CH7、Sort & Search

排序與搜尋

目錄：

Linear Search(線性)

作法：由頭到尾依序對 Data 一筆筆進行比對、並搜尋，可分為：

一、Non-Sential Linear Search

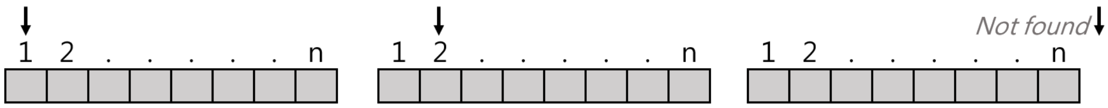
二、Sential Linear Search

一、Non-sential Linear Search

Def：由頭開始對 k(*欲找值*)搜尋，if found, return index(*位置*)；Not found, return 0

程式：

```
void non-sential(F, n, k)
//F 為 Data Array；n 為 Data size；k 為欲找值
{
        int i=1;
        while(i<=n)
        {
                if(F[i]==k)
                        return i;
                else
                        i=i+1;        //往下一格
        }
        return 0;
}
```



分析：

1. Time Complexity：O(n)，平均=(1+2+…+n)/n = (n+1)/2 = O(n)
2. 資料無需做任何的前置作業(unsorted is ok)
3. 支援 Sequential(Linked List)或 Random Acces(Array)之結構，皆可進行
4. 實作容易

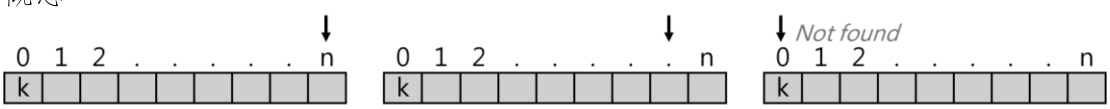
二、Sential Linear Search(守門員)

Def：

1. 第 0 格放置欲找值(k)
2. 由後往前找 k，找到則 return index　=0, not found

   $\neq$0, found

概念：



程式：

```
void sential(F, n, k)
//F 為 Data Array；n 為 Data size；k 為欲找值
{
	F[0]=k;
	int i=n;
	while(F[i]!=k)
	{
		i=i+1;
	}
	return i;
}
```

分析：

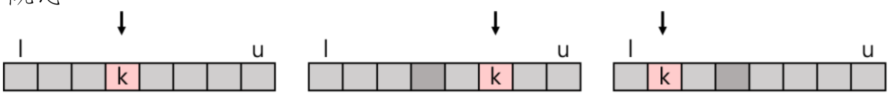
1. 效率較 Non-sential 好，因為少一個判別式
2. 多花一格的空間*(以空間換取時間)*
3. Time Complexity=O(n)

Binary Search(二分搜尋)

前置作業：

1. Data 需事先排序*(由小到大 or 由大到小)*
2. 需 Random Access 支援*(Array 可以，Linked List 較不適合)*

概念：



令欲找值：k，則*(以由小到大為例)*：

1. F[i]==k，則 return i
2. F[mid]>k，用 Binary Search 找左半邊
3. F[mid]<k，用 Binary Search 找右半邊

程式*(Recursive)*：

```
int BS(int F[], int k, int l, int u)
{
	if(l>u)
		return -1;	//not found
	else
	{
		int mid=(l+u)/2;
		if(f[mid]==k)	return mid;
		else if(F[mid]>k)
			return BS(F, k, l, mid-1);
		else return BS(F, k mid+1, u);
	}
}
```

程式*(採 iterative 改寫)*：

```
int BS(int F[], int k, int l, int u)

{
        int l=1, u=n;
        while(l<=u)
        {
                int m=(l+u)/2;
                if(F[m]==k)        return m;
                else if(F[m]>k)    u=m-1;
                else    l=mid+1;
        }
        return -1     //Not found
}
```

分析：以 Recursive：

T(n) = T(n/2) +1 => O(log n)

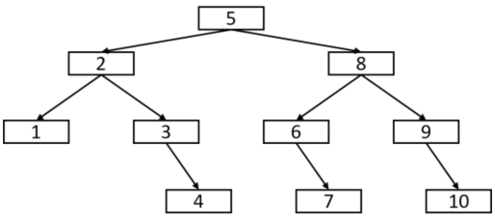
何時 Linear Search 比 Binary Search 更適合

1. Binary Search 需先做 Sorting => O(n log n)、找則花費 O(log n)，若只是『一次性』搜尋，此時 Linear Search 更為適合*(因為 O(n))*
2. 當結構採用 Linked Link 時，Linear 更適合
3. 若 Data 會動態增減時，Linear Search 更適合

Binary Search 的 Decision Tree(決策樹)

說明：依據 Binary Search 在 Search 的先後，建立出的樹狀結構，由此可知 Search 一鍵值的過程

例：



例：有 10 筆 Data：30, 10, 15, 20, 90, 80, 65, 70, 75, 45，以 Binary Search 找 45 需比幾次？

1. *Sorting：10, 15, 20, 30, 45, 65, 70, 75, 80, 90 = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10*
2. *建立 Index*
3. *建立 Decision Tree(45 在由小到大排序中，是第 5 號；63 是第 6 號)*
4. *由上可知，45 比 1 次、65 比 3 次(Search Path：45, 75, 65)*

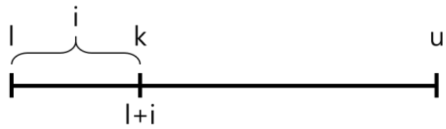
Note：n 筆 Data 採 Binary Search，最多比 log(n+1)次、最少比 1 次

插補搜尋(Interpolation Search)

Def：

1. 前置作業同 Binary Search
2. 較接近人類的搜尋行為

概念：



i = (k-F[l])/(F[u]-F[l]) * (u-l+1)

比 F[l+i]與 k：

1. F[l+i]==k return l+I;
2. F[l+i] > k 找左邊
3. F[l+i] < k 找右邊

程式：

```
int IPS(int F[], int k, int l, int u)
{
        if(l>u)
                return -1;            //not found
        else
        {
                int i=(k-F[l])/(F[u]-F[l])*(u-l+1);
                if(f[mid]==k)       return l+i;
                else if(F[mid]>k)
                        return IPS(F, k, l, l+i+1);
                else return IPS(F, k l+i+1, u);
        }
}
```

分析：

其效能取決於 Data 分佈情況：在 Uniform Distribution*(平均分佈)*之下，效能最佳*(比 Binary Search 更快)*；In worst case，退化成 O(n)

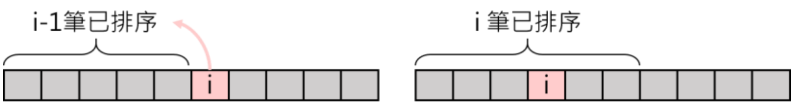
例：1, 2, 3, …, 1000 之類的

Sorting
依存放地可分為：
1. 內部排序：指資料於 Memory 中
2. 外部排序：當 Data 量過大，存於 Memory 之外進行(例：Disk)

依排序過程，相同 Data 出現順序是否會改變，分為：
1. Stable Sorting：
   …, 3, …, 3+, …
   …, 3, …, 3+, …
2. Unstable Sorting：
   …, 3, …, 3+, …
   …, 3+, … ,3, …

依演算法分為：
1. 初等排序：O(n^2)、採 Comparsion-Based
   (1) Insertion Sort
   (2) Selection Sort
   (3) Bubble Sort
   (4) Shell Sort
2. 高等排序：O(n log n) 、採 Comparsion-Based
   (1) Quick Sort：[DS 版]與[演算法版]
   (2) Merge Sort：Selection Tree
   (3) Heap Sort
3. 線性排序：O(n)
   (1) Radix Sort：Radix 與 Bucket
   (2) Counting Sort

Insertion Sort(插入排序)
作法：將第 i 筆 Data 插入到前面(i-1)筆已排好的記錄串列中，使之成為 i 筆已排好序的串列
概念：



例：Data：26, 5, 33, 17, 2，採 Insertion Sort 問過程？

Pass	*Data*
Initial	*26, 5, 33, 17, 2*
1	*5, 26, 33, 17, 2*
2	*5, 26, 33, 17, 2*
3	*5, 17, 26, 33, 2*
4	*2, 5, 17, 26, 33*

程式：

```
void insert(int key, int *list, int i)      //副程式
{
        while(key<list[i])
        {
                list[i+1]=list[i];
                i=i-1
        }
        list[i+1]=key;
}

void insertSort(int *list, int n)           //主程式
{
        for(int j=1;j<n;j++)                //n-1 回合
                insert(list[j], list, j-1); //將第 j 筆插到前(j-1)筆已排好序的 Data 中
}
```

分析：

1. Time Complexity：

    (1) Best Case：Data 已由小到大排序

    例：1, 2, 3, 4

Pass	*Data*
Initial	*1, 2, 3, 4*
1	*1, 2, 3, 4*
2	*1, 2, 3, 4*
3	*1, 2, 3, 4*

    [法一]量化比較次數：$\sum 1 = O(n)$

    [法二]採 Recursive 分析

    $T(n) = T(n-1) + 1 \Rightarrow O(n)$

    [法三]採 Inversion(x)之概念

    Def：出現在 x 左邊的字串中，大於 x 之數字個數

    例：

Input	*13*	*5*	*8*	*2*	*15*	*14*	*6*
Inservsion	*0*	*1*	*1*	*3*	*0*	*1*	*4*

    作用：

    當$\sum$ inversion(x) = 0，代表排序已完成

    排序的 Effort，即將$\sum$ inversion(x) = 0 過程

    只能知道 No Swap Effort，但是無法確定是否為 Best Case

    (2) Worst Case：Data 由大到小(反序)

    例：Data：4, 3, 2, 1

Initial	*4, 3, 2, 1*
1	*3, 4, 2, 1*
2	*2, 3, 4, 1*
3	*1, 2, 3, 4*

以[法一]表示
以[法二]計算時間複雜度：$T(n) = T(n-1) + n-1 \Rightarrow T(n^2)$

(3) Average Case：$O(n^2)$
$T(n) = T(n-1) + n/2 \Rightarrow O(n^2)$

2. Space Complexity：O(1)，只需常數個暫存數即可
3. 為 Stable Sorting 演算法
   例：有 Data：
   未排序：…, 5, …, 5+, …
   已排序：…, 5, 5+, …

[補充] Insertion Sort 之變形
1. Binary Insertion Sort
2. Linear Insertion Sort

分析：傳統 Insertion Sort 中，可視為(n-1)回合，各回合有 2 個主要工作
1. 找出 r 的正確插入位置
2. Data Movement

即：

動作	用的 Data Structure 或機制	Time
1	用 Linear Search	O(n)
2	使用 Array 存 Data => Insert 後面皆需移動	O(n)

改進：
採 Binary Insertion Sort：

動作	用的 Data Structure 或機制	Time
1	用 **Binary Search**	O(log n)
2	使用 Array 存 Data => Insert 後面皆需移動	O(n)

採 Linear Insertion Sort：

動作	用的 Data Structure 或機制	Time
1	用 Binary Search	O(log n)
2	使用 **Linked List** 存 Data => **方便**	O(n)

小結：
Binary Insertion Sort 改善了 1.，但 2.不變
Linear Insertion Sort 改善了 2.，但 1.不變
結論：依然為 $O(n^2)$，但效能有所提升

Selection Sort(選擇排序)

作法：從第 i 到 n 筆 Data 中，挑最小值，與第 i 筆元素做 swap=>反覆作(n-1)回合

例：Data：2, 3, 5, 26, 77, 13，採 Selection Sort，過程為何？

Pass	*Data*
Initial	*23, 5, 26, 77, 13*
1	*5, 23, 26, 77, 13*
2	*5, 13, 26, 77, 23*
3	*5, 13, 23, 77, 26*
4	*5, 13, 23, 26, 77*

程式：

```
void selectionSort(int *list, int n)
{
        for(int i=0;i<n-1;i++)                          //作 n-1 回合
        {
                m=i;                                    //記錄此回合最小值
                for(int j=i+1;j<n;j++)                  //求此回合之最小值
                        if(list[j]<list[m])  m=j;
                if(i!=m) swap(list[i], list[m]);        //判別 i 跟 m，若不同才 swap(可省略判別式)
        }
}
```

Note：以 Swap 而言，每回合最多做 1 次，故(n-1)回合，只需做(n-1)次：在 Data 量大時適用，因為 Swap 次數很少

分析：

1. Time Complexity：$O(n^2)$*(不論 Best, Worst, Average Case)*
2. Space Complexity：O(1)
3. 為 Unstable Sorting 演算法
   例：
   未排序：3, 5, …, 3+, …, 1
   已排序：1, 5, …, 3+, …, 3

Bubble Sort(氣泡排序)

作法：將元素兩兩互相比較，若前者>後者則 Swap，在每回合中，將最大元素上升到最高格(由左而右處理，作 n-1 回合)

例：26, 5, 33, 17, 2，採 Bubble Sort，過程為何？

Pass	*Data*
Initial	*26, 5, 33, 17, 2*
1	*5, 26, 17, 2, 33*
2	*5, 17, 2, 26, 33*
3	*5, 2, 17, 26, 33*
4	*2, 5, 17, 26, 33*

[版本二]：將元素兩兩互相比較，若前者>後者則 Swap，在每回合中，將最小元素下降到最低格(由右而左處理，作 n-1 回合)

例：26, 5, 33, 17, 2，採 Bubble Sort，過程為何？

Pass	*Data*
Initial	*26, 5, 33, 17, 2*
1	***2,*** *26, 5, 33, 17*
2	***2, 5,*** *26, 17, 33*
3	***2, 5, 17,*** *26, 33*
4	***2, 5, 17, 26,*** *33*

程式：

```
void bubbleSort(int *list, int n)
{
    for(int i=n-1;i>=1;i--)                //作 n-1 回合
    {
        bool flag=false;                   //判別此回合有無 Swap
        for(int j=0;j<i;j++)               //兩兩相比
        {
            if(list[j]>list[j+1])
            {
                swap(list[j+1],list[j]);
                flag=true;
            }
        }
        if(flag==false)     break;
    }
}
//若拿到 Bool 值的 flag 仍可運作，但效能會下降
```

分析：

1. Time Complexity：
   (1) Best Case：Data 由小到大已排序 => $O(n)$
   例：Data：1, 2, 3, 4

Pass	*Data*
Initial	*1, 2, 3, 4*

   $T(n) = n-1 + T(0) => O(n)$
   (2) Worst Case：Data 由大到小
   例：Data：4, 3, 2, 1

Pass	*Data*
Initial	*1, 2, 3, 4*
1	*3, 2, 1,* ***4***
2	*2, 1,* ***3, 4***
3	*1,* ***2, 3, 4***

   $T(n) = T(n-1) + n-1 => O(n^2)$
   (3) Average Case：$O(n^2)$

2. Space Complexity：O(1)，因為只需常數個暫存變數
3. 為 Stable Sorting 演算法

   例：有 Data：

   未排序：…, 5, 5+, …*(因為 5 不大於 5+，故 no swap)*

   已排序：…, 5, 5+, …

例：有下列 Data 作 Sort：13, 17, 2, 5, 8, 55, 2+

1. 用 Insertion Sort，Pass 3 後之結果為何？
2. 用 Selection Sort，Pass 2 後之結果為何？
3. 用 Bubble Sort，Pass 3 後之結果為何？

1. *2, 5, 13, 17, 8, 55, 2+*
2. *2, 2+, 13, 5, 8, 55, 17*
3. *2, 5, 8, 2+, 13, 17, 55*

Shell Sort(謝爾排序)

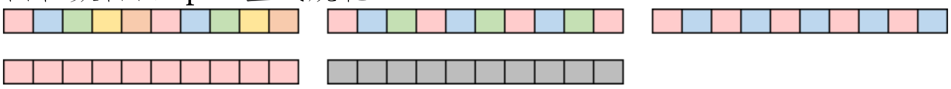
[原始版本]概念：

比較第[i]與第[i+span]筆資料，若前者大於後者則 Swap

Each Pass，i 值由 i to (n-span)

每回合之上述工作，需持續進行，直到 No Swap 發生，才進入下一回合

回合數目由 Span 型式規範



1. 內定型：$[n/2^k]$ or $[n]/2^k$ 型

   例=10

Pass 1	[10/2] = 5
Pass 2	[5/2] = 3
Pass 3	[3/2] = 2
Pass 4	[2/2] = 1

2. $n=2^k-1$ 型：ex：Span 依序為：15, 7, 3, 1
3. 2^i, 3^i 型：
4. 自訂型，但最後 span 必為 1

[另一版本]概念：有利於計算型使用

若此回合之 Span 值為 k，則代表有 k 條 sublists 要進行 insertion Sort

各 sublist 之資料量約[n/k]

例：9, 8, 1, 4, 10, 6, 2, 3, 7, 5，採 Shell Sort，Span 用[n/k]

9	8	1	4	10	6	2	3	7	5

6	2	1	4	5	9	8	3	7	10

4	2	1	6	3	7	8	5	9	10

1	2	3	6	4	5	8	7	9	10

1	2	3	4	5	6	7	8	9	10

程式(較少考)：

```
void shellSort(int list[], int n)
{
        span=[n/2];
        while(span>=1) do
        {
                f=0;  //有無 swap，0 為無，1 為有
                repeat
                {
                        for i=1 to (n-span) do
                                if(list[i]>list[i+span])
                                {
                                        swap(list[i], list[i+span]);
                                        f=1;
                                }
                }
                until(f=0);
        span=span/2;
        }
}
```

分析：

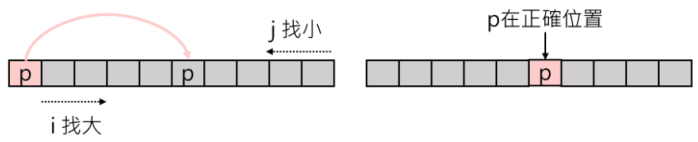
1. Time Complexity
   (1) Worst Case：$O(n^2)$
   (2) Average Case：$O(n^2)$
   (3) Best Case：$O(n^{3/2})$ or $O(n^{5/4})$ => 尚無定論，取決於 Span 型式

2. Space Complexity：O(1)
3. 為 Unstable Sorting 演算法
   例：
   未排序：5, 5+, 1, 8
   已排序：1, 5+, 5, 8

Quick Sort(**快速排序**)

特性：平均下最快的排序演算法(基於 Comparsion-Based)
採用”Divide-and-Conquer”分割並克服(Master Method 亦是)

概念：



將 pivot 放到最適當的位置

作法：

1. 每一回合拿一值當 pivot(control key，一般拿第一筆；演算法版拿最後一筆)
2. 使用 2 變數
   (1) i：從第 1 筆開始找比 control key 大的
   j：從第 n 筆開始找比 control key 小的
   (2) if (i<j), swap(data[i], data[j])
   otherwise, swap(ck, data[j])
   (3) 對 ck 的左、右兩邊做 Quick Sort

例：[26, 5, 37, 1, 61, 11, 59, 15, 48, 19]作 Quick Sort 過程為何？

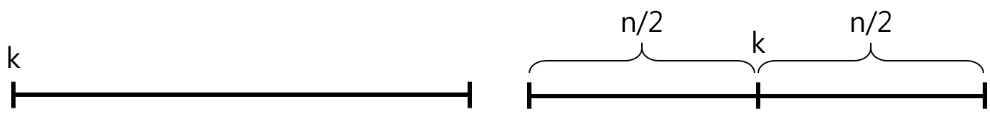
Pass	*Data*
Initial	*26, 5, 37, 1, 61, 11, 59, 15, 48, 19*
1	*11, 5, 14, 1, 15, 26, 59, 61, 48, 37*
2	*1, 5, 11, 19, 15, 26, 59, 61, 48, 31*
3	*1, 5, 11, 19, 15, 26, 59, 61, 48, 37*
4	*1, 5, 11, 15, 19, 26, 59, 61, 48, 37*
5	*1, 5, 11, 15, 19, 26, 48, 37, 59, 61*
6	*1, 5, 11, 15, 19, 26, 48, 37, 59, 61*
7	*1, 5, 11, 15, 19, 26, 37, 48, 59, 61*

程式：

```
void QS(int *list, int l, int r)
//Data Array；left(下限)；righ(上限)
{
        if(l<r)
        {
                int i=l, j=r+1, pivot=list[l];            //取第 1 筆
                do{
                        do i++; while(list[i]<pivot);
                        do j++; while(list[j]>pivot);
                        if(i<j) swap(list[i], list[j]);
                }while(i<j)
        swap(list[l], list[j]);
        QS(list, l, j-1);                                 //對左邊做 Quick Sort
        QS(list, j+1, r);                                 //對右邊做 Quick Sort
        }
}
```

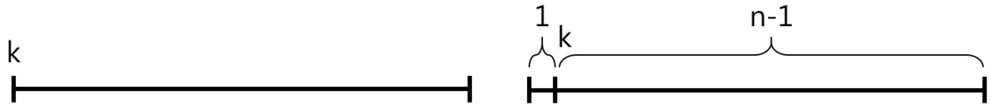
分析：

1. Time Complexity
   (1) Best Case：將左、右 2 邊對半切割



$T(n) = T(n/2) + T(n/2) + cn \Rightarrow O(\log n)$

(2) Worst Case：當 Data 有 Sorting(不論正序或反序)



$T(n) = T(n-1) + T(0) + cn \Rightarrow O(n^2)$

Quick Sort Worst Case 的改善方法
[法一]採用”Middle of Three”(最常見)
目的：避免 pivot key 為 Max 或 Min
[法二]：演算法版”Randomized-Quick Sort”
使用亂數隨機挑一值當 pk(pivot key) => 可降低，但最差仍為 $O(n^2)$
[法三]：演算法版”Median-of-medians”
中間的中間值

(3) Average Case：O(n log n)

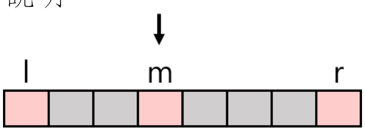
2. Space Complexity
   (1) Best Case：O(log n)
   (2) Worst Case：O(n)

3. 為 Unstable Sorting 演算法
   例：
   未排序：3, 5, …, 5+, 1
   已排序：3, 1, …, 5+, 5

Middle of Three

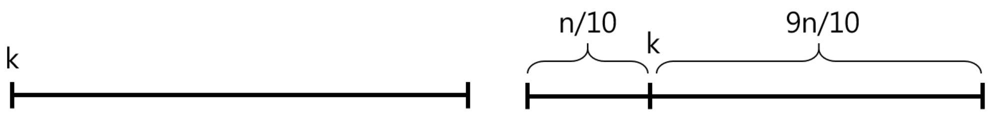
(避免 Quick Sort 之 Worst Case 為 $O(n^2)$)

說明：



1. m=(l+u)/2
2. 比較 A[l], A[m], A[r]這 3 筆，找出中間值
3. 將中間與 A[l]交換
4. A[l]無 pivot key

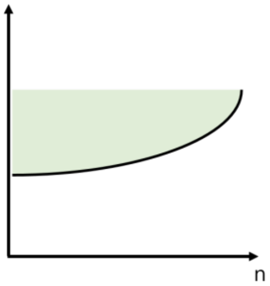
如此在”不失一般性”之下，若 Quick Sort 之後為：



$T(n) = T(n/10) + T(9n/10) + cn$
採”Recursive Tree”解 => 各層成本相同，故 Time Complexity 為 O(log n)
故此時在 Worst Case 之下，亦可確保 Time Complexity 為 O(n log n)

排序最快可達到多快？(Comparsion-Base)

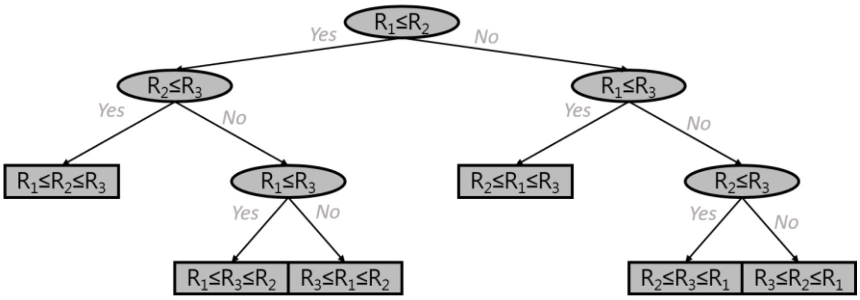
採 Comparion-Based 之下，最快可達 Ω (n log n)



Note：非 Comparsion-Base 不在此限

概念：若有 3 筆 Data：R_1, R_2, R_3

Decision Tree Sorting 描述比較過程



Note：

1. Non-leaf 表”比較”Node
   Leaf 表個排序結果
2. N 個 Data 會有 n!種可能的結果(Leaf Node 數)
3. 比較次數會與樹高”成正比”

[證明]：Sorting n 筆 Records，會有 n!種可能結果，以 Sorting 之 Decision Tree 而言，即代表會有 n!個 Leaf，又 Decision Tree 是 Binary Tree，以高度會≥ $[\log_2(n!)]+1$ *(定理 1)*

⇨ 比較次數≥ [log2(n!)] ≥ log2(n/2) = (n/2)log(n/2) => Ω (n log n)

從此可知，其最快的排序時間為 Ω (n log n)

[演算法版]
程式：

```
QuickSort(A, p, r)                      //主程式
{
        if p<r then
        {
                q=Partition(A, p, r);
                QuickSort(A, p, q-1);   //左邊
                QuickSort(A, q+1, r);   //右邊
        }
}

Parition(A, p, r)                       //副程式
{
        x=A[r];                         //x 為 pk，拿最後一筆
        i=p-1;
        for j=p to (r-1) do
        {
                if(A[j]<=x) then
                {
                        i=i+1;
                        swap(A[i], A[j]);
                }
        }
        swap(A[i+1], A[r]);
        return i+1;
}
```

例 1：Data：2, 8, 7, 1, 3, 5, 6, 4 之過程？

Initial	*2, 8, 7, 1, 3, 5, 6, 4*
1	*2, 1, 3,* **4**, *7, 5, 6, 8*
2	…

例 2：Data：9, 8, 1, 3, 5, 6, 7, 4，採 Parition 方式做 Quick Sort 之 Pass 1 Output 為何？

Initial	*9, 8, 1, 3, 5, 6, 7, 4*
1	*1, 3,* **4**, *8, 5, 6, 7, 9*

例 3：Data：5, 5, 5, 5, 5+，採 Parition 方式做 Quick Sort 之 Pass 1 Output 為何？

Initial	*5, 5, 5, 5, 5+*
1	*5, 5, 5, 5,* **5+**

Note：因此當 Data 都相同時，為 Worst Case，但此情況在 DS 版為 Best Case

1. Time Complexity：
   (1) Best Case：O(n log n)
   (2) Average Case：O(n log n)

(3) Worst Case：$O(n^2)$
   a. 由小到大
   b. 由大到小
   c. Data 皆相同
      (加速方式：採 Medium of mediums 來改善：O(n log n))

2. Space Complexity：
   (1) Best Case：O(log n)
   (2) Worst Case：O(n)

3. 為 Unstable Sorting 演算法

Merge Sort(合併排序)

Def：是 External Sort(外部排序)常用方法之一

分為 2 版本：

一、Iterative

二、Recursive

術語：

1. Run：排序好的片段資料
2. Run 長度：指 Run 當中的 Data 個數

一、Iterative Merge Sort(以 2-way Merge 為例)

例：26, 3, 5, 17, 10, 2, 8, 4, 9, 12

Pass	
Initial	26, 3, 5, 17, 10, 2, 8, 4, 9, 12
1	[3, 26], [5, 17], [2, 10], [4, 8], [9, 12]
2	[3, 5, 17, 26], [2, 4, 8, 10], [9, 12]
3	[2, 3, 4, 5, 8, 10, 17, 26], [9, 12]
4	[2, 3, 4, 5, 8, 9, 10, 12, 17, 26]

例 1：3, 26, 5, 77

Pass	*Data*
Initial	*3, 26, 5, 77*
1	*[3, 26], [5, 77]*
2	*[3, 5, 26, 77]*

最多比 3 次 (n-1 次)

例 2：[3, 26], [40, 50, 60, 70]

Pass	*Data*
Initial	*[3, 26], [40, 50, 60, 70]*
1	*[3, 26, 40, 50, 60, 70]*

小結：Run1 長度=m，Run2 長度=n
則：
最多比 m+n 次=>n 次
最少比 m or n 次=> n/2 次

如何合併 Run1, Run2 成為一個 Run？
令 p 指向 Run1, q 指向 Run2：
程式：

```
while(Run1 and Run2 尚未 scan 完)
{
        if(p.data<=q.data)
        {
                p.data output to NewRun;
                p=p+1;
        }
        else
        {
                q.data output NewRun;
                q=q+1;
        }
}
while(Run1 尚未 scan 完)
{
        copy Run1 剩下的 data 到 NewRun;
}
while(Run2 尚未 scan 完)
{
        copy Run2 剩下的 data 到 NewRun;
}
```

二、Recursive Merge Sort
採"Divide-and-Conquer"
步驟：
1. 分割成 2 等份
2. 左、右半部各自 Merge Sort
3. 合併左、右 2 個 Run 成 NewRun

例：採 Recursive Merge Sort：26, 3, 5, 17, 10, 2, 8, 4, 9, 12？

Pass	*Data*
Initial	*26, 3, 5, 17, 10, 2, 8, 4, 9, 12*
1	*[3, 26], 5, [10, 17], [2, 8], 4, [9, 12]*
2	*[3, 5, 26], [10, 17], [2, 4, 8], [9, 12]*
3	*]3, 5, 10, 17, 26], [2, 4, 8, 9, 12]*
4	*[2, 3, 4, 5, 8, 9, 10, 12, 17, 26]*

程式：

```
rMergeSort(x, l, u, p)
//排 x[l]~x[u]，成為 run: p
{
	if(l>=u) p=l;
	else
	{
		m=(l+u)/2;
		rMergeSort(x, l, m, q);		//左
		rMergeSort(x, m+1, u, r);	//右
		ListMerge(x, q, r, p);		//將 runq 與 runr Merge 到 runp
	}
}
```

思考：

T(n) = T(n/2) + T(n/2) + cn => O(n log n)

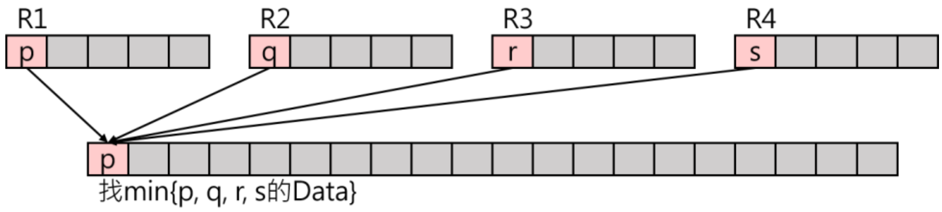
分析：

1. Time Complexity：O(n log n)
2. Space Complexity：O(n)，額外空間需求來自暫存每一回合之合併結果，因此所需 Space 跟 Input Data 相同
3. 為 Stable Sorting 演算法
   例：
   未排序：3, 2, 5, 5+
   已排序：[2, 3], [5, 5+]

Selection Tree(選擇樹, k>2)

目的：協助 k-way Merge(一次合併 k 個 Run 成為一個 Run)之進行，加速從 k 個 Runs 找出 min 值，輸出到 NewRun

討論：當沒採用 Selection Tree 時，ex：4-way Merge：



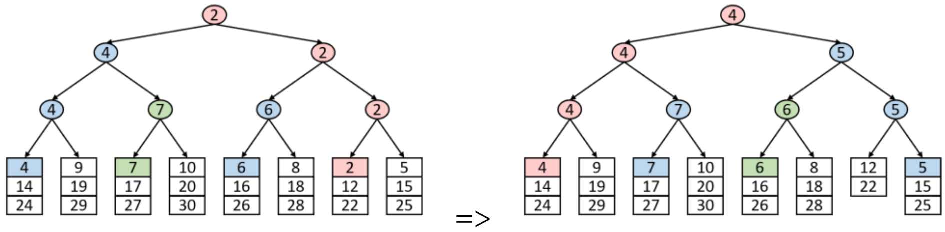
分析：令有 k 個 Runs，n 為 Data 數

⇨ 每次從 k 個 Runs 找出 min 值花 O(k)時間，最多比 n-1 次，因此需 O(n*k)

當採 Selection Tree，分為：

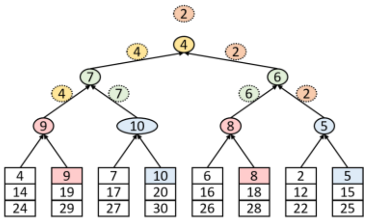
1. Winner Tree
2. Loser Tree

1. Winner Tree，例：8-way Merge



分析：
建立 Tree 需花 O(k)
而每回需找 min，花 O(log k)
又需執行 O(n)回合
故總時間 = O(k) + O(n log k)

2. Loser Tree，例：8-way Merge

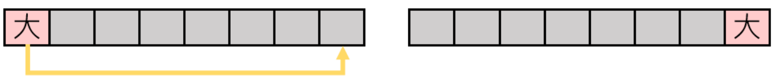


分析：
Time Complexity 和 Winner Tree 一樣，但會更快一些

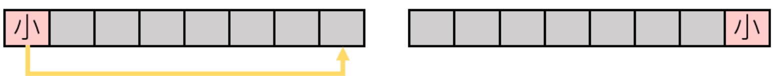
Heap Sort

概念：將 Data 建立成 Heap 之後，做 n-1 次的 Heap Delete

Note：
Max-Heap：
Delete & Outpu => 大到小
用 Array 存放 => 小到大



Min-Heap：
Delete & Outpu => 小到大
用 Array 存放 => 大到小



程式：

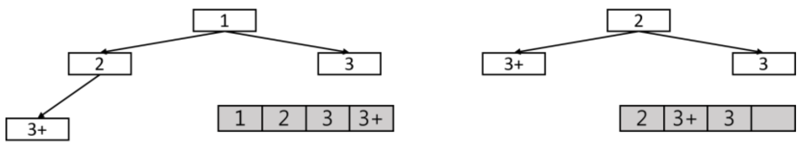
```
HeapSort(tree, n)                       //主程式
{
        for i=[n/2] down to 1 do        //i=最後父點
                Adjust(tree, i, n);
        for i=m-1 down to 1 do
        {
                swap(tree[i], tree[i+1])    //最後的 Node
                Adjust(tree, 1, i);         //Heap 調整：O(log n)
        }
}

Procedure adjust(tree, i, n)            //副程式
{
        int j, k;
        Node r;
        bool done;
        done=false; r=tree[i];   k=tree[i].key;
        j=2*i;
        while(j<=n && !done)
        {
                if(j<n)                     //成立有右兒子
                        if(tree[j].key<tree[j+1].key)
                                j=j+1;
                if(k>=tree[j].key)
                        done=true;
                else
                {
                        tree[j/2]=tree[j];
                        j=2*j;
                }
        }
        tree[j/2]=r;
}
```

分析：

1. Time Complexity：建立+Sorting = O(n) + O(n log n) = O(n log n)
2. Space Complexity：O(1)
3. 為 Unstable Sorting 演算法

   例：



順序對調

線性排序(Linear-Time Sorting)：O(n)

若採非 Comparsion-Based 排序，則有機會突破 Ω (n log n)之 Time Complexity，來到 O(n)

方法：

1. Radix Sort：偏 DS
2. Bucket Sort(Bin Sort)：偏演算法
3. Counting Sort：偏演算法

Radix Sort(基底排序)

又稱 Bin Sort 或 Bucket Sort，採用”Distribution & Merge”技巧，(DS 版)可分為：

1. LSD Radix Sort(常見)：即是演算法版中的 Radix Sort
2. MSD Radix Sort：即是演算法版中的 Bucket/Bin Sort

LSD Radix Sort

令 r 為採用之基底(Base)，則需準備 r 個桶子(編號 0~r-1)

令 d 為 Input Data 中最大鍵值之位數個數，則表示需作 d 回合完成 Sorting

例：99, 28, 36, 8, 357(d=3 位數)

從最低位數→高位數，依序做 Distribution & Merge

Note：each pass 需做

1. 分派(Distribution)：依 Data 某數值，將它分派到對應之 Bucket 中
2. 合併(Merge)：由 0→(r-1)的 Bucket 依序合併其中之 Data

例：LSD Radix Sort 採 10 進位制，input：179, 208, 306, 93, 859, 984, 55, 9, 271, 33？

179, 208, 306, 93, 859, 984, 55, 9, 271, 33

Pass 1	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
依個位數		*271*		*93* *33*	*984*	*55*	*306*		*208*	*179* *859* *9*

⇨ *271, 93, 33, 984, 55, 306, 208, 179, 859, 9*

Pass 2	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
依十位數	*306* *208* *9*			*33*		*55* *859*		*179*	*984*	*93*

⇨ *306, 208, 9, 33, 55, 859, 271, 179, 984, 93*

Pass 3	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
依百位數	*9* *33* *55* *93*	*179*	*208* *271*	*306*					*859*	*984*

⇨ *9, 33, 55, 93, 179, 208, 271, 306, 859, 984*

分析：

1. Time Complexity：O(d*(n+r))
   說明：因為要作"d"回合，且每回合需作 2 動作：
   (1) 分派：花 O(n)
   (2) 合併：花 O(r)，因為是針對 Bucket 內容合併
   因此一回合需花 O(n+r)，故總時間花 O(d*(n+r))

   Why O(d*(n+r))視為 Linear？
   因為 r 為基底，可視為常數；若 Sorting 的 Data Range 已知，則 d 可視為常數，故 O(d*(n+r))視為 Linear = O(n)

2. Space Complexity：需要 r 個 Bucket，且每個 Bucket 的最大格數為 n
   因此為 O(r*n)

3. 為 Stable Sorting 演算法
   例：
   未排序：…, 5, …, 5+, …
   已排序：…, (5, 5+), …

MSD Radix Sort(**演算法版中的** Bucket Sort)

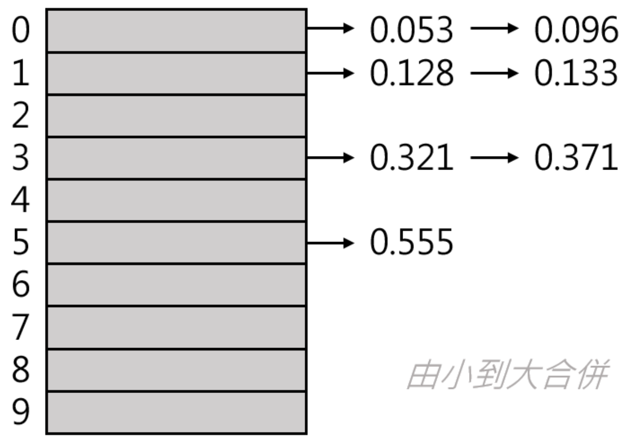
步驟：

1. 依各 Data 之最高位數值分派到對應之 Bucket
2. 每個 Bucket 各自 Sorting
3. 最後合併 Bucket 之 Data，由 0→(r-1)依序合併

Note：跟 LSD 不同，分派及合併只作"1 次"

例：採 MSD Radix Sort(Bucket Sort)，Input：371, 128, 96, 133, 555, 321, 53

⇨ *0.371, 0.128, 0.096, 0.133, 0.555, 0.321, 0.053*
   依小數後的"第一位"數值做分派



Counting Sort

例：鍵值 Range：1~6；n：8 筆 Data；Input：3, 1, 4, 2, 6, 1, 2, 2,

1. *準備一個 count[1: 6]，初始值為 0，統計各鍵值出現次數*

	1	2	3	4	5	6
count	2	3	1	1	0	1

2. *求出各鍵值將來排序之後的起始位置，記錄在 start[1: 6]之中，其中 start[1]=1*
   Note：start[i] = start[i-1] + count[i+1]

	1	2	3	4	5	6
start	1	3	6	7	8	8

3. *依 start[i]指示，將鍵值 i 的 Data 置入 Output Array 的正確位置，且 start[i]往下+1*

	1	2	3	4	5	6
start	1	3	6	7	8	8

	1	2	3	4	5	6	7	8
Output	1	2	2	2	2	3	4	6

程式：

```
for i=1 to k do                              //1.O(k)
        count[i]=0;
for i=1 to n do                              //O(n)算出各 key 出現次數
        count[A[i]]=count[A[i]]+1;

start[i]=1;                                  //2.O(k)
for i=2 to k do                              //算各 key 的起始位置
        start[i]=start[i-1]+count[i-1];
for i=1 to n do                              //3.O(n)
{
        output[start[A[i]]=A[i]];            //正式 Sorting
        start[A[i]]=start[A[i]]+1;
}
```

由上可知，其 Time Complexity 為 O(n+k)

Why Linear？

一般 k 為 Data Range，若已受限制，則 k 為常數，故 O(n)

例：已知 Counting Sort 之 Time Complexity 為 O(n+k)，且若 k∈ O(n)線性等級，則 Counting Sort 為 Linear Time。但是：若 k∈ $O(n^2)$等級，則 $O(n+k)=O(n+n^2)$ => $O(n^2)$
問：在此之下，可否建立出 Linear Time 的 Sorting？

假設 k: 0~9，Data Range: 0~99
可採 Radix Sort 概念搭配做 Counting Sort

1. *針對個位數做 Counting Sort：key%1 => 得個位數 => Counting Sort 後之結果*
2. *針對十位數做 Counting Sort：key%10 => 得十位數 => Counting Sort 後之結果*
3. …

*結果由小到大，Time Complexity：2*O(n) ∈O(n)*
⇨ *不論多大，皆為 O(n)*

[重要]比較表

		Time Complexity			Space Complexity	Stable/Unstable
		Best	Average	Worst		
初等	Insertion	**O(n)**	$O(n^2)$	$O(n^2)$	O(1)	**Stable**
	Selection	O(n2)	$O(n^2)$	$O(n^2)$	O(1)	**Unstable**
	Bubble	**O(n)**	$O(n^2)$	$O(n^2)$	O(1)	**Stable**
	Shell	**$O(n^{3/2})$**	$O(n^2)$	$O(n^2)$	O(1)	**Unstable**
高等	Quick	O(log n)	O(log n)	**$O(n^2)$**	**O(log n)~O(n)**	**Unstable**
	Merge	O(log n)	O(log n)	O(log n)	**O(n)**	**Stable**
	Heap	O(log n)	O(log n)	O(log n)	**O(1)**	**Unstable**
線性	Radix	**O(d*(n+r))** => O(n)			**O(r*n)**	**Stable**
	Counting	**O(n+k)** => O(n)			**O(n+k)**	**Stable**

Selection Problem(**演算法問題**)

在 n 個 Data 中 Select ith smallest item(挑出第 i 小的資料)

例：3, 5, 8, 1, 7, 6, 9, 4，問第 4 小的 Data 為何？

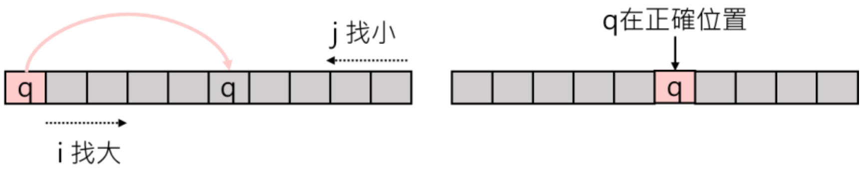
How to do？

[法一]直覺作法：

1. 先由小到大排序：O(n log n)
2. Return A[i]：O(1)

⇨ O(n log n)

[法二]利用 Quick Sort 之"Partition"實施



k = q-l+1 => pk 為 k^{th} 小的值*(k：此次 pk 所在；i：欲找的 Data 次序)*

承上，若 q=10, l=3, u=20，pk 為$(q-l+1)^{th}$小的 Data

判別：

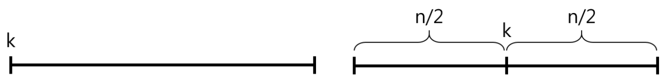
1. i = 8^{th} = k => found, pk 即是
2. i = 5^{th} < k => 往左邊找 5^{th} 小的值
3. i = 12^{th} > k => 往右邊找第 4 小的值

程式：

```
Select(A, l, u, i)                          //在 A[l]~A[u]找 ith 小的 Data
{
        if(l<u)
        {
                q=Partition(A, l, u);       //Quick Sort 演算法版中的副程式
                k=q-l+1;                            //pk 是 kth 小
                if(i==k)       return A[q];
                else if(i<k)
                        return Select(A, l, q-1, i);
                else
                        return Select(A, q+1, u, i-k);
        }
}
```

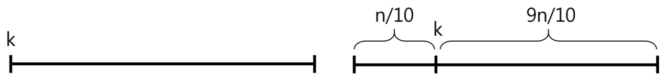
分析：

1. Time Complexity
   (1) Best Case：O(n)
   說明：



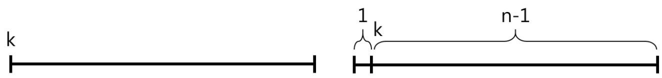
$T(n) = T(n/2) + cn \Rightarrow O(n)$

   (2) Average Case：
   說明：在不失一般性的情況下



$T(n) = T(9n/10) + cn \Rightarrow O(n)$

   (3) Worst Case：
   說明：



$T(n) = T(n-1) + cn \quad \Rightarrow O(n^2)$

改善方式：採 Medium of Mediums 時，可使在 Worst Case 之下，依舊為 O(n)

Find Min and Max among n items

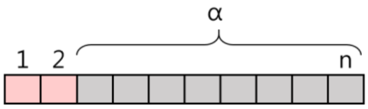
[法一]直覺法

花(n-1)次比較找出 min/max

全部花費的時間為：2(n-1)次之比較

[法二]

作法：



1. 比 A[1], A[2]一次，可知誰大誰小
   令 x=min(A[1], A[2])、y=max(A[1], A[2])
2. 針對後面的(n-2)筆 Data，實施”Find min and max”之”Recursive”，即可得(n-2)筆中的 min 與 max 值(α, β)
3. X 跟 α 比，得 min、y 跟 β 比，得 max
   $T(n) = T(n-2) + 3 \Rightarrow O(3n/2)$
   可知：[法二]較佳，因為比較次數較少