle：要回到起點；Path：不一定要回到起點
20. Bipartite Graph：二元圖
    圖形 G 的頂點集合 V，分為 disjoint setV1、V2
    亦即 V1∪V2=V，V1∩V2=empty 而且
    V1 中的任兩頂點在 G 中皆不相鄰
    V2 中的任兩頂點在 G 中皆不相鄰
21. Biconnected Components：不含有 articulation point 的 graph
22. Articulation point：圖形中一旦移除就會造成圖形一分為二的節點

二、圖形表示法

1. **Adjacent matrix**
2. **Adjacent list**
   注意一下它的 sequential representation(循序表示法)
3. **Inverse adjacency lists**
   有向圖 Adjacency list 表示法找 out-degree 很快，而找 in-degree 就比較麻煩，反轉鄰接串列可以彌補這個缺點。而且也應用在 AOE，在反推各頂點的最晚完成時間時使用
4. **Adjacent multi-list**：有向圖 ok，無向圖??
   鄰接多重串列可以讓 in-degree、out-degree 的計算比較快
5. **Incidence matrix**：紀錄 vertex 與 edge 的關係

		Adjacency matrix	Adjacency list
無向圖	計算 Vi 的 Degree	O(n)	O(len(list[Vi])
	計算 G 的總邊數	$O(n^2)$	O(n+e)
	找出 Vi 的鄰接頂點	O(n)	O(len(list[Vi])
	檢查 Vi，Vj 是否相鄰	O(1)	O(len(list[Vi])
有向圖	Vi 的 in-degree	$O(n^2)$	O(n+e)
	Vi 的 out-degree	O(n)	O(len(list[Vi])

三、圖形搜尋法

1. DFS(Depth First Search)：運用遞迴或直接用 Stack
   ▲DFS 是在把頂點 pop 出來時 visited[Vi]才設為 true
   ▲DFS 不使用 stack 的 iterative 方法，一邊執行 DFS 時就記載各 node 的 prev[Vi]，走到底時就延著這個指標回來
2. BFS(Breadth First Search)：運用 Queue
   BFS 是在把頂點 add_queue 時就把 visited[Vi]設為 true
3. **DFS、BFS 的時間複雜度：**
   O(n+e)或 O(e)使用 adjacent list
   $O(n^2)$使用 adjacent matrix
4. D-search：DFS 的另一種版本，主要差別是已經在 Stack 中的 node，就不再放入
5. DFS、BFS 都可應用在無向圖的 connected components 尋找
6. DFS 還適合用在有向圖的 connected components 尋找
7. 檢查有向圖是否有 cycle 的方法：

   a.DFS：往下的途中要做記號；到了底部回程時拿掉記號
   b.Topological sort：還沒搜尋完全部頂點就發生找不到 in-degree 為 0 的頂點的情況

8. Bipartite graph 的驗證

```
Procedure DFS(v)
{
        for each edge (v,w)
        {
                if (group[w]=0)
                {
                        group[w]=3-group[v];
                        DFS(w);
                }
                else if (group[w]=group[v])
                        output "not a bipartite graph";
        }
}
```

Group[v]=	0	尚未追蹤
	1	v 已追蹤且 v 屬於 V1
	2	v 已追蹤且 v 屬於 V2

9. 找到包含自已點的最小 cycle

```
DFS_change(v)
{
     visited[v] = true;
     for each vertex < v, w >
     {
     if   (w= Va) { length[v] = 1 ;}
     else { if not visited[w] then DFS_Change(w);
            length[ v] = min (length[v],length[w]+1);
   //每一個點都會比一次，之後會取到最小的 length[w]+1
          }
     }
}

main   就把所有 visited 都設為 false，
                  length[v]設無限大
       去做 DFS(Va)

Why?之後得到的 Length 就是自已走到自已的最短距離了
```

10. 名人問題

    使用 adjacency list 來表示圖形，統計每個 degree 是否有人 in-degree 為 n-1 out-degree 是 0

四、Spanning Tree

1. 定義：
   圖形 G 中有 n 個頂點，若有 n-1 邊可以連接 n 個頂點，則此 n-1 個邊連接而成的 connected graph 即為 spanning tree，如果這 n-1 個邊的成本和是最小，則亦稱為 minimum cost spanning tree
2. 可用 DFS、BFS 搜尋得到 spanning tree
3. 尋找 minimum cost spanning tree 的方法
   a. Kruskal's algorithm：O(eloge+elogn)、O(eloge)、O(elogn)

b. Prim's algorithm：$O(n^2)$
c. Sollin's algorithm：$O(n^2)$
要會列出它們計算過程的詳細列式

▲Kruskal's algorithm：

```
T={};
While (T 中邊的個數<n-1 且 E 不是 empty)
{
        choose a least cost edge(v,w) from E;          //註一
        delete (v,w) from E;
        if ((v,w)不會造成 T 中的迴圈);                 //註二
                add(v,w) to T;
        else
                discard (v,w);
}
if (T 中邊的個數<n-1)
        printf("找不到 spanning tree");
```

1.註一：根據所有的 edge 的成本建立 min-heap。O(eloge)
2.尋找所有未拜訪過的頂點中成本最小的
步驟 1 已建立 min-heap，故選取最小的邊的時間 O(1)，重整 min-heap 的時間 O(loge)
3.檢查加入本次挑到的邊是否會造成 cyclic，用到 disjoint set 的 union、find 方法 O(logn)
4.步驟 2、3 最多重覆 e 次(雖然 while 迴圈條件是<n-1，但是 worst case 是所有的邊都被檢查過，所以是 e 次)
→總執行次數 O(eloge+eloge+elogn)
表達成 O(eloge+elogn)、O(eloge)、O(elogn)皆可

▲Prim's algorithm：

```
T={};                   //T 是邊的集合，TV 是頂點的集合
TV={0};                 //從頂點 0 開始
While(T 中的邊數<n-1)
{
        挑選一個最小成本的邊(u,v)，其中 u 屬於 TV，且 v
不屬於 TV;
        if (不存在這樣的邊) break;
        add v to TV;
        add (u,v) to T;
}
if (T 中邊的個數<n-1)
        printf("找不到 spanning tree");
```

1.任選一頂點開始，把頂點分成兩群 U、V
2.每次尋找連接 U、V 之間的最短路徑
3.時間複雜度：$O(n^2)$
(詳細演算法見王四回 p49 的 ex36)，while 迴圈共執行 n 次，而步驟 add v to TV 每次執行會花 O(n)的時間，有點類似 Dijkstra 的概念，每加入一個新的頂點到 TV 集合中，就重新計算 TV 集合透過新加入的點到另一個集合中的頂點是否有更短的距離，猜可能是一個 known 和最近會被挑的點來表示）

prim 有另外的方法：用 adjency matrix list+min heap 表示，可以得到 O(eloge)的時間

▲Sollin's algorithm：
Phase I ：每個頂點的最短邊先找好(所以會有一堆兩點一邊的 component) $O(n^2)$
PhaseⅡ：用最小成本的邊連接未連通的 component $O(n^2)$

▲常問那個快
不一定，要看邊的個數，因為圖形的邊介於(n-1)~(n(n-1)/2)
所以邊很多時，Kruskal 是 $O(n^2*\log n)$，Prim 是 $O(n^2)$，
kruskal 適合接近 n 的（邊少）**Prim 適合接近 n^2** 的（邊多）

Kruskal	Prim	Sollin
O(eloge)	$O(n^2)$	$O(n^2)$
需檢查加入的邊是否會造成 cycle	不需 check	不需 check
搜尋成本最低的 edge 逐次加入	運用互斥集合的方式找兩個集合相連最短的邊	

五、 Biconnected Components

1. 定義：
   a.不含有 articulation point 的 graph
   b.所謂 articulation point 是指從圖形中移除此點後，圖形會因而形成「二個以上」分離的連通單元
2. 名詞：
   a. back edges：由子孫回到祖先的邊(不含父親)
   b. forward edges：由祖先到後代的邊(不含兒子)
   c. cross edges：頂點可以相，但之間沒有祖先後代的關係

▲項目 3、4 的前提是當圖形以 DFS 搜尋時

3. 無向圖：沒有 cross edge，back edge=forward edge
4. 有向圖：可能有 cross edge，back edge<>forward edge
5. 計算方式：
   dfn(depth first number)：是指用 DFS 搜尋圖形時節點被搜尋到的順序
   low(u)：min{dfn(u),
   min{low(w) | w 是 u 的 child},
   min{dfn(w) | (u,w)是 back edge}}
   當 u 的某個 child w 滿足 low(w)>=dfn(u)時，則 u 為 articulation point

六、 最短路徑問題

1. Single source all destination**(Dijkstra's algorithm)**

```
Void shortestpath(int v,int cost[][max_vertices],int distance[],int
n,short int found[])
{
        int i,u,w;
        for (i=0,i<n;i++)
        {found[i]=False;
         distance[i]=cost[v][i];}
        found[v]=true;
        distance[v]=0;
        for (i=0;i<n-2;i++)
        {
            u=choose(distance,n,found);//選擇未拜訪過的頂點
            中，distance 最小的
            found[u]=true;
            for(w=0;w<n;w++)
                if (!found[w])
                    if (distance[u]+cost[u][w]<distance[w])
                        distance[w]=distance[u]+cost[u][w];
        }
}
```

Dijkstra's algorithm 時間複雜度：$O(n^2)$ adjacent matrix
O(e+nlogn) adjacent list with Fibonacci heap

2. negative edge：成本值為負的邊，會使得 Dijkstra 不正確(猛坤說：用暴力法解)
   negative cost cycle：成本和為負值的循環部分
3. All pairs shortest paths(Floyd-Warshall)
   使用 Dijkstra 要 $O(n^3)$
   使用 Floyd-Warshall：$\theta(n^3)$

```
for k=1 to n
    for i=1 to n
        for j=1 to n
            if a[i,k]+a[k,j]<a[i,j]
```

計算過程的 A^0、A^1、A^2…………

```
路徑輸出的演算法
find_path(x,y)
{
        while(x <> y)
        {
                print(x, π (x,y));
                x= π (x,y);
        }
}
```

七、圖形的直徑、半徑問題

直徑：兩點之間最長的距離

半徑：對任一子樹的 root 而言，其左右子樹中的最大半徑+1 即為該 root 所在子樹的半徑

Devide and conquer，找出 sub-tree 的最短路徑，假如 empty，則 diameter = 0 ,redius = -1

去找，子樹中 diameter 大的為 d，半徑中最大的是 r1 第二大的 r2 則：

diameter = max { d, r1+r2+2}

radius r1 +1

▲要求圖形的直徑：用 all pair shortest path 計算出結果後，表格中的最大值即為圖形的直徑。

▲要求使圖形具有最小半徑的根節點該如何選擇：則承上，從每一列中挑出最大的，再從最大的中取最小的，則以該點為 root 有最小半徑

八、Transitive closure(遞移封閉集合)

A(i,j)=1 if i→j 有=1 的路徑相通

Transitive closure：$A^+(i,j)=1$ if i→j 有>=1 的路徑相通

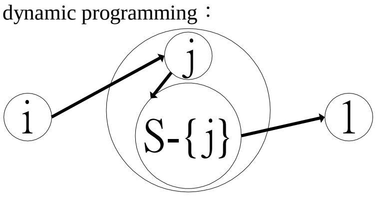
Reflexive closure：$A^*(i,j)=1$ if i→j 有>=0 的路徑相通

注意有向圖中的 $A^+(i,i)$，如果有路回到自己 $A^+(i,i)=1$

九、Traveling Salesperson Problem(TSP 巡迴銷售員問題)

在一個圖形中，找出一條最短路徑，可以從一個頂點開始，遊走所有的頂點各一次，然後回到起點

dynamic programming：



$$g(i,S)=\min_{j\in S}\{C_{ij}+g(j,S-\{j\})\}$$

i 是起點，S 是必需經過 1 次的頂點集合

時間複雜度：$O(n^2 2^n)$

十、Topological sort

錳坤的解法比較好理解：

用一個 queue，每次去 check，看有沒有減了以後 in-degree=0 的點，有就把他加到 queue 裡面（也可以用 stack 做，但出來的結果會不一樣，效果不同）

```
void Topsort(Graph G);       /*      O(|E|+|V|)      */
{       queue Q;
        int Counter=0;
        vertex V,W;
        Q=CreateQueue(NumVertex);       MakeEmpty(Q);
        for each vertex V
                if   (Indegree[V] == 0)
                        Enqueue(V,Q);
        While (!IsEmpty(Q)) {
                V=Dequeue(Q);
                TopNum[V] = ++Counter;
                for each W adjacent to V
                        if (--Indegree[W] == 0)
                                Enqueue(W,Q);           }
        if (Counter != NumVertex)
                Error("Cycle!");
        DisposeQueue(Q);
}
```

小強解法，你在講什麼阿小強？

```
struct node{
        int vertex;
        node *link;
};

struct hdnode{
        int count;
        node *link;
};

//先計算好每個頂點的 in-degree：O(n+e)

void topsort(hdnode graph[],int n)
{
        int top=-1;
        for (i=0;i<n;i++)
                if (!graph[i].count)
                {
                        graph[i].count=top;    /*找到第一個可輸出的頂點後，count 欄位就可以移做它用，在此將之模擬成 stack 的功能*/
                        top=i;
                }

        for (i=0;i<n;i++)
        {
                if (top= =-1) {"圖形有迴圈"};
                else
                {
                        j=top;
                        top=graph[count].count;
                        ouput(j);
                        for (w=graph[j].link;w;w=w->link)
                        {
                                k=w->vertex;
                                graph[k].count--;
                                if (!graph[k].count)
                                {
                                        graph[k].count=top;
                                        top=k;
                                }
                        }
                }
        }
}
```

Time complexity：O(n+e)

十一、 AOV network

1. 是一個有向圖
2. 以頂點表示工作或活動
3. 以邊表示工作之間的先後順序
4. 應用在 topological sort
5. 時間複雜度 O(n+e)

十二、 AOE network

1. 是一個有向圖
2. 邊代表工作或活動
3. 頂點代表事件
4. critical path 即起點到終點最長的路徑
5. ee(k)：表示 event k 最早可能開始的時間，亦即 ee(k)=由起點到頂點 k 的最長路徑

藍色 code 是為了算 AOE 調整的

```
void topsort(hdnode graph[],int n)
{
    int top=-1;
    for (i=0;i<n;i++)
        ee[i]=0;
        if (!graph[i].count)
        {
            graph[i].count=top;
            top=i;
        }

    for (i=0;i<n;i++)
    {
        if (top= =-1) {"圖形有迴圈"};
        else
        {
            j=top;
            top=graph[count].count;
            output (j,ee[j]);
            for (w=graph[j].link;w;w=w->link)
            {
                k=w->vertex;
                graph[k].count--;
                if (!graph[k].count)
                {
                    graph[k].count=top;
                    top=k;
                ee(k)=max{ee(k),ee(j)+DUR(j,k)};
                }
            }
        }
    }
}
```

6. le(k)：表示 event k 最晚開始而不會延遲工作的總進度的時間
   計算 le(k)要先建立 inverse adjacency list(因為要從尾往回追蹤)，然後 count 欄位要記錄的是 out-degree
   把上述程式中的 ee( )都改成 le( )
   ee(k)=max{ee(k),ee(j)+DUR(j,k)};
   改成 le(k)=min{le(k),le(j)-DUR(j,k)};

可以定義一個 slack time (v, w)：le(w) -ee(v) –cost(v,w)
critical path 就是 longest path 也就是 slack time = 0 的 path

對於一個 activity(i,j)也可以定義：
e(i)：最早可以開始的時間 e(i) = ee(k)
l(i)：做晚要開始而不會延遲的時間，l(i) = le(j) – cost(i,j)

十三、 最大流量問題

小強說只能看運氣，但好像有方法可以解，參考離散。

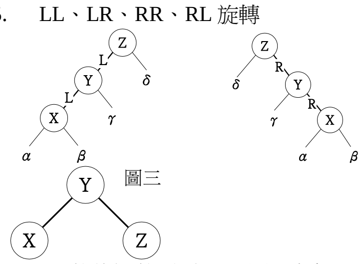
第十章 搜尋結構

一、 **E = I + 2N；證明如下：**
N=N_L+N_R
I=(I_L+N_L)+(I_L+N_L)
E=(E_L+N_L+1)+(E_R+N_R+1)
E 是外部路徑總長：由 root 到所有外部節點的路徑長總合
I 是內部路徑總長：由 root 到所有內部節點的路徑長總合
N 是節點總數
介紹這些名詞的用意在於平均搜尋次數的計算
$\frac{I}{n}+1=2\ln n\approx 1.39\log n$

二、 **Huffman Algorithm(霍夫曼 Tree)**
這個方法不能用在 OBST 是因為 OBST 是有順序關係的，而 Huffman 編碼的節點是**沒有順序關係的**

三、 **OBST(Optimal Binary Search Trees)**
P10 的計算方式要記得(dynamic programming)
原理是某一節點為根節點時會有最小成本
p_i：內部節點 i 的出現機率；q_i：外部節點 i 的出現機率
$initial：W_{ii}=q_j;\ C_{ii}=0;\ R_{ii}=0$ 時間複雜度：O(n^2)
$W_{ij}=W_{i,j-1}+p_j+q_j$
$C_{ij}=\min\{C_{i,k-1}+C_{k,j}\}+W_{ij}$

四、 **AVL-Tree(Height-Balanced Tree)（考政大要注意感覺錳坤很愛）**
1. 定義：
   a.是 binary search tree，可為 empty
   b.任一子樹的根節點，左右子樹的高差<=1
   c.左右子樹也都是 AVL-Tree
2. Balance Factor(BF)，就是左右子樹的高差
3. Insert、Delete、Search 都是 O(logn)
   logn 的時間主要花在 BF 值的檢查調整上，因為 insert、delete 最多只要兩次旋轉就完成(point 接一接)，只是旋轉可能發生在由下往上的某個節點上，所以要檢查 logn 的時間
4. 當 AVL-tree 的節點數最少時，存在著公式：$n=F_{h+2}-1$
   其中 F 是費氏級數
5. LL、LR、RR、RL 旋轉



圖三

6. BF 值的調整原則(以上圖三為例)

Insert	0→±1	±1→0	Delete	0→±1	±1→0
X	Y 的 BF 值加 1	Y 的 BF 值不變	X	Y 的 BF 值不變	Y 的 BF 值減 1
Z	Y 的 BF 值減 1	Y 的 BF 值不變	Z	Y 的 BF 值不變	Y 的 BF 值加 1

7. AVL-tree 的 Data Structure

left	bf	data	Right

五、 **Splay Tree(斜張樹)**
1. 定義：
   a.是 binary search tree
   b.每個 operation 完成後都需進行 splay 運算
2. splay 的起點
   a. search：由搜尋到的節點進行 splay
   b. insert：由插入的節點進行 splay
   c. delete：由刪除的節點的 parent 進行 splay
   時間都在 O(logn)~O(n/2)
3. splay 的用意：
   a.連某一資料會連續存取時
   temporal locality. Ex: loop、stack
   b.最近被存取過的資料也會離根節點很近，所以存取比較快
   spatial locality. Ex: array、循序存取

六、 M-way search tree
1. 定義：
   a.可為 empty
   b.每個 node 的 degree<=m
   c.Ki<Ki+1，Si<Ki<Si+1<Ki+1……..
   d.leaf node 可以不在同一 level

七、 B-Tree of order m
1. 定義：
   a.m-way search tree，可為 empty
   b.root 至少有兩個 children
   c.其它 node 至少有 $\left\lceil \frac{m}{2} \right\rceil$ 個 children
   d.所有 leaf node 皆在同一 level
2. $2\left\lceil \frac{m}{2} \right\rceil^{h-1}-1\le keys數目\le m^h-1$
3. B-Tree 的 insert、delete 的旋轉、合併條件與方式特別注意(練習 p23 範例；p24 有一堆文字敘述)
4. B-Tree 的資料結構

P_0	K_1	R_1	P_1	K_2	R_2	P_2	…	P_{n-1}	K_n	R_3	P_n

   P：指向下一層的 node
   K：key 值
   R：key 值所對應的實際資料存放的 block 指標
5. 其它怪樹 B’-Tree、B*-Tree
6. B^+-Tree
   與 B-Tree 最大的不同在於所有的資料都存放在 leaf node，並且 leaf node 間還建立 link
   B^+-Tree 的資料結構

P_0	K_1	P_1	K_2	P_2	…	P_{n-1}	K_n	P_n

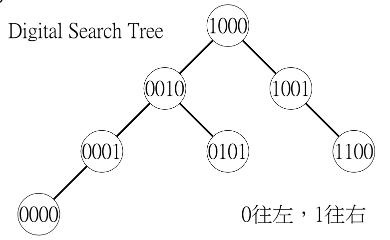
   因此 B^+-Tree 也可進行 range search
7. B’-Tree
   跟 B^+-Tree 不同在於 leaf 的節點有串起來

八、 2-3 Tree、**2-3-4 Tree**
1. 可視為 **order = 4** 的 B-Tree
2. 2-3-4 Tree 的 insert、delete 方式可以用 B-Tree 的 **Backward insertion**，也可以採用其專屬的 **Forward insertion**。
3. **Forward insertion**
   由於 B-Tree 的操作要由上往下找到所在位置 O(logn)，若需調整，則需再由下往上調整 O(logn)；而使用 Forward insertion 只要由上往下一次的過程即完成。
   ▲做法如下：
   insert 時，由上而下搜尋插入節點的過程中，遇到 3node 或 4node 就要進行 split
   delete 時，在由上而下搜尋的過程中，遇到 2node 就先找兄弟點併成 3node 或 4node

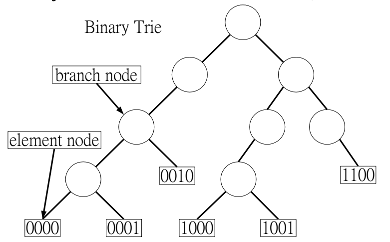
九、 **Red-Black Tree(紅黑樹)**

1. 把 2-3-4 Tree 中所有 keys>1 的 node 都做旋轉，這樣的動作產生的邊或 node，就是紅邊或紅節點，而原先就存在的點或邊就是黑邊或黑節點
2. 操作的規則很繁複，小強老師說就用 2-3-4 樹的 Forward insertion 版本進行操作，然後再將結果轉成紅黑樹。
3. 紅黑樹的用意在於提升指標的使用率，因為 degree 越高，指標使用率可能越差
4. height always O(logn)

十、 Digital Search Trees



▲Binary Tries：可以應用在字典的單字查詢中



▲Compressed Binary Tries

上圖的 branch node 如果都沒有連接的 element node，則可以刪除；也因此所有的 branch node 要加一個記錄自己是第幾層

▲Patricia：p34 的 insert 過程練習一下，挺瑣碎的

第十一章 內部排序；十二章 外部排序

一、 各種排序法比較

		Best case	Average case	Worst case	Stable Unstable	Additional stroage	說明
Insertion sort		O(n) 比較：n-1 次 搬動：0 次 **sorted data**	$O(n^2)$ 比較次數： n(n-1)/4+(n-1)-$\sum_{i=2}^{n}\frac{1}{i}$ 搬動：n(n-1)/4 次	$O(n^2)$ 比較：n(n-1)/2 次 搬動：n(n-1)/2 次 **reversed sorted data**	**Stable**	O(1)	
Bubble sort		O(n) 比較：n-1 次 搬動：0 次 **sorted data**	$O(n^2)$ 比較次數： n(n-1)/4+(n-1)-$\sum_{i=2}^{n}\frac{1}{i}$ 搬動：n(n-1)/4 次	$O(n^2)$ 比較：n(n-1)/2 次 搬動：n(n-1)/2 次 **reversed sorted data**	**Stable**	O(1)	▲best case：是指發現整趟比較過程都沒有發生任何交換的動作，則排序結束 ▲可由左往右，也可由右往左
Selection sort		$O(n^2)$ 比較：n(n-1)/2 次 搬動：(n-1)次			O(1)：**UnStable** 只用一個額外空間的實作法是 unstable O(n)：**Stable** 另用一個一樣 size 的陣列的實作法是 stable		▲逐一挑出最大或最小的，然後做交換，因此三種 case 都一樣慢，而且即使已排序好的資料，雖然實際上沒有搬動，但還是做了 n-1 次的自己跟自己互搬
Shell sort		$O(n\log^2 n)$、$O(n^{1.25})$、$O(n^{1+1/\sqrt{\lg n}})$ 應該不會考吧			**UnStable**	O(1)	
Quick sort		O(nlogn)	O(nlogn)	$O(n^2)$ 可應用 median of three、randomly choosing 等方法來降低 wort case 的發生	**UnStable**	O(logn)~O(n)	▲Quick sort 的 average case 證明：(詳見 p16)，Cn 是指 n 項資料做 quick sort 的比較比較次數 $C_n = n+1+\frac{1}{n}\sum_{k=1}^{n}(C_{k-1}+C_{n-k})$ 計算結果為 2lnn
Merge sort		O(nlogn) 共$\lceil \log_2 n \rceil$passes 每一回，比較(n/2~n-1 次)，搬動 n 次			**Stable**	O(n)	
Heap sort		O(nlogn) Construct max heap：O(n) 逐一輸出，並調整 max heap：O(nlogn)			**UnStable**	O(1)	▲heap sort 的 overhead 比 quicksort 來得大，所以一般資料量時還是會採用 quicksort
Counting sort		$O(n^2)$ 比較：n(n-1)/2 次 搬動：n 次			**Stable**	O(n)	▲適用在 record 很長，而 key 值很短的時候
Distributive counting sort		M：排序資料的最大 key 值 N：資料總數 O(M+N)			**Stable**	O(M+N)	▲適用在重覆的 key 值多，且 key 值不大的狀況
Radix(Bucket sort)	LSD	d：執行回數，亦即資料中的最長資料長度 n：資料筆數 r：radix 基數，例如用 16 進位為 16 O(d(n+r)) n 是每次都要對 n 筆資料找到對應的桶子 r 的時間是每回結束，要把各桶子中的資料串起來所花的時間			**Stable**	O(n+r)	▲Least Significant Digit First ▲額外空間 O(n+r)其中 n 是每筆資料要加上用來指向下一筆資料的指標；r，其實是 2r 是每個桶子用來指向記錄的指標，一個指向放在桶子中的資料的頭，一個指向尾。
	MSD	$O(n\log_r n)$		O(dn)	**Stable**	$O(\log_r n)$~O(n)	▲Most Significant Digit First ▲又稱為 binary tree distribution sort ▲這個要 recursive 了 適用在不定長度型態的比較上例如 ABC NETWORK

▲外部排序

		時間複雜度
K-way merge on m runs		比較次數 n．(k-1)．$\log_k m$ 搬動次數 n．$\log_k m$ 一共是 $\log_k m$ 的 passes，每個 pass 要比較 n．(k-1)次
Selection Tree	Winner's Tree	在 K-way merge 中每個 pass 的比較 n．(k-1)次，利用 winner't tree 的特性可以降低為 n．$\log_2 k$ 因此總執行次數為 n．$\log_2 k$．$\log_k m$，式子可以轉換成 n．$\log_2 m$ ，使得時間複雜度與 k 值無關
	Loser's Tree	時間複雜度還是 n．$\log_2 k$．$\log_k m$，不過在 $\log_2 k$ 的實際執行上比 winner's 要來得省時，因為 loser's tree 只要在樹中由上而下一趟就完成，而 winner's tree 要由上而下，再由下而上兩趟

▲何謂 **comparison sort?**
排序時若是用比較兩筆資料鍵值大小，再決定如何移動資料者，稱為 comparison sort
▲何謂 **distributive sort?**
排序時根據鍵值計算或分析出資料所應存放之位置，直接將資料存入該位置

二、 inversion table

▲反轉表：例如 5,9,1,8,2,6,4,7,3，其反轉表為 2,3,6,4,0,2,2,1,0，因為 1 的左邊有兩筆資料比它大，以此類推
▲由反轉表的觀察得知，反轉表的最大值決定 bubble sort 的執行次數，而反轉表的總和，決定 bubble sort 的交換次數

三、 QuickSort

```
void quicksort(int a[],int left,int right)
{
        if (left<right)
        {
                int i,j,pivot;
                pivot=a[left];
                i=left;
                j=right+1;
                do {
                                do
                                        i++;
                                while(a[i]<pivot);
                                do
                                        j--;
                                while(a[j]>pivot);
                                if (i<j)
                                        swap(a[i],a[j]);
                } while (i<j);

                swap(a[left],a[j]);
                quicksort(left,j-1);
                quicksort(j+1,right);
        }
}
```

四、 排序時間的下限

▲n 筆資料的排序可能有 n!種結果
則決策樹總共有 2n!-1 個節點，則存在下列的關係式

$$2^{h-1} \le 2n! - 1 \le 2^h - 1$$

h>=log(2n!)>Ω(nlogn)
▲the lower bound of the **worst case** for **comparison and exchange** sort is Ω(nlogn)

五、 Counting sort

```
for (i=1;i<=n,i++) count[i]=0;

for (i=1;i<=n-1,i++)
  for (j=i+1;j<=n,j++)
          if (a[i]<=a[j])
                  count[j]+=1
          else
                  count[i]+=1

for (i=1;i<=n-1,i++)
  b[count[i]+1]=a[i];
```

六、 Distributive counting sort

M：排序資料的最大 key 值
N：資料總數

1. 根據資料的最大 key 值(找到最大 key 也要 O(N)時間吧)建立一個 O(M)的陣列
   (所以若 M>n^2 時，這個方法就會比其它方法都差)
2. 統計每個 key 值出現的次數，此步驟用掉 O(N)的時間
3. 由左往右加總(此步驟用了 O(M)的時間)
4. 逐一讀取 N 筆資料，根據其 key 值去找對應的 M 陣列中所指向的位置，就是它的所在位置，同時將 M 陣列中的值加 1，此步驟用了 O(N)的時間

▲舉例如下：
排序資料：5,7,9,4,7,5,5,1,9
根據最大 key 值建立的 M 陣列如下

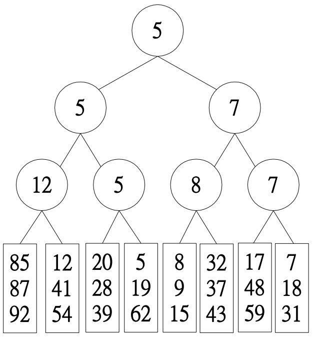
1	2	3	4	5	6	7	8	9
1	0	0	0	3	0	1	0	2

1	2	3	4	5	6	7	8	9
1	?	?	?	4	?	5	?	7

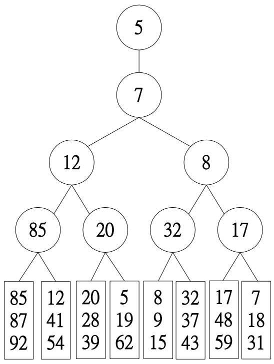
?是老師沒講，不知道要怎麼填
所以一讀到資料 5，就知道要放在位置 4，同時上表 5 的對應位置要加 1，下一個 5 再參照時，就會放到位置 5，以此類推
▲因此：適用在重覆的 key 值多，且 key 值不大的狀況

七、 Selection Trees

1. Winner's Tree



2. Loser's Tree



八、 Polyphase merge(Fibonacci merge)

(P51)，尋找資料的切割點，使外部排序可以用到最少的磁帶以及最快的排序，此方法是比較常考的一個

第十三章 搜尋法

一、 循序搜尋法(Sequential Search、Linear Search)
時間複雜度：O(n)
平均比較次數：(n+1)/2
資料可以未經排序

二、 二分搜尋法(Binary Search)
資料必須經過排序
時間複雜度：O(logn)
worst case 比較次數：$\lfloor \log_2 n \rfloor + 1$

三、 費氏搜尋法(Fibonacci Search)
資料必須經過排序
時間複雜度：O(logn)
▲利用費氏級數來找到中間項，因此只用到加減法，不像二分搜尋法要用到乘、除法

費氏級數	1	2	3		5			8					13
資料項	12	24	35	46	57	68	71	84	92	103	111	124	139

如上表，目前最大項 F_n 是第 13 項，F_{n-1} 是第 8 項，F_{n-2} 是第 5 項，只要用 F_n-F_{n-2} 就可以找到中間項是第 8 項

如果資料項數不是費氏級數，則在搜尋時，目標值先跟中間項比較，若<=中間項，則繼續搜尋，否則要做平移的搜尋方式(詳見 p7、p8)

四、 內插搜尋法(Interpolation Search)
平均分佈的資料：O(loglogn)
worst case：O(n)
Robust 的改良方法：O($(\log n)^2$)
中間項的搜尋公式：

$$mid = low + \frac{x - key[low]}{key[high] - key[low]} \times (high - low)$$

(真不想背呀!!)

五、 Hashing

1. Hashing 的基本觀念：
   ▲如何設計一個良好的 hasing function?
   1.計算簡單
   2.碰撞少
   3.資料平均分佈
   ▲hashing 通常應用在那些地方?
   1.拼字檢查器
   2.字典
   3.DBMS 的 Data Dictionary
   4.Loader、Assembler、Compiler 的 symbol table
   ▲hashing 的優點：
   1.資料的搜尋、貯存不需先經排序
   2.資料搜尋、貯存的時間通常在 O(1)等級
   3.保密性高
   4.可做資料壓縮

2. 基本名詞
   Bucket：一個 hash table 分成多個 buckets
   Slot：每個 bucket 可以存放多個 slots
   Synonym：若 keys X_1<>X_2，但是 $F(X_1)=F(X_2)$，則稱 X_1、X_2 互為 synonyms
   Collision：當存入一筆資料 X 時，若在 bucket F(X)中已有別的資料 X'，則稱為 collision
   Overflow：collision 發生時，若 bucket 中的 slots 全滿，使得資料無法存放入此 bucket，則稱為 overflow
   Loading factor：α=n/(bxs)
   Home value：key 值第一次經過 hashing function 計算後，預計放置的位置
   Perfect Hashing Function：不會發生 collision 的 hashing function
   Minimal Perfect Hashing Function：hashing table 的 size 剛好等於資料筆數，且都沒有發生碰撞

3. Hashing Function
   i. Mid-Square：將 key 平方後，最中央一段數字；例如取長度 r，則 buckets 必需是 2^r
   ii. Division
   iii. Folding：又分為 shift folding、folding at the boundaries
   iv. Multiplicative：trunc(Mxfrac(cxX))，M 為 buckets，0<c<1，X 是 key 值
   v. Digit Analysis

4. Overflow Handling
   i. Open Addressing
      Linear Probing
      Quadratic Probing
      Quadratic Residue
      Random Probing
   ii. Separate Chaining

5. Primary Clustering
   ▲Rehashing path 上連續的 buckets 被佔用後，會使平均 probing 次數增加的現象
   ▲兩個 home value 不同的 rehashing path 若重疊，表示容易發生 primary clustering

6. Secondary Clustering
   ▲不同的 key 值若 home value 相同，則會使用相同的 rehashing path 的現象

7. Double Hashing
   為了避免 Primary、Secondary Clustering 的現象，double hasing 的做法即使用另一個 hashing 函數用來計算 rehashing 中的增量

六、 Dynamic Hashing(p19、20)
分為有目錄及無目錄的兩種

第十四章 演算法

一、簡介：

Greedy Methods	Minimum Spanning Tree	Kruskal's Algorithm：$O(n^2)$或 O(eloge+elogn)
		Prim's Algorithm：$O(n^2)$
		Sollin's Algorithm：$O(n^2)$
	OS 的 SJF(Shortest Job First)	$O(n^2)$
	Loading problem	O(nlogn)
	Fractional Knapsack problem	O(nlogn)
	Convex Hull	O(nlogn)
	Single source all destination	Dijkstra's Algorithm：O(e+nlogn)
	Selection Sort	$O(n^2)$
	Huffman code	
Dynamic Programming	0/1 Knapsack problem；fk(c)是指 k 個物件，重量限制為 c $f_k(c)=\max\{f_{k-1}(c),f_{k-1}(c-w_k)+p_k\}$； $f_k(c)$表示 k 個東西，在 $\sum_{i=1}^{n} w_i x_i \le c$ 的條件下的最佳解 $f_k(c)=\max(\sum_{i=1}^{k} p_i x_i)$，時間複雜度為 O(nW)或 $O(2^n)$，其中 W 為重量限制	
	String Editing cost(i,j)=min{cost(i,j-1)+I(j),D(i)+cost(i-1,j),cost(i-1,j-1)+C(i,j)} 時間複雜度：O(mn)，m、n 分別為兩個字串的長度	
	Longest Common Subsequence(LCS) L(i,j)=L(i-1,j-1)+1 if $a_i=b_j$ L(i,j)=max(L(i-1,j),L(i,j-1)) if a_i<>b_j 時間複雜度：O(mn)，m、n 分別為兩個字串的長度	
	Longest Increasing Sub-sequence 時間複雜度：$O(n^2)$	
	組合公式 C(n,k)	O(nk)
	Matrix Product Chain，如果使用窮舉法，則 n 個矩陣會有 $\frac{1}{n+1}\times C_n^{2n}$ 種相乘方式，以下是使用 dynamic programming 的方式，時間複雜度為 $O(n^3)$ A[i,j]=0, i=j $A[i,j]=\min_{i\le k<j}\{A[i,k]+A[k+1,j]+d[i-1]\times d[j]\times d[k]\}$ 其中 i<j A[i,j]的意思是矩陣的第 i 個乘到第 j 個，最少的相乘次數 以 A_{4*2}、B_{2*3}、C_{3*5}，則 $d_0=4,d_1=2,d_1=2,d_2=3,d_2=3,d_3=5$	
	All pairs shortest path	Floyd-Warshall：$\theta(n^3)$
	OBST(Optimal Binary Search Tree) 原理是某一節點為根節點時會有最小成本 p_i：內部節點 i 的出現機率；q_i：外部節點 i 的出現機率 $initial$：$W_{ii}=q_j$; $C_{ii}=0$; $R_{ii}=0$ $W_{ij}=W_{i,j-1}+p_j+q_j$ $C_{ij}=\min\{C_{i,k-1}+C_{k,j}\}+W_{ij}$ 時間複雜度：$O(n^2)$	
	TSP(Traveling Salesperson Problem) $g(i,S)=\min_{j\in S}\{C_{ij}+g(j,S-\{j\})\}$ i 是起點，S 是必需經過 1 次的頂點集合 時間複雜度：$O(n^2 2^n)$	
Divide and conquer	Multiplication of Long Integers	$O(n^2)\to O(n^{\log_2 3})$
	Matrix multiplication	Strassen's：$O(n^3)\to O(n^{\log_2 7})$
	Finding the Majority	O(n)
	Binary Search	O(logn)
	Merge Sort	O(nlogn)
	Quick Sort	O(nlogn)
Backtracking	0/1 Knapsack problem	$\theta(2^n)$
	八皇后問題	
	Mazing problem	O(mn)，m 是迷宮的行數，n 是列數
	圖形的 DFS 搜尋	O(n+e)

	Sum of subset 從 n 個數列中，找出總合=M 的子集合 時間複雜度：$O(2^n)$	
	圖形著色	$O(m^n)$，m 是顏色數，n 是圖形的區塊數
	Hamiltonian cycle	$O(d^n)$，d 是 node 的最大 degree
NP Theory		
Branch and Bound		

二、Bin Packing(裝箱問題)

箱子個數、容量固定，如何裝最多的東西

三、Loading Problem

四、Fractional Knapsack problem

五、Convex Hull(凸多邊形包圍)

△問題：平面上 n 個點座標，從此 n 個點中找出一個凸多邊形，將其餘的點全部包含在內

△解法：

1. 找出 P_0 為 y 座標最小的點。O(n)
2. 將 P_0 分別與 P_i 連線，計算其與 x 軸的夾角，並由小到大排序。O(nlogn)
3. 計算

```
push(P0),push(P1),push(P2)
for i=3 to n
{
    while(目前 stack 頂端兩點的連線與新加入的點的連線形成凹角)
        pop();
    push(Pi)
}
```

最後留在 stack 中的頂點集合就是所要的 convex hull

此步驟 O(n)，因為 pop 次數<=push 次數<=n

六、長整數的相乘(Multiplication of Long Integers)

△一般的作法

$$xy=(x_l 10^{n/2}+x_r)(y_l 10^{n/2}+y_r)$$
$$=x_l y_l 10^n+(x_l y_r+x_r y_l)10^{n/2}+x_r y_r$$

例如：x=1234，則 $x_l=12$，$x_r=34$；$x=x_l*10^2+x_r$

其時間函數為：$T(n)=4T(n/2)+cn\to O(n^2)$

亦即要分別做 $x_l y_l$、$x_l y_r$、$x_r y_l$、$x_r y_r$ 共四次的分別相乘

△divide and conquer 的作法，把上式中的 $(x_l y_r+x_r y_l)$ 簡化

如右 $x_l y_r+x_r y_l=(x_l+x_r)(y_l+y_r)-x_l y_l-x_r y_r$

因此只要算出 $x_l y_l$、$x_r y_r$、$(x_l+x_r)(y_l+y_r)$三項資料重複利用

其時間函數為：$T(n)=3T(n/2)+cn\to O(n^{\log_2 3})$

七、Matrix Multiplication

使用 divide and conquer 的方法 n*n 的 A、B 矩陣相乘，可化為

A_{11}	A_{12}	×	B_{11}	B_{12}	=	C_{11}	C_{12}
A_{21}	A_{22}		B_{21}	B_{22}		C_{21}	C_{22}

$C_{11}=A_{11}\times B_{11}+A_{12}\times B_{21}$

…….

時間函數為：$T(n)=8T(n/2)+cn^2\to O(n^3)$

沒有比原來的 $O(n^3)$的演算法好

```
for i=1 to n
    for j=1 to n
        for k=1 to n
            C[i][j]=C[I][j]+A[i][k]×B[k][j]
```

程式實例：

```
void main()
{
    int a[2][3]={{1,2,3},{4,5,6}};
    int b[3][2]={{1,4},{2,5},{3,6}};
    int c[2][2]={{0,0},{0,0}};
    int i,j,k;
    for (i=0;i<2;i++)
        for (j=0;j<2;j++)
            for (k=0;k<3;k++)
                c[i][j]+=a[i][k]*b[k][j];
}
```

▲Strassen's Matrix Multiplication(式子在講義 p30)
其演算法時間函數為：$T(n)=7T(n/2)+cn^2 \rightarrow O(n^{\log_2 7})$

八、 Finding the Majority
由 n 個數目之中找出是否有任一元素超過一半(即 n/2 個)以上，此一元素則稱為 majority element
△方法一：使用 select kth smallest element 的方法：O(n)
△方法二：配對消去法：O(n)

九、 0/1 Knapsack problem
△與 Fractional Knapsack problem 不同點在於此問題中的物件必需全拿或全不拿，不能只拿物件的一部份
△不能用 Greedy Method；用 weight、profit、density 都不保證最佳解
▲要用 dynamic programming，計算原則在上頁表格中

十、 String Editing

十一、 Longest Common Subsequence(LCS)

十二、 Longest Increasing Sub-sequence
找出 n 個相異整數的序列中，最長的遞增數列
△方法一：
1. 把原始數列 S 排序成 T。O(nlogn)
2. 用 S 來跟 T 進行 LCS 運算，所得的結果即為最長遞增數列。O(mn)，其中 m=n→$O(n^2)$

時間複雜度：$O(n^2)$

△方法二：(沒弄懂)
時間複雜度：O(nlogn)

十三、 Sum of subset(使用 Backtracking)
△從 n 個數列中，找出總合=M 的子集合；時間複雜度：$O(2^n)$
△不能用 greedy method、divide and conquer、dynamic programming
△方法一：Exhaustive Search：$O(2^n)$
令數列為 w[i], i=1~n
x[i]=0 當 w[i]不選取
x[i]=1 當 w[i]被選取

```
void subset(int x[],int i,int n,int sum)
{
        if (i>n)
                {if (sum==M);}//output
        else
        {
                x[i]=0;
                subset(x,i+1,n,sum);
                x[i]=1;
                subset(x,i+1,n,sum+w[i]);
        }
}
```

此演算法跟集合 S 的所以可能子集合的意義一樣

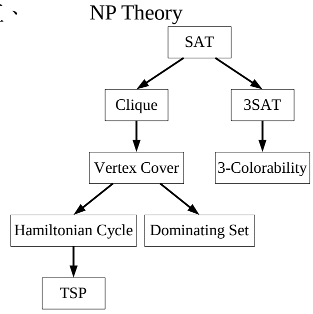
△方法二：Heuristic Functions：$O(2^n)$
時間複雜度雖然跟方法一同等級，但實作上會比較快，因為有些分枝確定不可行了，就不用再往下運算

```
void subset(int x[],int i,int n,int sum,int remain)
{
        if (sum= =M);//ouput();
        else if (sum+remain>=M && sum+w[i+1]<=M)
        {
                x[i]=0;
                subset(x,i+1,n,sum,remain);
                x[i]=1;
                subset(x,i+1,n,sum+w[i],remain-w[i]);
        }
}
```

別小看這麼一點條件，我用{1,2,3,4,5}，找總和=7 的子集合，方法一跑了 63 次，方法二只有 7 次!!

十四、 Hamiltonian cycle/path(使用 Backtracking)
△在 Graph G 中含有 n 個 vertices 找出一個 cycle 遊歷所有 vertices 各一次，回到起點
時間複雜度：假設每個 node 的 degree<=d，則總執行時間為
$1+d^1+d^2+\ldots\ldots\ldots\ldots+d^n=(1-d^n)/(1-d) \rightarrow O(d^n)$

十五、 NP Theory



Linear time：線性時間：一個問題的處理時間，與問題的大小無關，亦即其時間複雜度的函數中，沒有 n 值，例如：O(1)；此即為線性時間(make sure 一下!?)
Polynomial time：多項式時間：例如 $O(n^3)$就是多項式時間；$O(n^{100})$也算是多項式時間，而 O(nlogn)雖然不是「多項式」，但由於 O(nlogn)可以找到一個多項式時間如 $O(n^2)$為其上限，因此 O(nlogn)也屬於 class P
Exponential time：指數時間：例如 $O(2^n)$就是指數時間
Deterministic algorithm：確定性演算法：亦即演算的過程中，每一個步驟都只有唯一的選擇
Non-deterministic algorithm：不確定性演算法：亦即演算的過程中，每一個步驟可能有多個選擇
Turing Machine：是一個抽象的計算機
Decision Problem：決策問題：輸入一組資料後，輸出的結果為 yes 或 no
Class P：可以在 polynominal time 內處理的問題，歸類為 P
Class NP：可以用 non-deterministic algorithm 找到可能的解，並且在 polynominal time 內驗證此解是否正確的問題，歸類為 NP
NP Hard：若在 NP 中的每個問題可以在 polynominal time 內 reduce 成 problem X，則稱 problem X 為 NP Hard Problem
NP Complete：若一個問題是 NP Problem 且又是 NP Hard Problem，則此問題為 NP Complete Problem
Cook's Theorem：SAT Problem 為 NP Complete Problem
Lemma：若 L_1 可以多項式時間轉化成 L_2，且 L_1 為一 NP Complete Problem，而 $L_2 \in \mathcal{NP}$，則 L_2 亦為一 NP Complete Problem
SAT：可滿足性問題主要在探討 "給予某一特別型式的布氏式(boolean expression) e，對 e 中之所有變數，是否存在一種真假值的指定方式(truth assignment)，使得整個 e 的值為真"
CNF：連結正規式(conjunctive normal form)。若 $t_1, t_2, \ldots, t_n$ 皆為子句(clause)，(其中 $t_1, t_2, \ldots, t_n$ 每個變數稱之為 literal)，則型式 $t_1 \wedge t_2 \wedge \ldots \wedge t_n$ 稱之為一 CNF。
Clique problem：給予任一圖 G 及任一整數 k，G 中是否存在一有 k 個節點的完整子圖？
△證明一：若 G 為任一給予之圖，我們可以不定性的方式選擇 G 中的任 k 個節點，檢查其中任二節點是否相鄰，以此決定該 k 節點所構成的子圖是否為一完整圖。
△證明二：證明可滿足性問題可以多項式時間轉化成完全子圖問題??

TSP：可以由 Hamiltonian Cycle 證明其為 NP-Complete，方法如下：建立一個圖形 G，其頂點跟 TSP 圖形 T 中的頂點皆相同，將 G 建立成 complete graph，其中若任一 edge 也存在 T 中，則其成本為 1，否則成本為 2，於是由 Hamiltonian Cycle 轉換過來的決策問題變成：能否在 G 中找到一個 Hamiltonian Cycle，其路徑成本恰好等於|V|(亦即頂點個數，表示所走的路徑就是 TSP 中的路徑)。由此得證 TSP 為 NP Complete。

十六、 Branch and bound