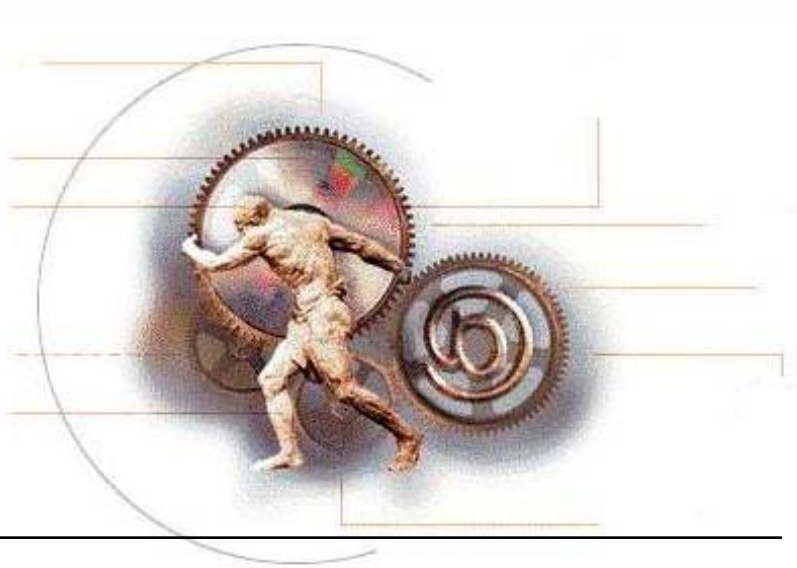


Course 3

排序
Sort

國立聯合大學 資訊管理學系 陳士杰老師

Outlines

- 本章重點
  - Sort分類觀點
  - 初等排序方法 (Avg. Case: $O(n^2)$)
    - Insertion Sort
    - Selection Sort
    - Bubble Sort
  - 高等排序方法 (Avg. Case: O(n log n))
    - Quick Sort
    - Merge Sort
    - Heap Sort
  - Radix Sort

Sort 分類觀點

- Internal Sort v.s. External Sort.
- Stable Sorting Method v.s. Unstable Sorting Method.

Internal Sort v.s. External Sort

- 觀點: 資料量的多寡
- Internal Sort:
  - Def: 資料量少，可以一次全部置於Memory中進行sort之工作
  - 目前大多數的Sort方法皆屬此類
- External Sort:
  - Def: 資料量大，無法一次全置於Memory中，須藉助輔助儲存體 (E.g. Disk)進行sort之工作

Stable Sorting Method v.s. Unstable Sorting Method

- 假設欲排序的資料中，有多個記錄具有相同的鍵值 (如: ..., k, ..., k, ...)，經過排序之後，結果可能爲:
  - ..., k, k, ...: 會得到此結果的排序方法稱之爲**Stable**
  - ..., k, k, ...: 會得到此結果的排序方法稱之爲**Untable**

曾有不必要的Swap發生!!

  - 例:

    原始: 5, 4, 2, 6, 4, 7, 1

    ⇨Stable: 1, 2, 4, 4, 5, 6, 7

    ⇨Unstable: 1, 2, 4, 4, 5, 6, 7

初等排序方法

- Avg. Case Time Complexity: $O(n^2)$
  - Insert Sort
  - Selection Sort
  - Bubble Sort

Insert Sort (插入排序)

- 將第i筆記錄插入到前面(i-1)筆已排序好的記錄串列中，使之成爲i筆已排序好的記錄串列。 (需執行n-1回合)
- 範例:
  - A sequence 9, 17, 1, 5, 10。以遞增(increase)排序

- 根據上例，可知若有n筆記錄，則需做(n-1)回合。
- Algorithm主要由2個副程式組成:
  - Insert副程式
    - 將某一筆記錄 x 插入到S[1] ~ S[i-1]等 i-1 筆已排序好的串列中，使之成爲 i 筆已排序好的串列。
    - 即: 決定x插入的位置
  - Sort副程式 (可當作主程式)
    - 將未排序好的記錄透過Insert的動作，使之成爲排序好的記錄
    - 共需做n-1回合，且由第二筆資料開始做起，∴迴圈: for i = 2 to n

Algorithm 7.1: Insertion Sort

Problem: Sort n keys in nondecreasing order.

Inputs: positive integer n; array of keys of keys S indexed from 1 to n.

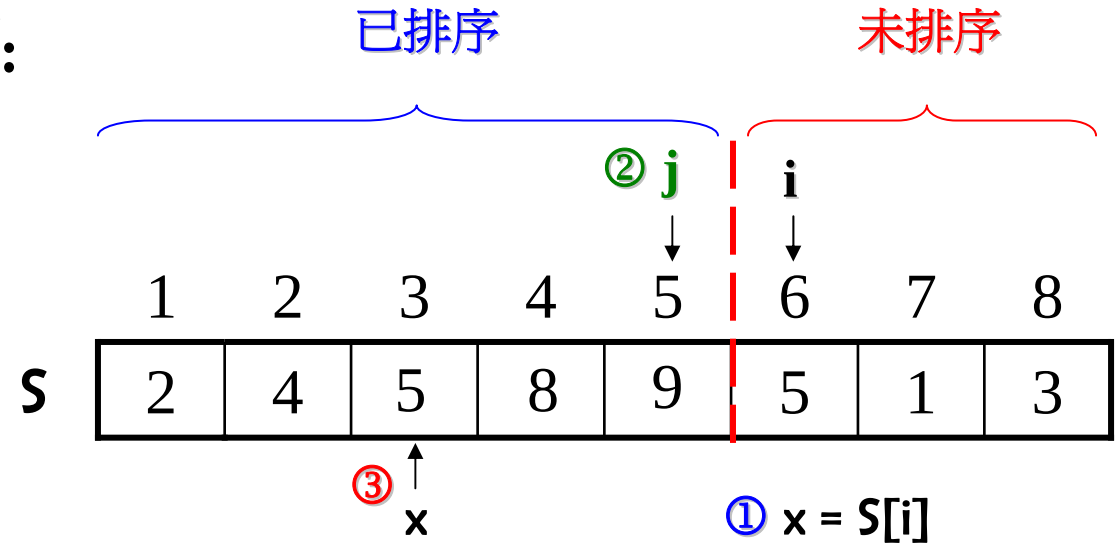
Outputs: the array S containing the keys in nondecreasing order.

```
void insertionsort (int n, keytype S[])
{
index i, j;
keytype x;
for (i=2; i <= n; i++){
     x = S[i];
     j = i-1;
     while (j > 0 && S[j] > x){
           S[j + 1] = S[j];
            j--;
      }
      S[j +1] = x
}
}
```

Sort副程式
(可看成主程式)

Insert副程式

- 範例:



- Insert副程式:

```
① x = S[i];
② j = i-1;
  while (j > 0 && S[j] > x){
        S[j + 1] = S[j];
         j--;
   }
③  S[j +1] = x
```

分析

- Time Complexity
  1) Best Case
  2) Worst Case
  3) Average Case
- Space Complexity
- Stable / Unstable

Time-Complexity

- **Best Case: O(n)**
  - 當輸入資料已經是由小到大排好時。

[分析方法 1]:

1, 5, 9, 10, 17

比較一次

比較一次

比較一次

比較一次

- n筆資料需作(n-1)回合，且每回合只比較1次即可決定x的插入位置。∴總共花費(n-1)次比較即完成Sort工作。
- Time complexity = O(n)

[分析方法 2]: 利用遞迴時間函數

前(n-1)筆比較的遞迴執行次數

第 n 筆資料的最佳比較次數

$$T(n) = T(n-1) + 1$$

$$= (T(n-2) + 1) + 1 = T(n-2) + 2$$

$$= (T(n-3) + 1) + 2 = T(n-3) + 3$$

$$= \dots$$

$$= T(0) + n$$

沒有資料，所以比較次數 T(0) = 0

$$\therefore T(n) = O(n)$$

- **Worst Case: $O(n^2)$**
  - 當輸入資料是由大到小排好時。

[分析方法 1]:

比一次

比二次

比三次

比四次

- n筆資料總共比較次數= 1+2+3...+(n-1) = [n(n-1)]/2。
- Time complexity = $O(n^2)$

[分析方法 2]: 利用遞迴時間函數

前(n-1)筆比較的執行次數

第 n 筆資料的最差比較次數

沒有資料，所以比較次數 T(0) = 0

$$T(n) = T(n-1) + (n-1)$$

$$= (T(n-2) + (n-2)) + (n-1) = T(n-2) + (n-2) + (n-1)$$

$$= (T(n-3) + (n-3)) + (n-2) + (n-1)$$

$$= \ldots$$

$$= T(0) + 0 + 1 + 2 + \ldots + (n-3) + (n-2) + (n-1)$$

$$= 1 + 2 + \ldots + (n-3) + (n-2) + (n-1) = n(n-1)/2$$

$$\therefore T(n) = O(n^2)$$

◆Average Case: O(n²)

[分析方法]: 利用遞迴時間函數

遞迴呼叫的執行次數

第 n 筆資料的比較次數

$$T(n) = T(n-1) + n/2$$
$$= T(n-1) + cn$$
$$= T(n-2) + c(n-1) + cn$$
$$= \ldots$$
$$= T(0) + c(1+2+\ldots+n)$$
$$= c\,[n(n+1)]/2$$
$$\therefore T(n) = O(n^2)$$

- 第n筆資料與前n-1筆資料可能的比較次數有 1次，2次，3次，...，(n-1)次。因此，第n筆資料與前n-1筆資料的比較次數總合爲: 1+2+3+...(n-1)
- 第n筆資料要比較的資料數爲(n-1)
- 因此，第n筆資料的平均比較次數爲:

$$\frac{1+2+3+\ldots+(n-1)}{n-1}$$
$$\Rightarrow \frac{\frac{n(n-1)}{2}}{n-1}$$
$$\Rightarrow \frac{n}{2}$$

Space-Complexity

Algorithm 7.1: Insertion Sort

Problem: Sort *n* keys in nondecreasing order.

Inputs: positive integer *n*; array of keys of keys *S* indexed from 1 to *n*.

Outputs: the array *S* containing the keys in nondecreasing order.

```
void insertionsort (int n, keytype S[])
{
index i, j;
keytype x;
for (i=2; i <= n; i++){
     x = S[i];
     j = i-1;
     while (j > 0 && S[j] > x){
           S[j + 1] = S[j];
            j--;
      }
      S[j +1] = x
}
}
```

Simple variables

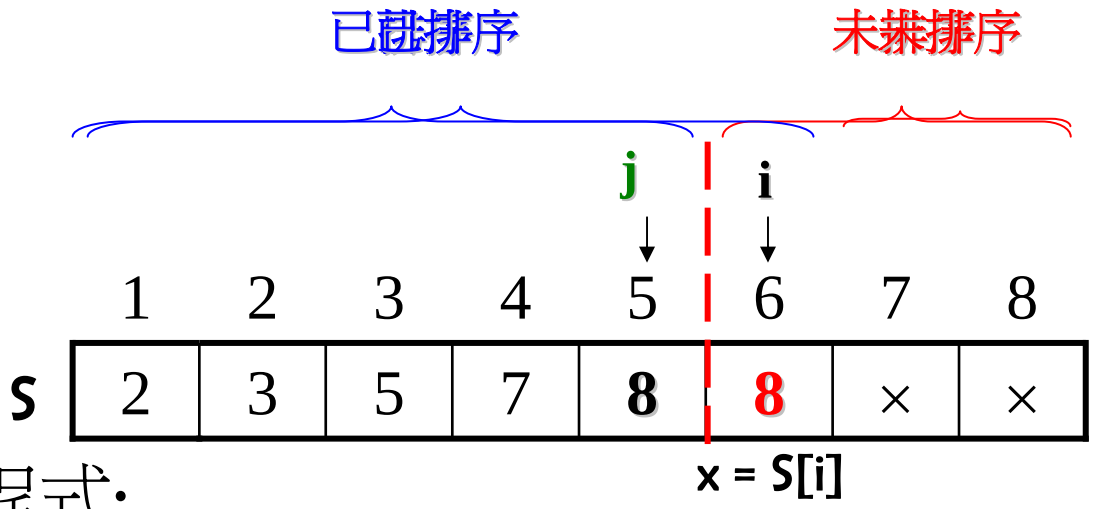
- 有structure variable，考量參數傳遞是不是call by value:
  - = n, S[ ]若爲**call by value** 傳遞 (根據主程式所傳來的數值型態與數值多寡)
  - = ● (或一常數，即起始位址值), S[ ]若爲 **call by address** 傳遞 (∵主程式只傳陣列的起始位址，無變動空間需求)
- 在C++中傳遞陣列一般是使用傳址的方式。
  - ∵在C++中，陣列的名稱是指向陣列的開始位址，所以呼叫函式時，只要將陣列名稱傳給函式即可

- 由以上分析，可以得知:
  - S(P) = C + SP(I)

    = C + 0 (或一常數)
  - 因此，除了存放輸入資料之外，額外的空間需求(Extra space)是固定的 (e.g., 變數 x, i, j, ... 等)。
- ∴ Space Complexity: Θ(1) (或Θ(C), C爲一常數)

Stable / Unstable

- Stable (穩定的)
- 說明:



- Insert副程式:

```
x = S[i];
j = i-1;
while (j > 0 && S[j] > x){
        S[j + 1] = S[j];
         j--;
 }
 S[j +1] = x
```

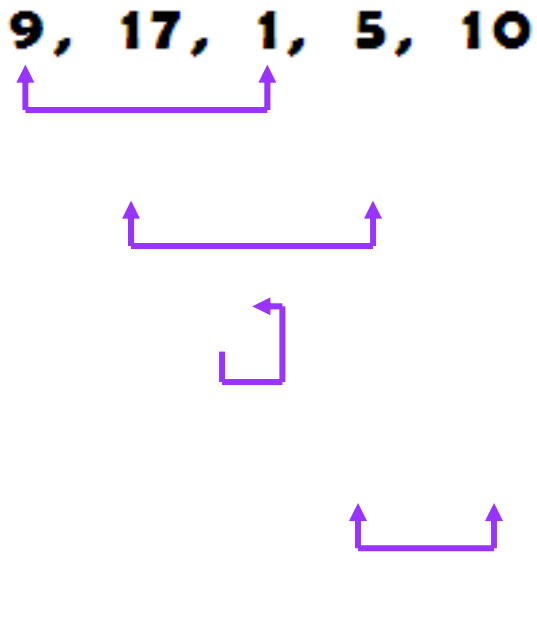
Sort前: 2 3 5 7 8 8 × ×

Sort後: 2 3 5 7 8 8 × ×

∵相同鍵值的記錄在排序後，其<u>相對位置沒有改變</u>，亦即無不必要的Swap發生, ∴**Stable**

Selection Sort (選擇排序)

- 自第i筆到第n筆記錄中，選擇出最小鍵值 (key) 的記錄，然後與第i筆記錄Swap。 (i 的值從1到n-1，需執行n-1回合)
- 範例:
  - A sequence 9, 17, 1, 5, 10。以遞增(increase)排序



◆根據上例，可知若有**n**筆記錄，則需做**(n-1)**回合。

Algorithm 7.2: Selection Sort

Problem: Sort *n* keys in nondecreasing order.

Inputs: positive integer *n*; array of keys *S* indexed from 1 to *n*.

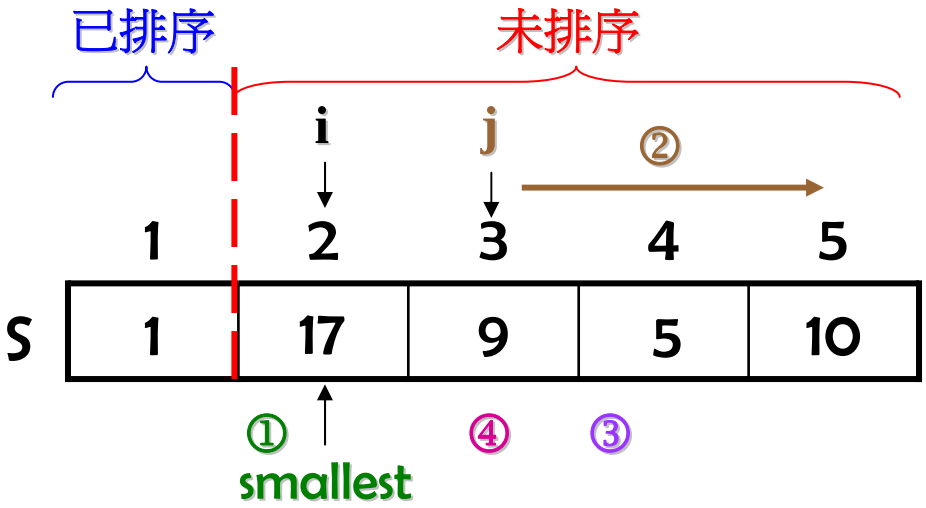
Outputs: the array *S* containing the keys in nondecreasing order.

```
void selectionsort (int n, keytype S[])
{
    index i, j, smallest;
    for (i = 1; i <= n - 1; i++){
      smallest = i;
      for (j = i + 1; j <= n; j++)
        if (S[j] < S[ smallest])
          smallest = j;
      exchange S[i] and S[ smallest];
    }
}
```

Sort副程式

Selection副程式

- 範例: (Pass 1 → Pass 2)



- Selection副程式:

```
① smallest = i;
② for (j = i + 1; j <= n; j++)
     if (S[j] < S[ smallest])
     ③ smallest = j;
④ exchange S[i] and S[ smallest];
```

分析

- Time Complexity
  1) Best Case
  2) Worst Case
  3) Average Case
- Space Complexity
- Stable / Unstable

Time-Complexity

- Best / Average / Worst Case: $O(n^2)$
  - 不論輸入資料如何，演算法中的兩個迴圈皆會執行。

```
void selectionsort (int n, keytype S[])
{
    index i, j, smallest;
    for (i = 1; i <= n - 1; i++){
      smallest = i;
      for (j = i + 1; j <= n; j++)
        if (S[j] < S[ smallest])
          smallest = j;
        exchange S[i] and S[ smallest];
    }
}
```

- 說明:

i值	i = 1	i = 2	...	i = n-1
j值	j = 2 to n	j = 3 to n	...	j = n-1 to n
第二層迴圈內if指令之比較次數	執行n-1次	執行n-2次	...	執行1次

⇨總執行次數: $\sum_{i=n-1}^{1} i = (n-1)+(n-2)+\ldots+1 = \frac{n(n-1)}{2}$

⇨ $O(n^2)$

Space-Complexity

Algorithm 7.2: Selection Sort

Problem: Sort *n* keys in nondecreasing order.

Inputs: positive integer *n*; array of keys *S* indexed from 1 to *n*.

Outputs: the array *S* containing the keys in nondecreasing order.

```
void selectionsort (int n, keytype S[])
{
    index i, j, smallest;
    for (i = 1; i <= n - 1; i++){
      smallest = i;
      for (j = i + 1; j <= n; j++)
        if (S[j] < S[ smallest])
          smallest = j;
        exchange S[i] and S[ small
  }
}
```

Simple variables

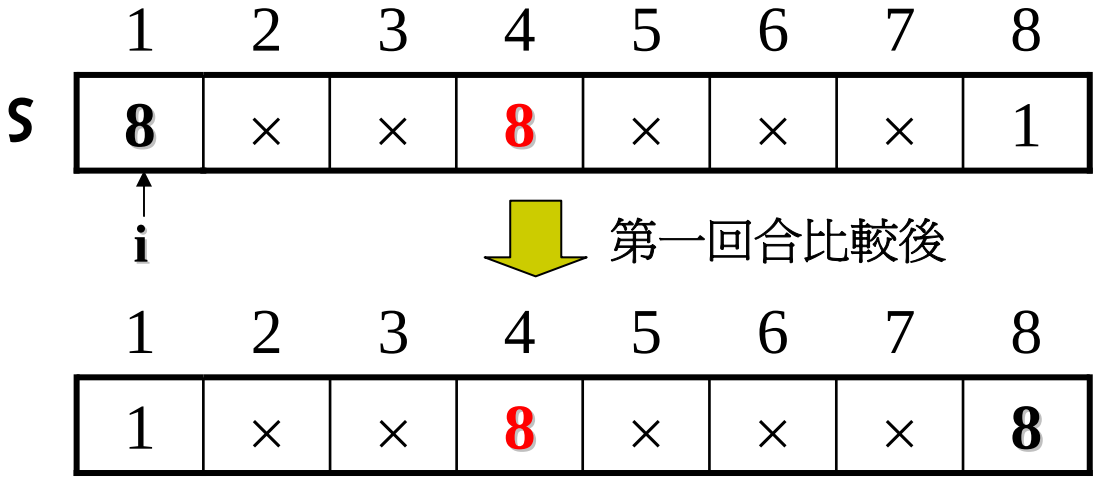
- 有structure variable，考量參數傳遞是不是call by value:
  - = n, S[ ]若爲**call by value** 傳遞 (根據主程式所傳來的數值型態與數值多寡)
  - = ● (或一常數，即起始位址值), S[ ]若爲**call by address** 傳遞 (∵主程式只傳陣列的起始位址，無變動空間需求)
- 在C++中傳遞陣列一般是使用傳址的方式。
  - ∵在C++中，陣列的名稱是指向陣列的開始位址，所以呼叫函式時，只要將陣列名稱傳給函式即可

- 由以上分析，可以得知:
  - S(P) = C + SP(I)

    = C + 0 (或一常數)
  - 此外，除了存放自主程式傳來的輸入資料之外，副程式本身額外的空間需求(Extra space)是固定的(e.g., 變數 i、j、smallest, temp in Swap函數, ...等) 。
- ∴ Space Complexity: Θ(1) (或Θ(C), C爲一常數)

Stable / Unstable

- Unstable (不穩定的)
- 說明: (1爲最小值，8爲最大值)



- Selection副程式:

```
smallest = i;
for (j = i + 1; j <= n; j++)
  if (S[j] < S[ smallest])
    smallest = j;
  exchange S[i] and S[ smallest];
```

∵相同鍵值的記錄在排序後，其<u>相對位置有改變</u>，亦即有<u>不必要的**Swap**發生</u>,

∴**Unstable**

Bubble Sort (氣泡排序)

- 由左至右，兩兩記錄依序互相比較 (需執行n-1回合)
  - if (前者 > 後者) then Swap(前者, 後者)
- 若在某回合處理過程中，沒有任何Swap動作發生，則Sort完成，後續回合不用執行。

- 範例 1:
  - A sequence 26, 5, 77, 19, 2。以遞增(increase)排序

	26	5	77	19	2
	比較	比較	比較	比較	
Pass 1:					
	比較	比較	比較		
Pass 2:					
	比較	比較			
Pass 3:					
	比較				
Pass 4:					

第一回合後，最大的**Bubble**在最高位置上

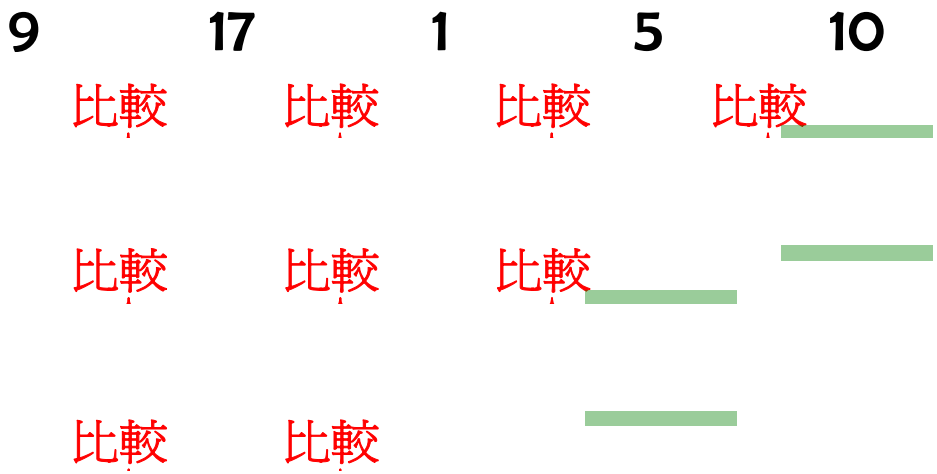
第二回合後，次大的**Bubble**在次高位置上

第三回合後，第三大的**Bubble**在第三高位置上

第四回合後，第四大的**Bubble**在第四高位置上

Solution:

- 範例 2:
  - A sequence 9, 17, 1, 5, 10。以遞增(increase)排序



第一回合後，最大的**Bubble**在最高位置上

第二回合後，次大的**Bubble**在次高位置上

第三回合後，沒有任何**Swap**發生。∴完成

- **Algorithm主要由2個副程式組成:**
  - **Bubble副程式**
    - 由左至右，兩兩記錄依序互相比較
    - **if (前者 > 後者) then Swap(前者, 後者)**
  - **Sort副程式 (可當作主程式)**
    - 將未排序好的記錄透過**Bubble**的動作，使之成爲排序好的記錄
    - 共需做 n-1 回合，且由第 1 筆資料開始做起，∴迴圈: **for i = 1 to (n-1)**
    - 若在某回合處理過程中，沒有任何**Swap**動作發生，則Sort完成，後續回合不用執行

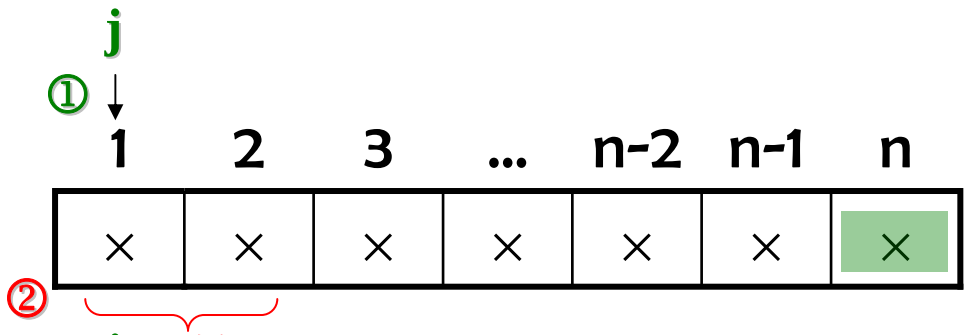
Sort副程式

```
void BubbleSort(S[ ], n)
{
  for (i = 1, i ≤ (n-1), i++)
    {
      f = 0;   //用以表示有無 Swap 發生。0 表示無 Swap 發生
      for (j = 1, j ≤ (n-i), j++)
        if (S[j+1] < S[j])
          {
            swap(S[j+1], S[j]);
            f = 1;   //有 Swap 發生
          };
      if (f=0)
        exit;   //若無任何 Swap 發生，則 Sort 完成
    };
}
```

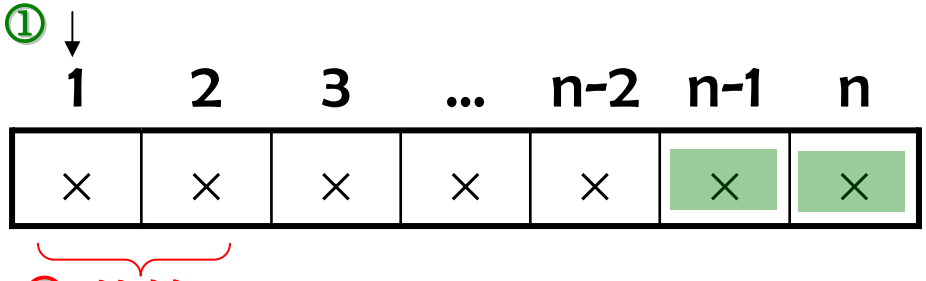
Bubble副程式

- 範例:

i = 1 (即 Pass 1):



i = 2 (即 Pass 2):



- Bubble副程式:

```
① for (j = 1, j ≤ (n-i), j++)
② if (S[j+1] < S[j])
      {
        swap(S[j+1], S[j]);
        f = 1;   //有 Swap 發生
      };
```

分析

- Time Complexity
  1) Best Case
  2) Worst Case
  3) Average Case
- Space Complexity
- Stable / Unstable

Time-Complexity

- Best Case: **O(n)**
  - 當輸入資料已經是由小到大排好時。

[說明]:

∵ 只執行**Pass 1**，且無任何**Swap**發生，表示Sort完成。
因此，總共只有 **(n-1)** 次比較即完成Sort。

∴ Time = O(n)。

- **Worst Case: O(n²)**
  - 當輸入資料是由大到小排好時。

[分析方法 1]:

i值	i = 1	i = 2	...	i = n-1
j值	j = 1 to n-1	j = 1 to n-2	...	j = 1 to 1
第二層迴圈內Swap執行次數	執行n-1次	執行n-2次	...	執行1次

⇨總執行次數: $\sum_{i=n-1}^{1} i = (n-1)+(n-2)+\ldots+1 = \frac{n(n-1)}{2}$

⇨ $O(n^2)$

[分析方法 2]: 利用遞迴時間函數

剩下之(n-1)筆資料的Swap 執行次數

n 筆資料於Pass 1 時的Swap次數

沒有資料，所以 Swap次數 T(0) = 0

$$T(n) = T(n-1) + (n-1)$$

$$= (T(n-2) + (n-2)) + (n-1) = T(n-2) + (n-2) + (n-1)$$

$$= (T(n-3) + (n-3)) + (n-2) + (n-1)$$

$$= \ldots$$

$$= T(0) + 0 + 1 + 2 + \ldots + (n-3) + (n-2) + (n-1)$$

$$= 1 + 2 + \ldots + (n-3) + (n-2) + (n-1) = n(n-1)/2$$

$$\therefore \underline{T(n) = O(n^2)}$$

◆Average Case: O(n²)

[分析方法]: 利用遞迴時間函數

剩下之(n-1)筆資料的Swap執行次數

n 筆資料於Pass 1的平均Swap次數

$$T(n) = T(n-1) + (n-1)/2$$
$$= T(n-1) + cn$$
$$= T(n-2) + c(n-1) + cn$$
$$= \ldots$$
$$= T(0) + c(1+2+\ldots+n)$$
$$= c\,[n(n+1)]/2$$
$$\therefore T(n) = O(n^2)$$

- n筆資料可能的Swap次數有 1次，2次，3次，...，(n-1)次。因此，n筆資料的Swap次數總合爲: 1+2+3+...(n-1)
- n筆資料要Swap的資料數爲n
- 因此，n筆資料的平均Swap次數爲:

$$\frac{1+2+3+\ldots+(n-1)}{n}$$
$$\Rightarrow \frac{\frac{n(n-1)}{2}}{n}$$
$$\Rightarrow \frac{n-1}{2}$$

Space-Complexity

```
void BubbleSort(S[ ], n)
{
  for (i = 1, i ≤ (n-1), i++)
    {
      f = 0;   //用以表示有無 Swap 發生。0 表示無 Swap 發生
      for (j = 1, j ≤ (n-i), j++)
        if (S[j+1] < S[j])
          {
            swap(S[j+1], S[j]);
            f = 1;   //有 Swap 發
          };
        if (f=0)
          exit;   //若無任何 Swap
      };
}
```

Simple variables

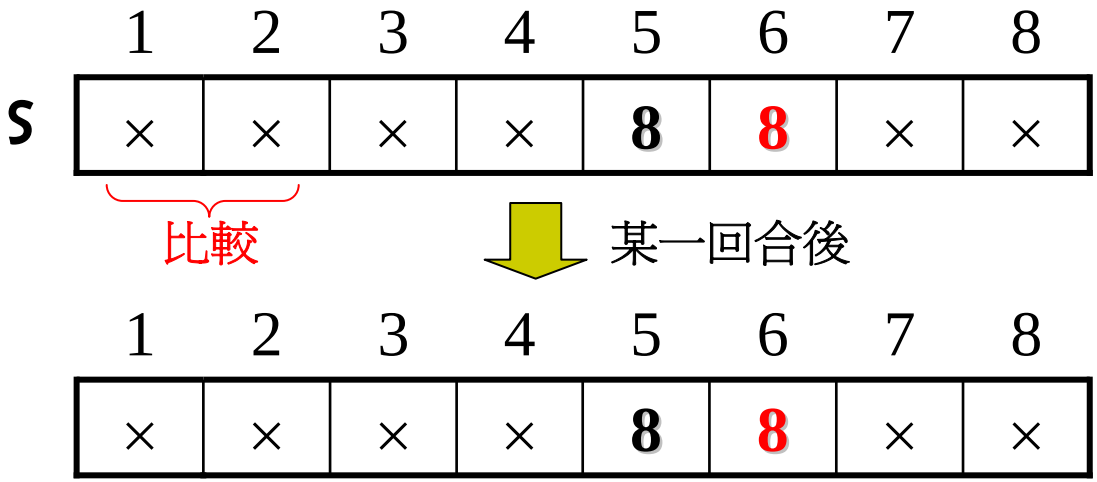
- 有structure variable，考量參數傳遞是不是call by value:
  - = n, S[ ]若爲**call by value** 傳遞 (根據主程式所傳來的**數值型態與數值多寡**)
  - = 0 (或一常數，即**起始位址值**), S[ ]若爲**call by address** 傳遞 (∵主程式只傳陣列的**起始位址**，無變動空間需求)
- 在C++中傳遞陣列一般是使用**傳址**的方式。
  - ∵在C++中，陣列的名稱是指向陣列的開始位址，所以呼叫函式時，只要將陣列名稱傳給函式即可

- 由以上分析，可以得知:
  - S(P) = C + SP(I)

    = C + 0 (或一常數)
  - 因此，除了存放由主程式傳送過來的輸入資料之外，額外的空間需求(Extra space)是固定的(e.g., 變數 i、j、f, temp in Swap函數, ...等) 。
- ∴ Space Complexity: Θ(1) (或Θ(C), C爲一常數)

Stable / Unstable

- Stable (穩定的)
- 說明:



- Bubble副程式:

```
for (j = 1, j ≤ (n-i), j++)
  if (S[j+1] < S[j])
    {
      swap(S[j+1], S[j]);
      f = 1;   //有 Swap 發生
    };
```

∵相同鍵值的記錄在排序後，其相對位置沒有改變，亦即沒有不必要的Swap發生,

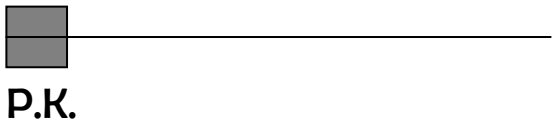
∴Stable

高等排序方法

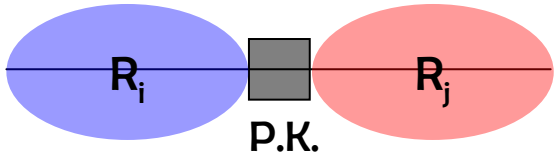
- Avg. Case Time Complexity: O(n log n)
  - Quick Sort
  - Merge Sort
  - Heap Sort

Quick Sort (快速排序)

- Avg. case 下，排序最快的algo.
- Def:
  - 將大且複雜的問題切成許多獨立的小問題，再加以解決各小問題後，即可求出問題的Solution。
  - 此即 "**Divide-and-Conquer**" (切割並征服)的解題策略。
- 觀念:
  - 將第一筆記錄視爲Pivot Key (樞紐鍵 (P.K.) ，或稱Control Key)，在Pass 1 (第一回合) 後，可將P.K.置於 "最正確" 的位置上。
  - Ex:



(經過Pass 1) ⇨

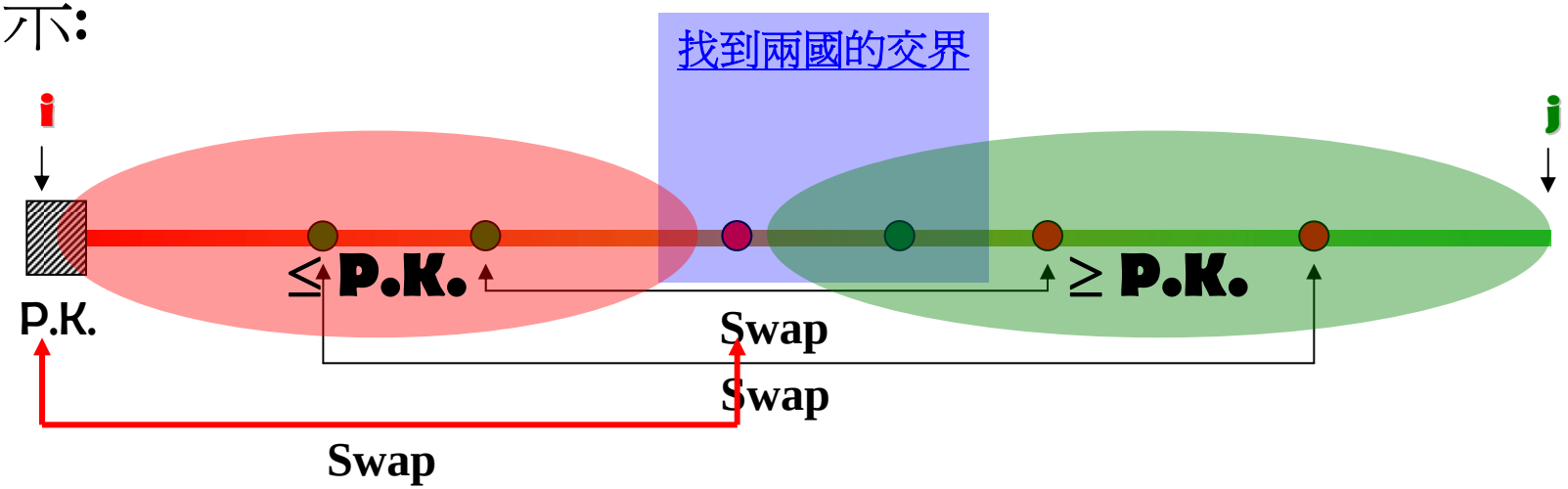


, $R_i.key \le P.K.$ 且 $R_j.key \ge P.K.$

  - 把P.K.擺在正確的位置 ⇨爲切割的概念 (∵可使用遞迴)

- **[關鍵點]**: 如何決定P.K.之 “最正確” 位置?
  - 設兩個整數變數 i，j
    - **i**: 找 ≥ **P.K.**者
    - **j**: 找 ≤ **P.K.**者

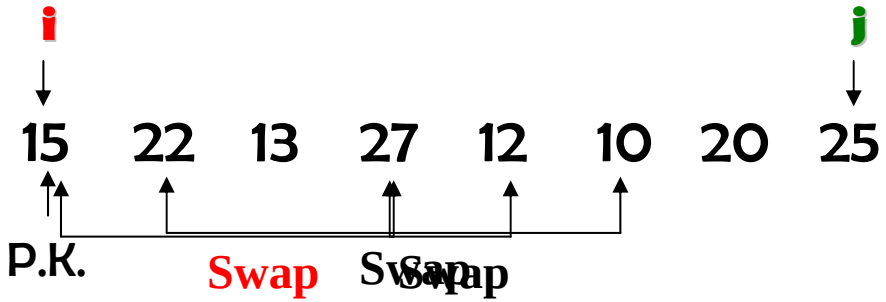
圖示:



- ●: 在數字串列中 ≥ P.K.的值
- ●: 在數字串列中 ≤ P.K.的值

◆範例: 15, 22, 13, 27, 12, 10, 20, 25由小至大排序

Sol:



Pass 1: [12 10 13] 15 [27 22 20 25]

Pass 2: [10] 12 [13] 15 [27 22 20 25]

Pass 3: 10 12 [13] 15 [27 22 20 25]

Pass 4: 10 12 13 15 [27 22 20 25]

Pass 5: 10 12 13 15 [25 22 20] 27

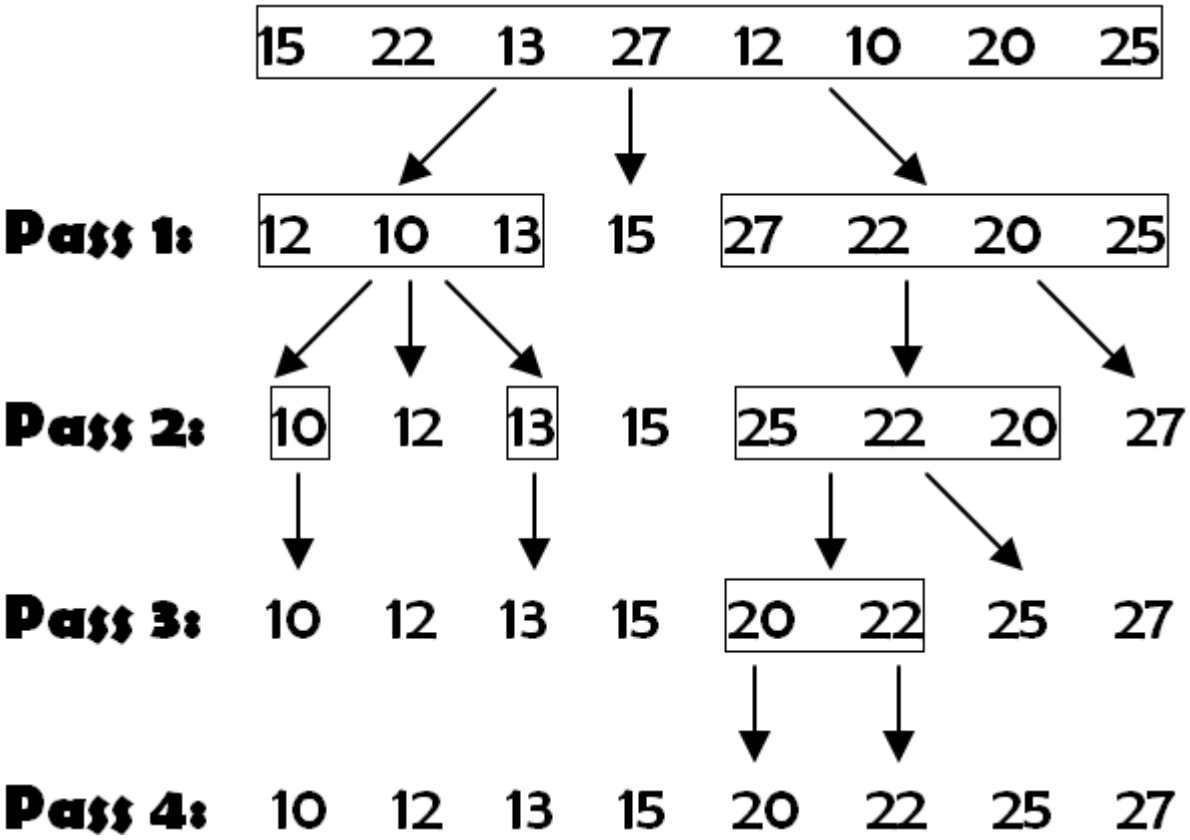
Pass 6: 10 12 13 15 [20 22] 25 27

Pass 7: 10 12 13 15 20 22 25 27

[Note]:

- 只有一顆CPU時:
  - 先排左半部
  - 再排右半部
- 若有多顆CPU，則左右半部可各交由不同的CPU執行!

- 多顆CPU時的運算過程:

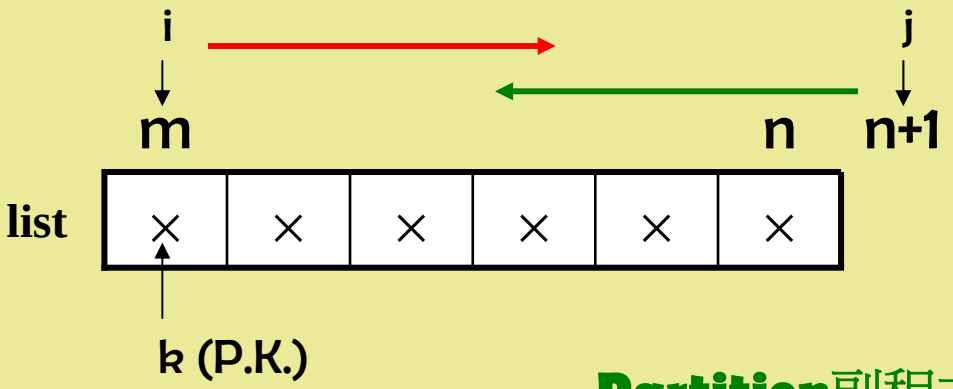


- **Algorithm主要由2個副程式組成:**
  - **Partition副程式**
    - 將記錄串中的第一筆記錄經過運算後，置於該記錄串中 "最正確" 的位置上。
    - 即: **找出P.K.的最正確位置**
  - **Sort副程式 (可當作主程式)**
    - 將Partition後、位於P.K.前後**未排序好**的兩筆記錄串，透過**遞迴**的方式分別執行Partition副程式，使之成爲排序好的記錄

Sort副程式

(可看成主程式)

```
procedure QSort(list[], m, n)    //Sort list[m] ~ list[n]
                                 //Assume "list[m] ≤ list[n+1]"
{
  if (m < n) then
    {
      i = m, j = n+1, k = list[m];
      Repeat
      {
        repeat
          i = i+1;
        until list[i] ≥ k;
        repeat
          j = j-1;
        until list[j] ≤ k;
        if (i < j) then Swap(list[i], list[j]);    //被找到的兩個數字 list[i]與 list[j]做交換
      } until (i ≥ j);
      Swap(list[m], list[j]);    //被找到的數字 list[j]與 P.K.做交換
      QSort(list, m, j-1);    //遞迴處理 P.K.左半部
      QSort(list, j+1, n);    //遞迴處理 P.K.右半部
    };
}
```



Partition副程式

分析

- Time Complexity
  1) Best Case
  2) Worst Case
  3) Average Case
- Space Complexity
- Stable / Unstable

Time-Complexity

- **Best Case: O(n log n)**
  - P.K.之最正確位置恰好將資料量均分成二等份
    - 以Multiprocessor來看，2個CPU的工作量相等，工作可同時做完，沒有誰等誰的問題

[說明]:

左半部 右半部

時間函數: $T(n) = c \times n + T(n/2) + T(n/2)$

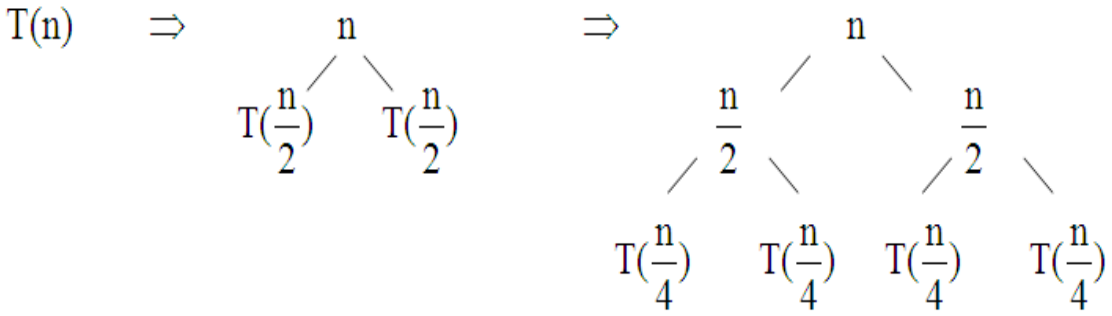
時間複雜度求法:

變數 i 與 j 最多花 n 個執行時間找記錄 (即: 決定P.K.最正確位置所花時間)

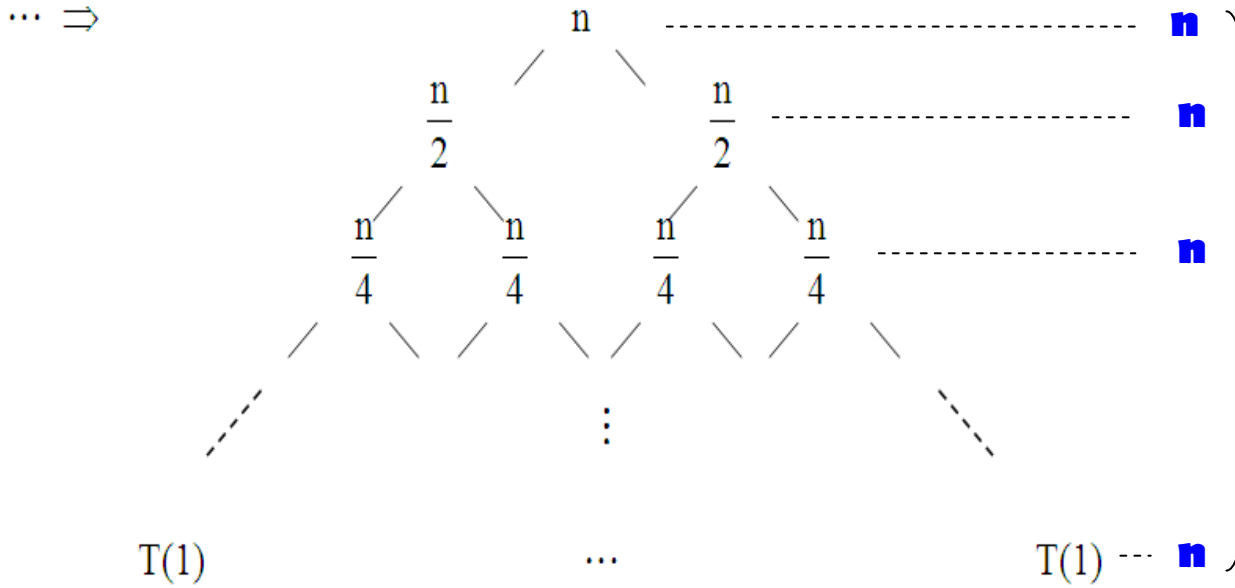
- 遞迴樹
  - 步驟:
    - 將原本問題照遞迴定義展開
    - 計算每一層的Cost
    - 加總每一層的Cost即爲所求
- 數學解法

遞迴樹

Step 1: 展開



Step 2: 計算每一層的Cost



共有$\log_2 n$層，表示有$\log_2 n$個n。

Step 3: total cost = $n \log_2 n$

∴ **T(n) = $O(n \log_2 n)$**

數學解法

時間函數: $T(n) = c \times n + T(n/2) + T(n/2)$

$\Rightarrow T(n) = 2T(n/2) + cn$, c爲 >0 的常數

$\downarrow$ $2(2T(n/4) + c(n/2)) + cn$

$= 4T(n/4) + 2cn$

$\downarrow$ $4(2T(n/8) + c(n/4)) + 2cn$

$= 8T(n/8) + 3cn$

$\downarrow$ $8(2T(n/16) + c(n/8)) + 3cn$

$= 16T(n/16) + 4cn$

$= \ldots$

$= nT(n/n) + \log_2 n \times cn = nT(1) + cn \log_2 n = n + cn \log_2 n$

T(1) = 1

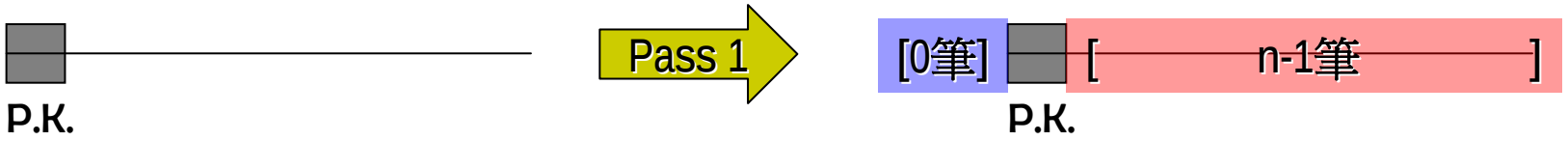
共有$\log_2 n$個計算結果

$\therefore T(n) = O(n \log_2 n)$

- **Worst Case: $O(n^2)$**
  - 當輸入資料是由大到小或由小到大排好時 (切割毫無用處)

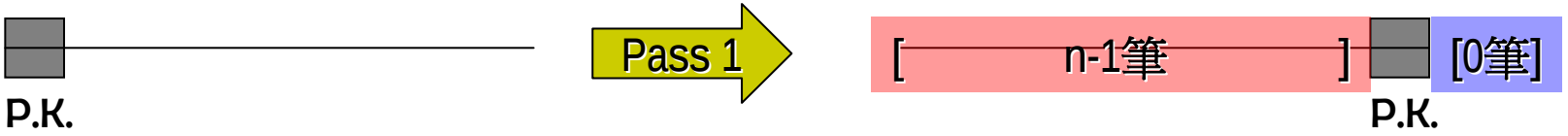
[說明]:

輸入資料: 小 大



或

輸入資料: 大 小



[分析]: 利用遞迴時間函數 (數學解法)

時間函數: $T(n) = c \times n + T(0) + T(n-1)$ (c爲>0常數, T(0)=0, T(1)=1)

$T(n) = T(n-1) + cn$

$= (T(n-2) + c(n-1)) + cn = T(n-2) + c(n-1) + cn$

$= \ldots$

$= T(0) + c\,(1 + 2 + \ldots + (n-3) + (n-2) + (n-1) + n)$

$= c \times n(n+1)/2$

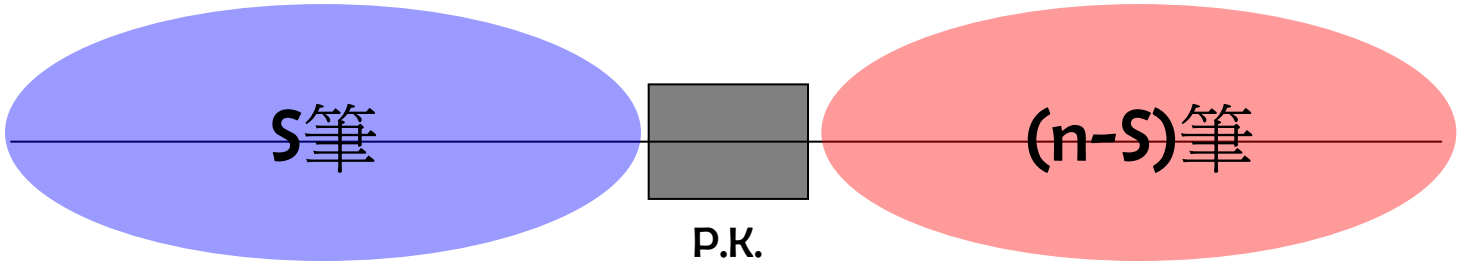
$\therefore T(n) = O(n^2)$

(改善方法在補充 1)

- Average Case: **O(n log n)**

[說明]:

$$\Rightarrow \quad T(n) = \frac{1}{n}\sum_{s=1}^{n}[T(s) + T(n-s)] + cn \qquad , T(0) = 0$$



Space-Complexity

- O(log n) ~ O(n)

[說明]:

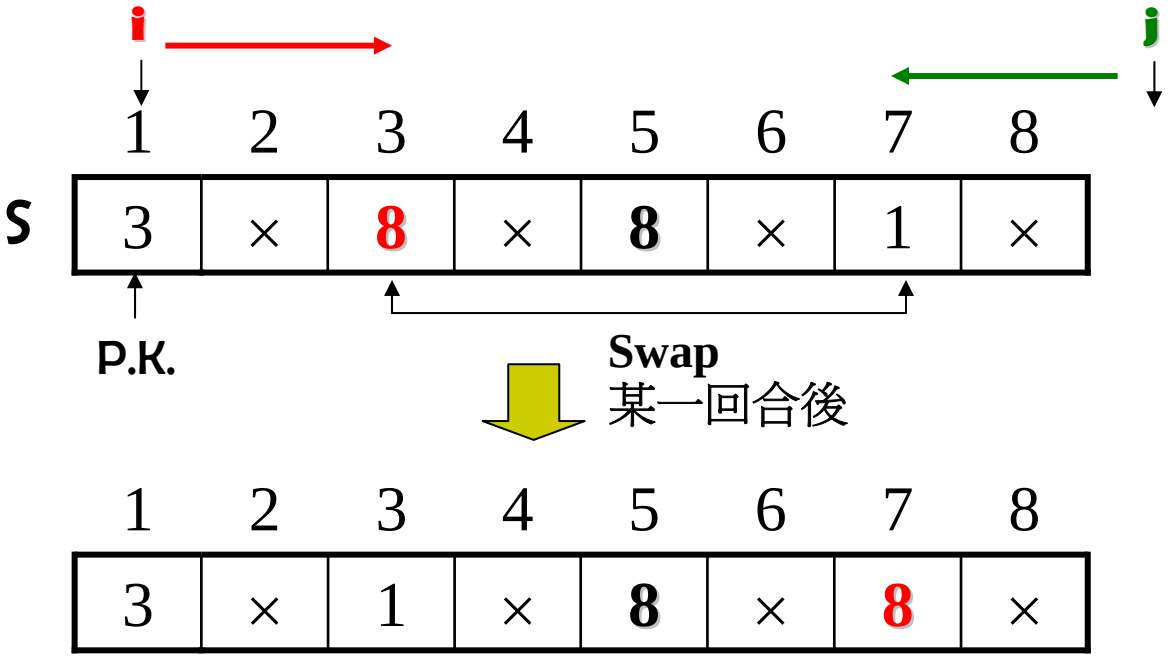
  - 額外的空間需求，來自於遞迴呼叫所需的**Stack空間**
  - 而Stack空間的大小，取決於**遞迴呼叫的深度**

- Best Case: **O(log n)**
  - 由Best Case的時間複雜度分析可以得知，**遞迴呼叫的深度是 log n**，即: 做過 log n 次呼叫後，整體資料量只剩 1 筆，即可停止呼叫

- Worst Case: **O(n)**
  - 由Worst Case的時間複雜度分析可以得知，**遞迴呼叫的深度是(n-1)**，即: 做過(n-1)次呼叫後，資料量只剩 1 筆，即可停止呼叫

Stable / Unstable

- Unstable (不穩定的)
- 說明:



∵相同鍵值的記錄在排序後，其相對位置有改變，亦即有不必要的Swap發生,

∴Unstable

Merge Sort (合併排序)

- 觀念:
  - 將兩個已排序過的記錄合併，而得到另一個排序好的記錄。
- 可分爲兩種類型:
  - **Recursive (遞迴)**
  - **Iterative (迴圈, 非遞迴)**

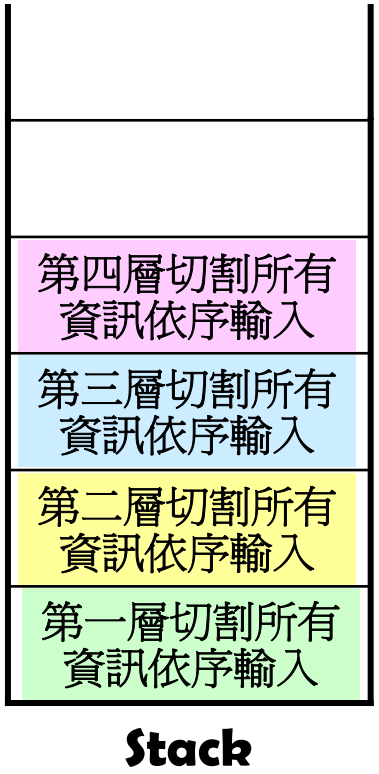
Recursive Merge Sort (遞迴合併排序)

- 將資料量 n 切成 n/2 與 n/2 兩半部，再各自Merge Sort，最後合併兩半部之排序結果即成。

- 切割資料量 n 的公式爲: $\left\lfloor \frac{(low + high)}{2} \right\rfloor$

- [ ]: Run, 已排序好的檔案記錄
- Run的長度: Run中記錄個數

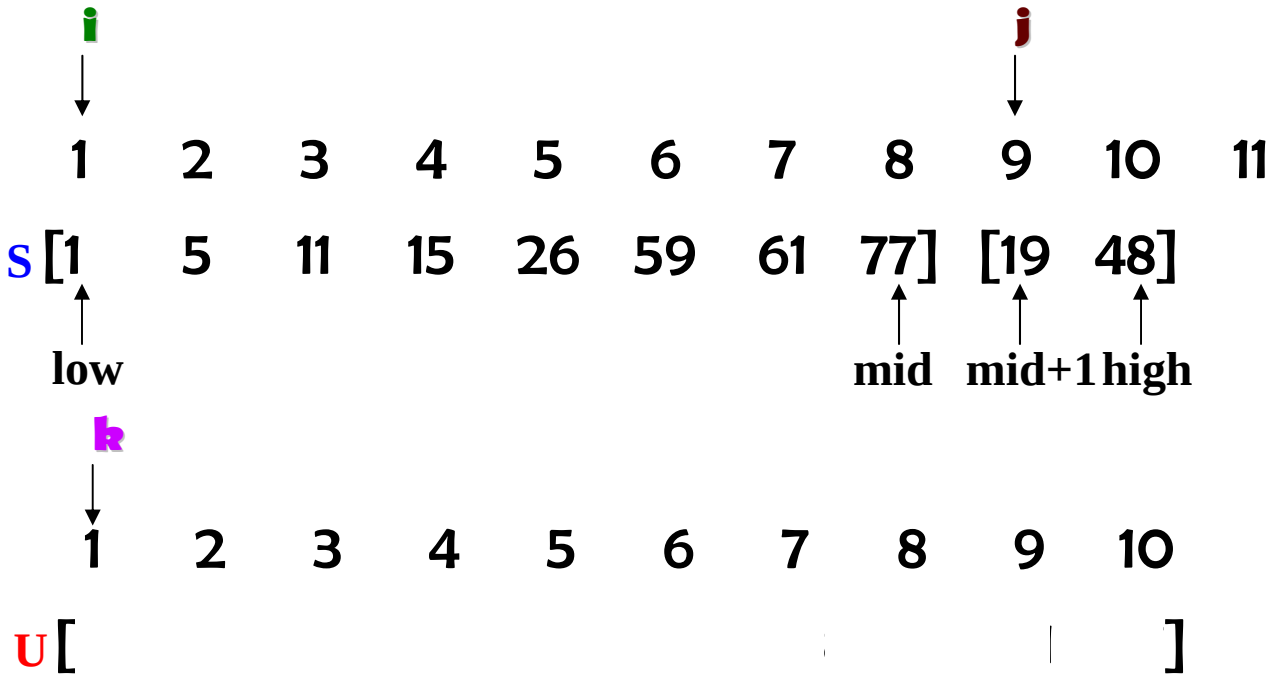
1	2	3	4	5	6	7	8	9	10
26	5	77	1	61	11	59	15	48	19



Merge the two sorted records (Merge副程式)

- 需要三個整數變數 i, j, k.
  - If **S[i] ≤ S[j]** then output S[i], i 前進且 k 前進

    else output S[j], j 前進且 k 前進
  - 當 i > mid 或 j > high 時則停止，並將剩餘記錄output



- 根據以上討論，Algorithm主要由2個副程式組成:
  - Merge2副程式
    - 將兩個Run的記錄 (即: 兩筆已排序的記錄)，合併成一筆已排序的記錄 U (即: 合併成一個Run)
  - MergeSort2副程式 (可當作主程式)
    - 執行整個記錄串的遞迴切割
    - 以堆疊中的遞迴切割次序，透過Merge2將所有的Run加以合併
- Run 1的長度爲 m，Run 2的長度爲 n，則合併兩個Run的最多比較次數爲 m+n-1 次

  Ex: [5,26] [1, 77] 比較3次後，會得到 [1, 5, 26, 77]

Merge2副程式

Algorithm 2.5: Merge 2

Problem: Merge the two sorted subarrays of *S* created in Mergesort 2.

Inputs: indices *low*, *mid*, and *high*, and the subarray of *S* indexed from *low* to *high*. The keys in array slots from *low* to *mid* are already sorted in nondecreasing order, as are the keys in array slots from *mid* + 1 to *high*.

Outputs: the subarray of *S* indexed from *low* to *high* containing the keys in nondecreasing order.

```
void merge2 (index low, index mid, index high)
{
   index i, j, k;
   keytype U[low .. high];      // A local array needed for the
                                // merging
   i = low; j = mid + 1; k = low;
   while (i ≤ mid && j ≤ high){
      if (S[i]S[j]){
      U[k] = S[i];
      i++;
   }
   else{
      U[k] = S[j];
      j++;
   }
      k++;
}
if (i > mid)
   move S[j] through S[high] to U[k] through U[high];
else
   move S[i] through S[mid] to U[k] through U[high];
 move U[low] through U[high] to S[low] through S[high];
}
```

- Run 1的長度爲 m，Run 2的長度爲 n，則合併兩個Run的最多比較次數爲 **m+n-1** 次

MergeSort2副程式

Algorithm 2.4: Mergesort 2

Problem: Sort *n* keys in nondecreasing sequence.

Inputs: positive integer *n*, array of keys *S* indexed from 1 to *n*.

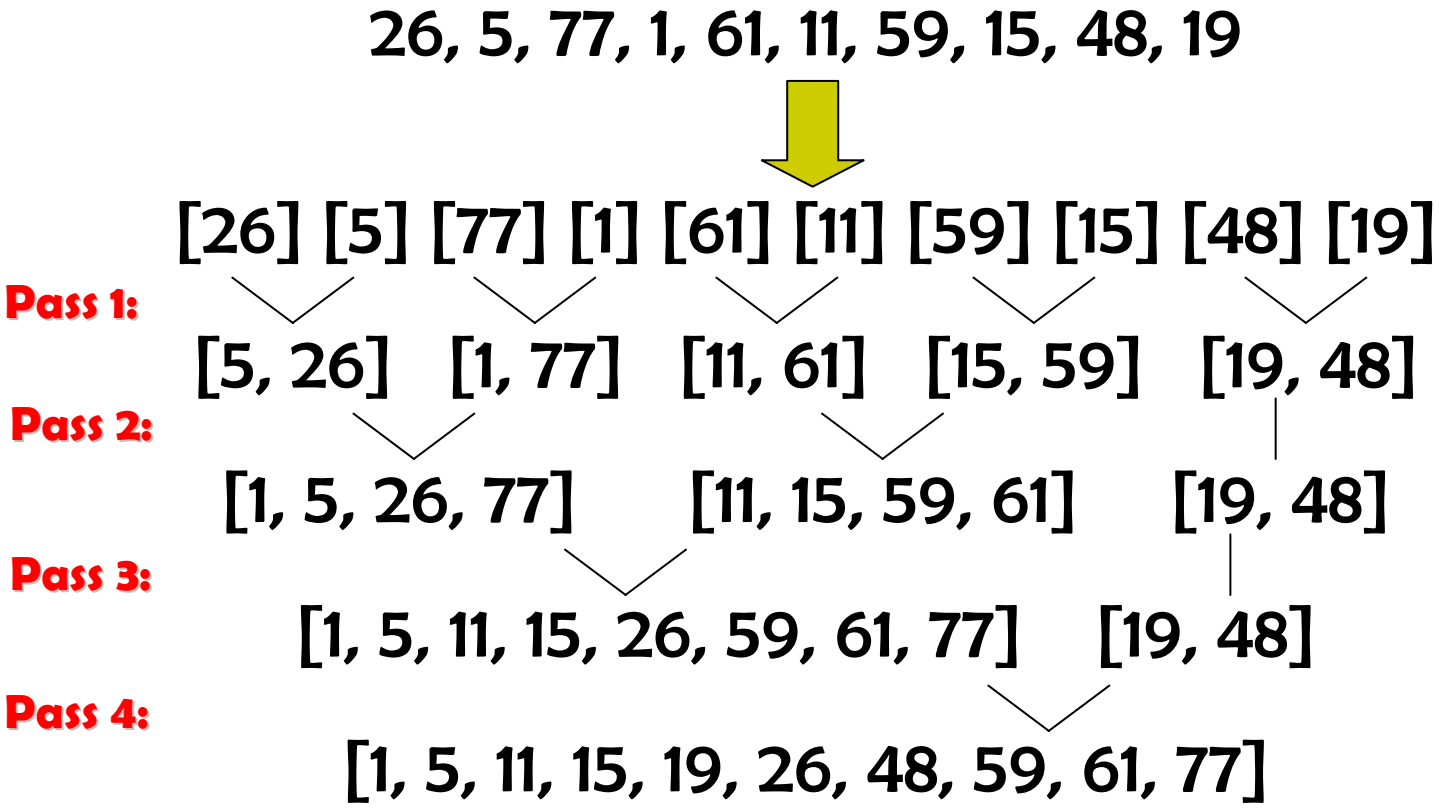
Outputs: the array *S* containing the keys in nondecreasing order.

```
void mergesort2 (index low, index high)
{
    index mid;

    if (low < high) {
        mid = [(low + high)/2];
        mergesort2(low, mid);
        mergesort2(mid + 1, high);
        merge2(low, mid, high);
    }
}
```

Iterative Merge Sort (非遞迴合併排序)

- [ ]: Run, 已排序好的檔案記錄
- Run的長度: Run中記錄個數



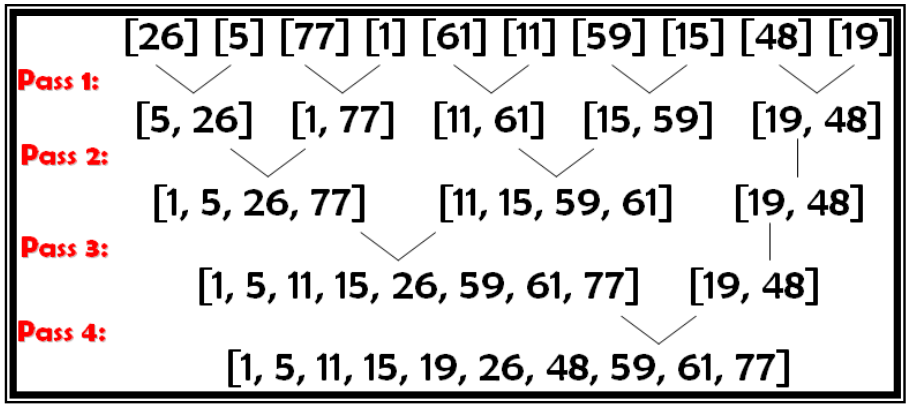
- 根據以上討論，**Algorithm**主要由**3**個副程式組成:
  - **Merge1**副程式
    - 將兩個**Run**的記錄 **(**即: 兩筆已排序的記錄**)**，合併成一筆已排序的記錄 **U (**即: 合併成一個**Run)**
    - 同**Recursion**的作法， **Run 1**的長度爲 **m**，**Run 2**的長度爲 **n**，則合併兩個**Run**的最多比較次數爲 **m+n-1** 次
  - **MergePass**副程式
    - 在每一回合 **(Pass)** 中，會處理多次的 "合併兩個**Run**" 之工作
  - **MergeSort**副程式 **(**可當作主程式**)**
    - 整個非遞迴的合併排序副程式需執行 $\lceil \log_2 n \rceil$ 回合 **(Pass)**
- **(補充 4)**

爲何需要執行 $\lceil \log_2 n \rceil$ 回合？

- 以執行完一回合後，Run的長度來看:
  - 有n個數字待排序，一開始每一個Run的長度爲 1，且有n個Run
    - 執行完Pass 1後，最長的Run長度爲 $2^1 = 2$
    - 執行完Pass 2後，最長的Run長度爲 $2^2 = 4$
    - 執行完Pass 3後，最長的Run長度爲 $2^3 = 8$
    - ...
    - 執行完Pass i 後，最長的Run長度爲 $2^i = n$ (停止)

    嚴格說應該爲 $2^i >= n$

    $\Rightarrow i = \log_2 n$



- 由於 n 不見得爲2的倍數，因此取上限整數 (Ceiling) $\lceil \log_2 n \rceil$ 以求得能真正完整處理n筆資料排序的回合數。

分析

- Time Complexity
  1) Best Case
  2) Worst Case
  3) Average Case
- Space Complexity
- Stable / Unstable

Time-Complexity

- Avg. / Worst / Best Case: **O(n log n)**
- 以Recursive Merge Sort角度:

[說明]:

左半部遞迴 右半部遞迴

時間函數: T(n) = T(n/2) + T(n/2) + c×n

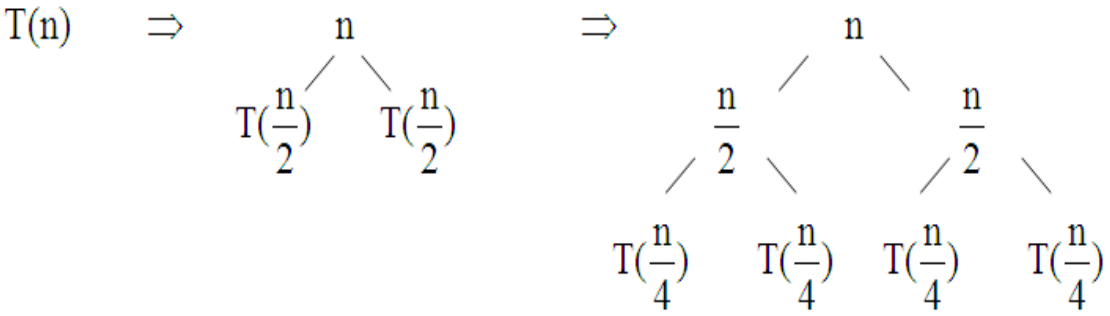
時間複雜度求法:

- 遞迴樹
  - 步驟:
    - 將原本問題照遞迴定義展開
    - 計算每一層的Cost
    - 加總每一層的Cost即爲所求
- 數學解法

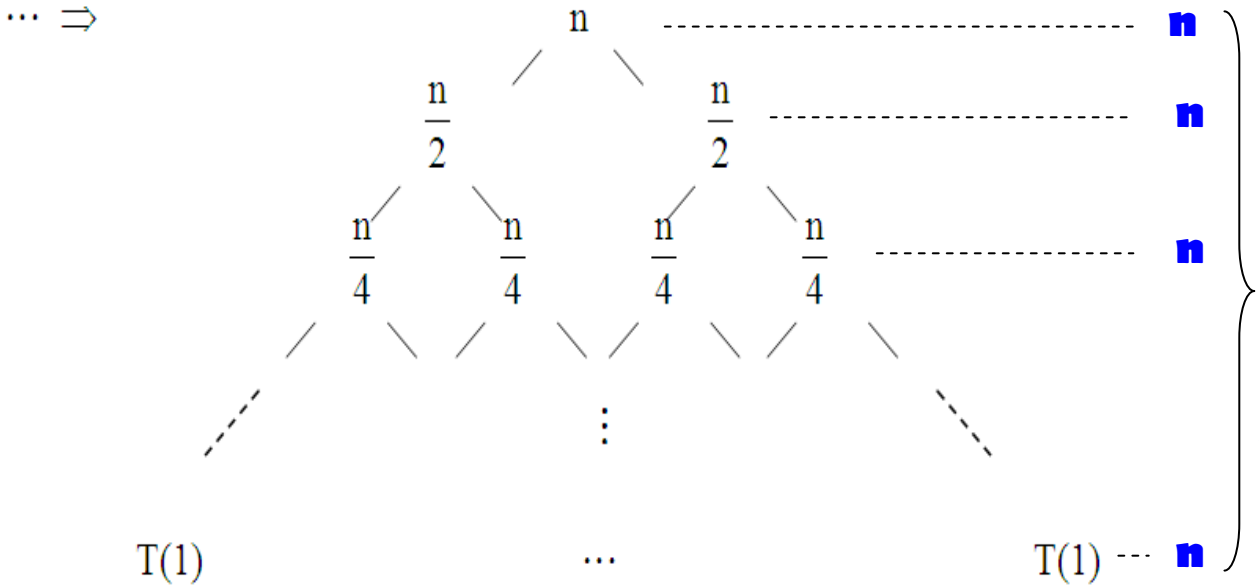
- 最後合併左右兩半部所花時間
  - ∵ 左、右半部排好之後，各只剩一個Run，且兩半部各有n/2的資料量，其最後一次合併時的比較次數"最多"爲 n/2 + n/2 -1 次，即約 n-1 次 (slide 72)
  - ∴時間的表示可爲 c×n 次(∵爲線性時間))

遞迴樹

Step 1: 展開



Step 2: 計算每一層的Cost



共有$\log_2 n$層，表示有$\log_2 n$個n。

Step 3: total cost = $n \log_2 n$

∴ $T(n) = O(n \log_2 n)$

數學解法

時間函數: $T(n) = c \times n + T(n/2) + T(n/2)$

$\Rightarrow T(n) = 2T(n/2) + cn$, c爲 >0 的常數

$\downarrow$ $2(2T(n/4) + c(n/2)) + cn$

$= 4T(n/4) + 2cn$

$\downarrow$ $4(2T(n/8) + c(n/4)) + 2cn$

$= 8T(n/8) + 3cn$

$\downarrow$ $8(2T(n/16) + c(n/8)) + 3cn$

$= 16T(n/16) + 4cn$

$= \ldots$

$= nT(n/n) + \log_2 n \times cn = nT(1) + cn \log_2 n = n + cn \log_2 n$

T(1) = 1

共有$\log_2 n$個計算結果

$\therefore T(n) = O(n \log_2 n)$

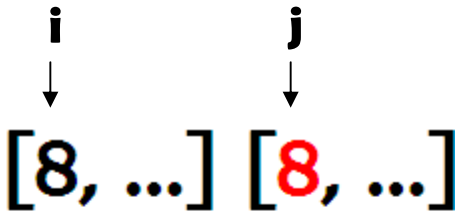
- 以 Iterative Merge Sort 角度:
  - 排序 n 個資料，需花費 $\lceil \log_2 n \rceil$ 回合，且每一回合需花費 n+m-1 = O(n) 時間做Merge
    - (不論哪一回合，merge的時間都是與資料量呈線性變化)
  - ∴ 總共花 **O(n log n)**

Space-Complexity

- 不論是遞迴或是非遞迴方式，都需要**暫時性的陣列空間**，目的是用來暫存每回合Merge後的Run之結果。
- n 愈大，Merge所需的暫存空間就愈多，因此額外的空間需求與 n 成正比。
- ∴ **O(n)**

Stable / Unstable

- Stable (穩定的)
- 說明:



∵ 8 ≤ 8，
∴先output 8，之後再輸出 8

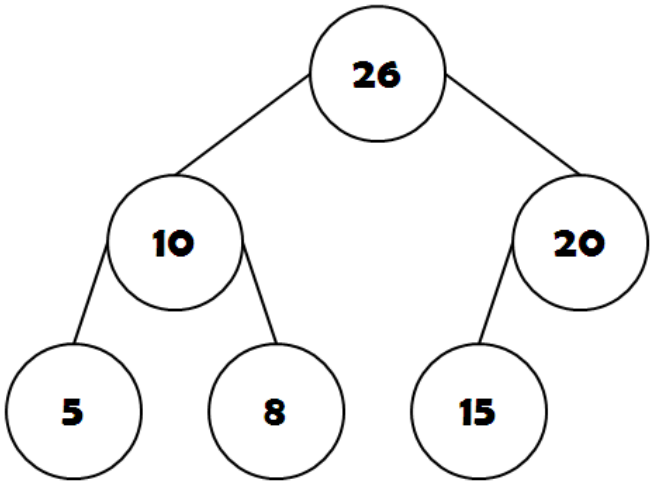
∵相同鍵值的記錄在排序後，
其相對位置沒有改變，亦即
沒有不必要的Swap發生，
∴Stable

Heap Sort (堆積排序)

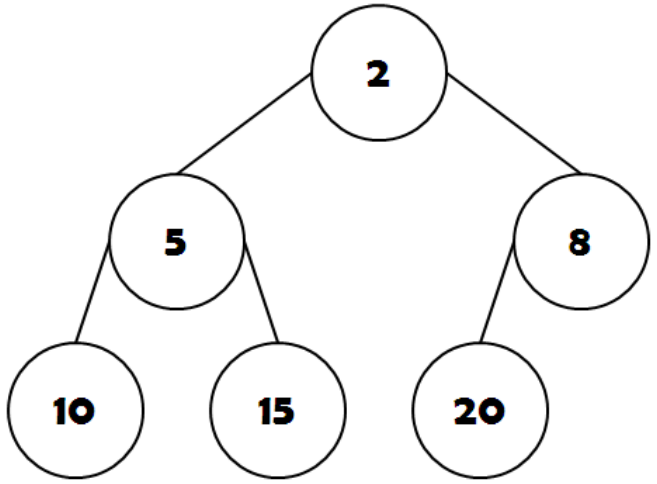
- Heap (堆積)
  - 種類
  - 相關的操作與分析
    - Insert
    - Delete
  - Heap的建立方式
    - Top-Down
    - Bottom-Up (課本所提之Siftdown)
- Heap Sort

※ Heap (堆積)

- 可分爲Max-Heap和Min-Heap
- Max-Heap
  - Def: 爲**Complete Binary Tree**，若不爲空，則滿足
    1. 所有父點的鍵值 ≥ 子點鍵值
    2. Root具有最大鍵值



- Min-Heap
  - Def: 爲**Complete Binary Tree**，若不爲空，則滿足
    1. 所有父點的鍵值 ≤ 子點鍵值
    2. Root具有最小鍵值



- Heap提供下列運作:
  - **Insert** element
  - **Delete** Max. (or Min.) element --兩者擇其一
- 以下講解皆以Max-Heap爲例

Heap之Insert x 動作

Step ①: 將 x 置於Last Node之後

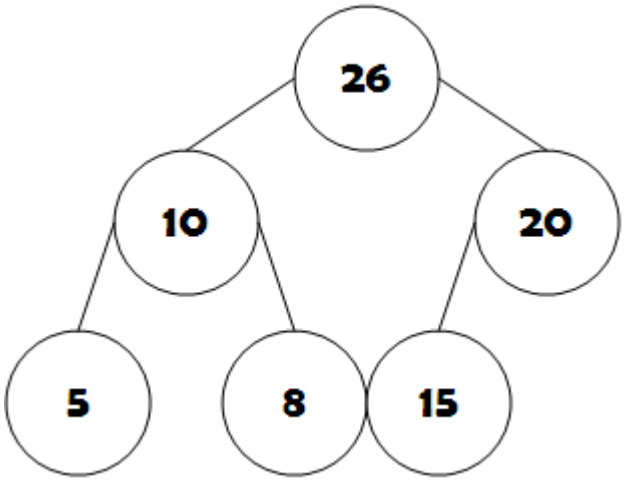
Step ②: x 向上挑戰父點 (即: 與父點的值比大小)，直到發生下列任一狀況爲止:

- 挑戰失敗
- 無父點

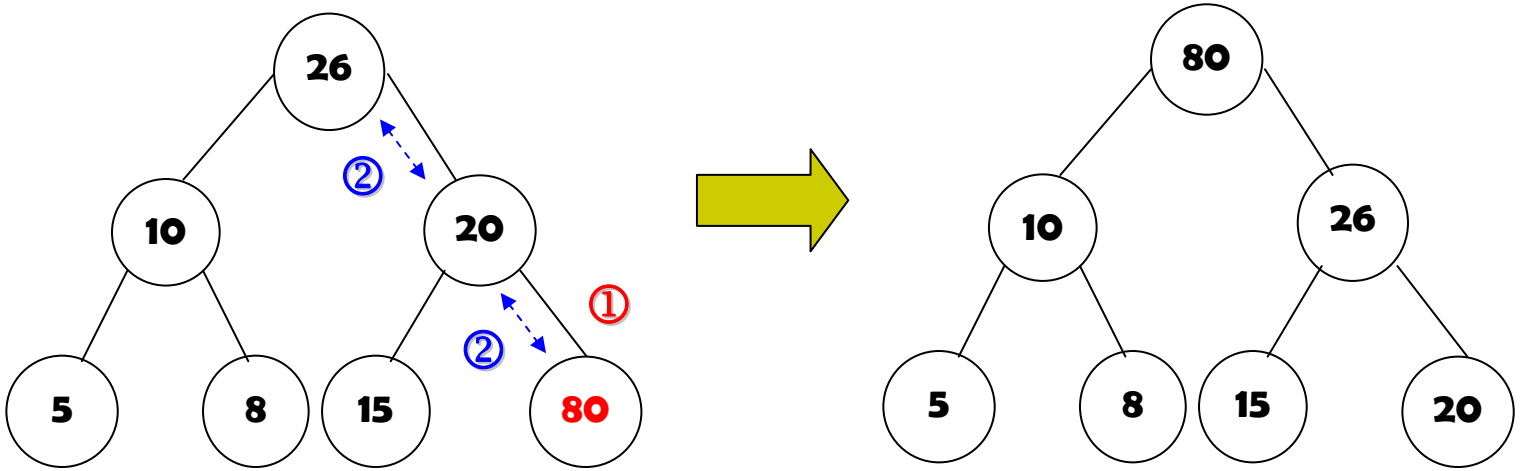
例: Max-Heap如下，試討論執行下列動作後之結果爲何?

(1) 插入 "80"

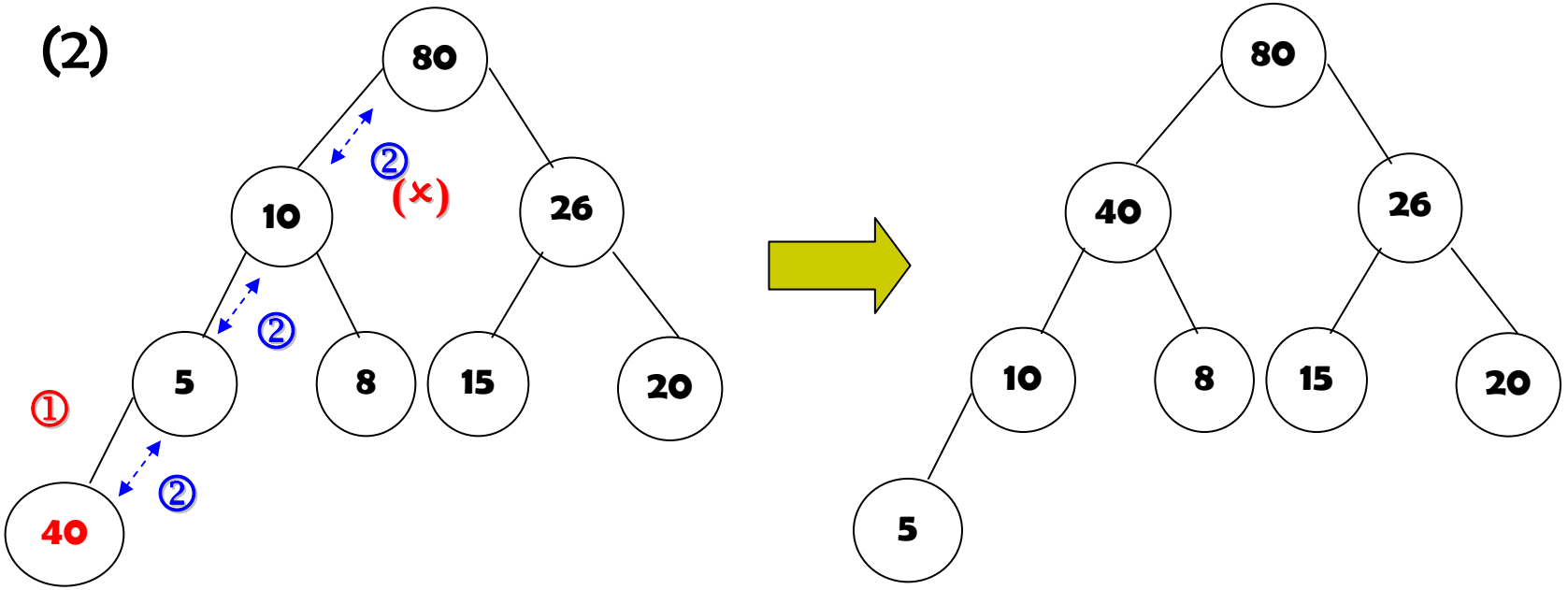
(2) 插入 "40"



Sol: (1)


26
10
20
5
8
15
80
②
②
①
80
10
26
5
8
15
20

(2)


80
10
26
5
8
15
20
40
②
(×)
②
①
②
80
40
26
10
8
15
20
5

Insert x 的Time之分析: O(log n)

- 說明:
  - Insert x後，x的最大移動距離爲 "**從leaf到root**"，即爲**樹高**，又Heap爲**Complete B.T.**
  - ∴當有n個nodes時，樹高爲: $\lceil \log_2(n+1) \rceil$

⇨Insert之Time爲O(log n)

Heap之Delete Max 動作

- 最大鍵值必定位於**Root**

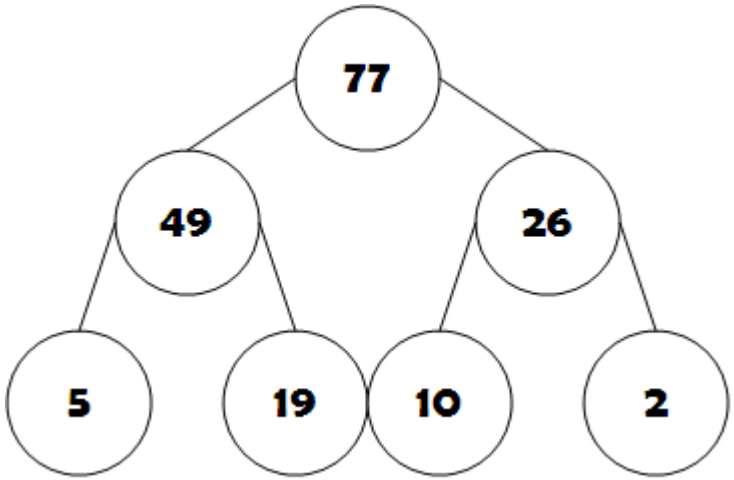
Step ①: 移走Root的元素

Step ②: 將Last Node x刪除並置於Root位置

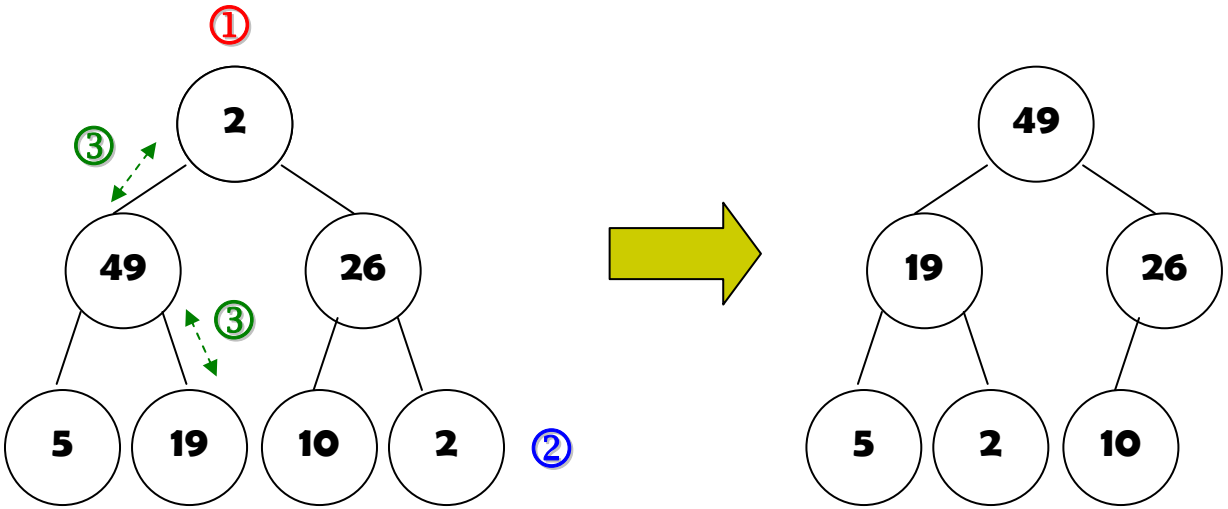
Step ③: 從Root往下調整成Max-Heap

(Max-Heap調整方法: 跟較大的兒子交換)

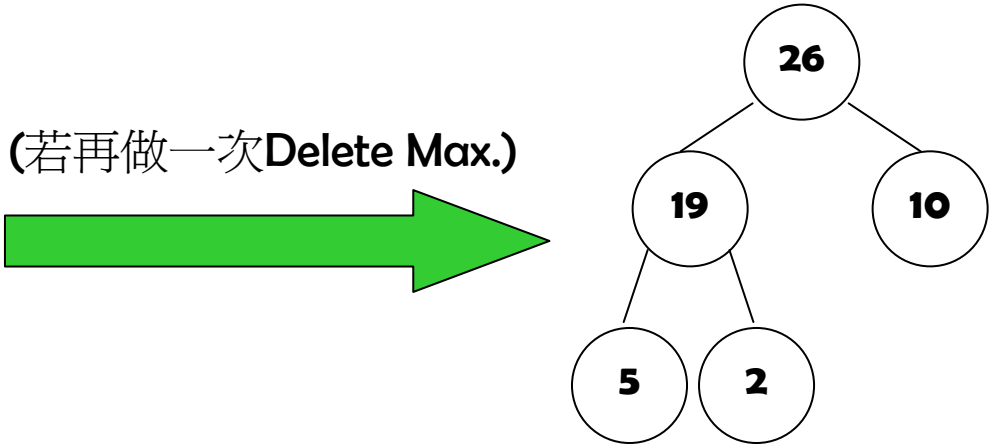
例: Max-Heap如下，試討論執行 Delete Max. 後之結果為何?



Sol:



(若再做一次Delete Max.)



Delete Max 的Time之分析: O(log n)

- 說明:
  - Step ①與Step ②的動作只花 O(1) (常數時間)
  - Step ③花費時間較多，故以此爲主
    - Last Node x的最大移動距離爲 "從Root到leaf"，即爲樹高，又Heap爲Complete B.T.
    - ∴當有n個nodes時，樹高爲: $\lceil \log_2(n+1) \rceil$

⇨Delete x之Time爲O(log n)

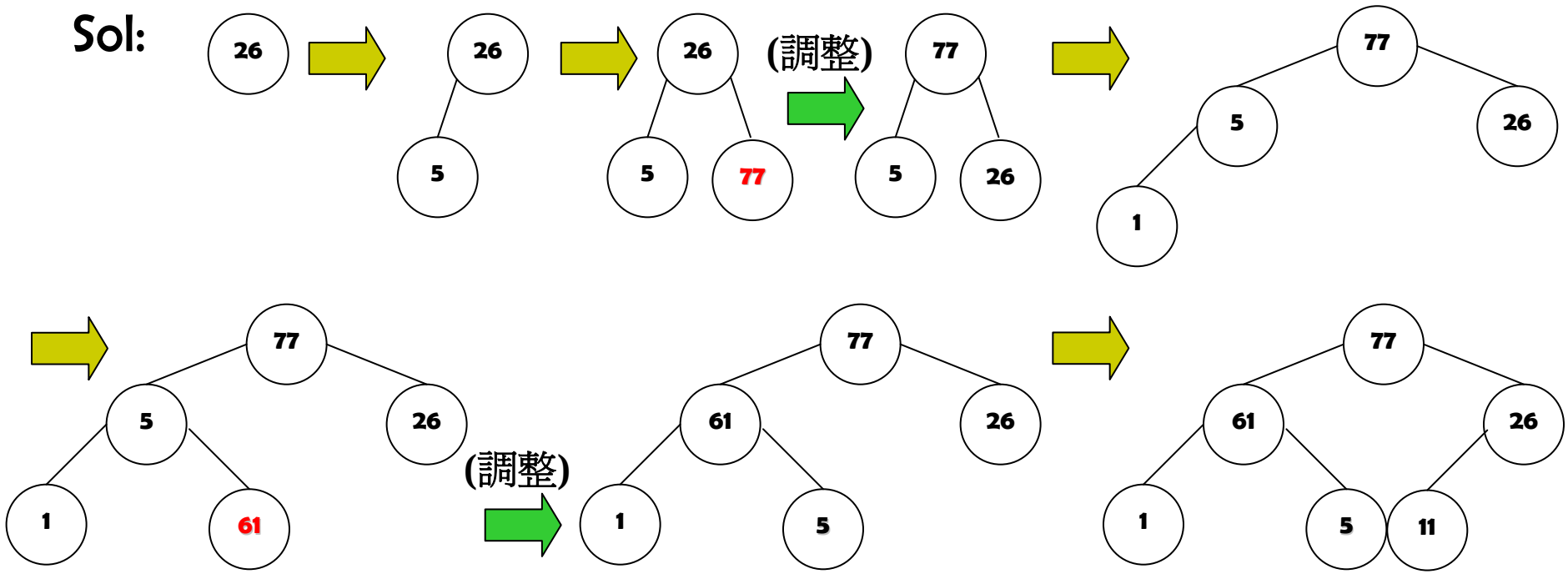
Heap的建立方式

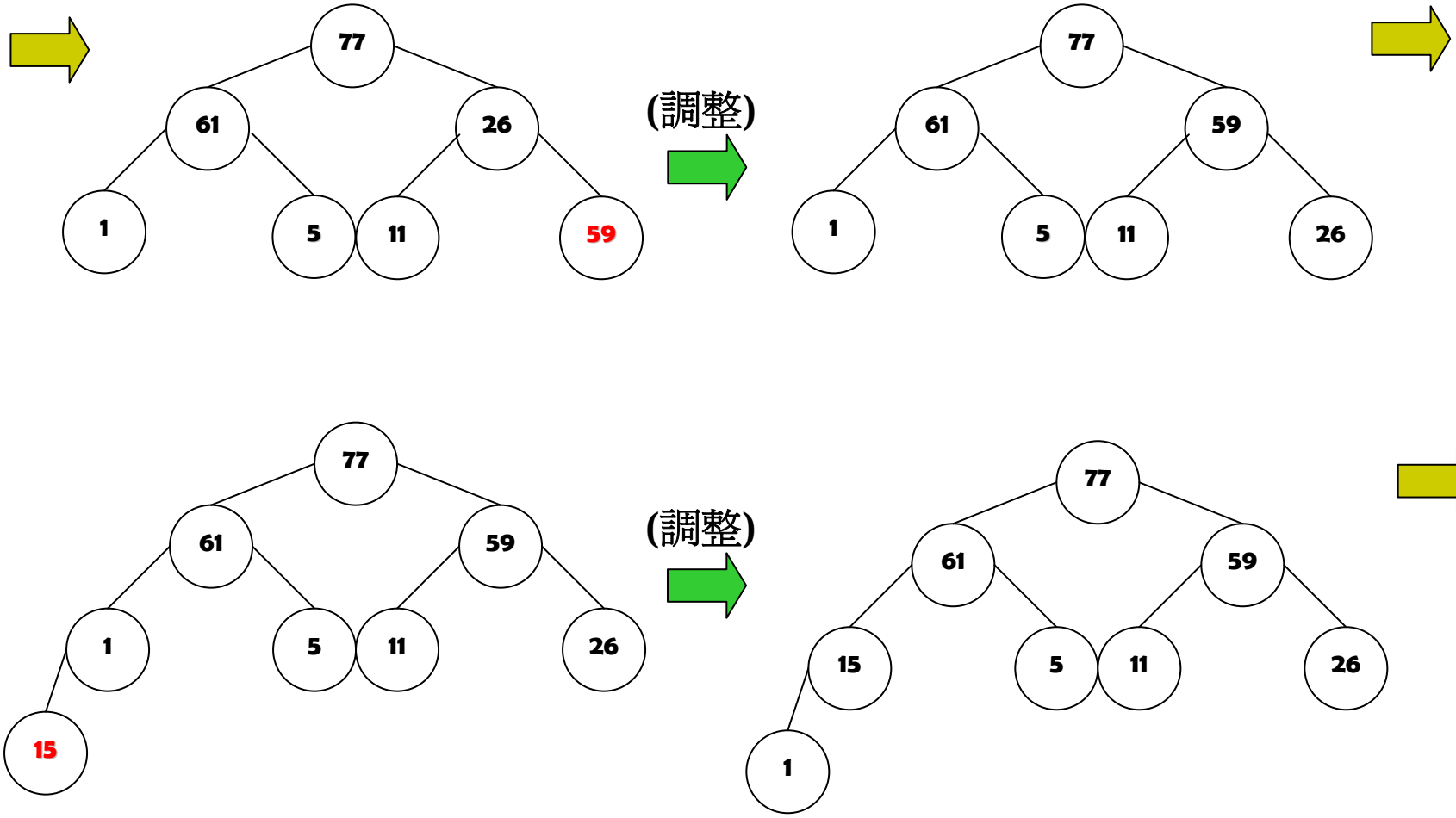
- 以演算法的角度來說，分爲兩種方式:
  - Top-Down
  - Bottom-Up (即課本所討論之Siftdown)

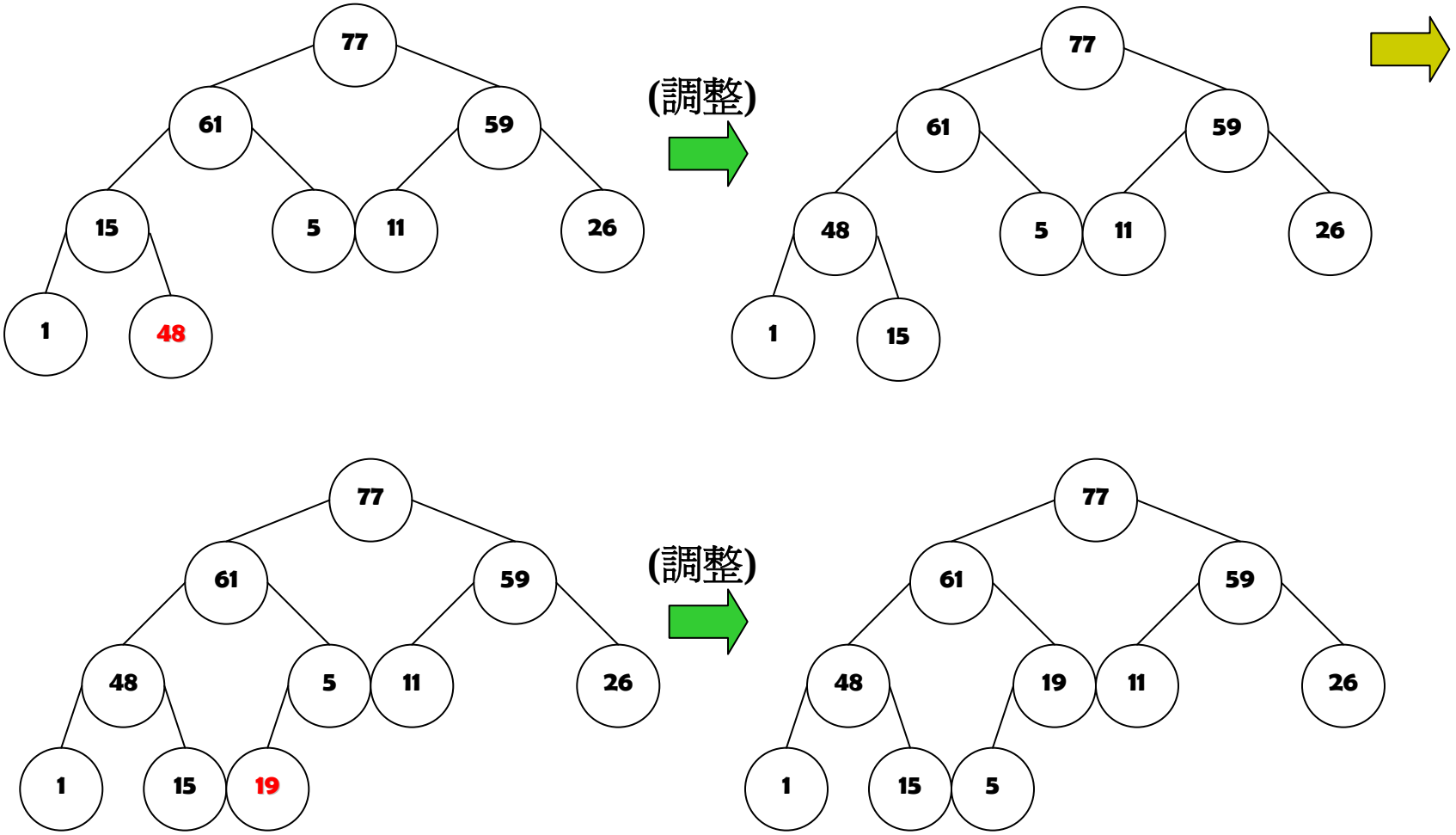
Top-Down

- 連續執行 "Insert" 的動作，每一個步驟執行後均維持Max-Heap
- 例: 給予 26, 5, 77, 1, 61, 11, 59, 15, 48, 19以Top-Down的方式建立Heap。

Sol:



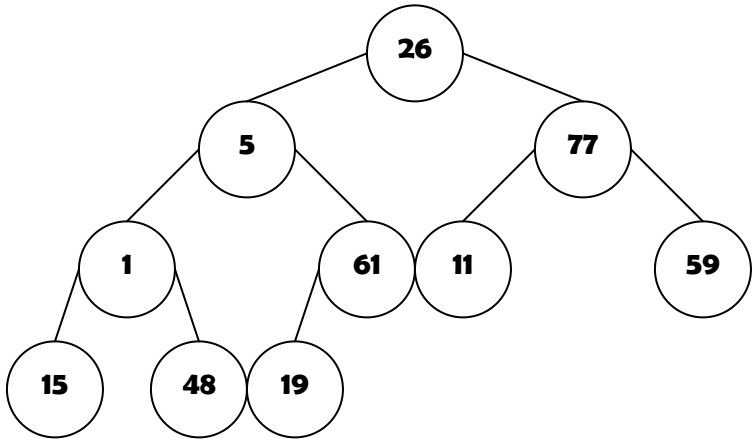

77
61
26
1
5
11
59
(調整)
77
61
59
1
5
11
26
77
61
59
1
5
11
26
15
(調整)
77
61
59
15
5
11
26
1


77
61
59
15
5
11
26
1
48
(調整)
77
61
59
48
5
11
26
1
15
77
61
59
48
5
11
26
1
15
19
(調整)
77
61
59
48
19
11
26
1
15
5

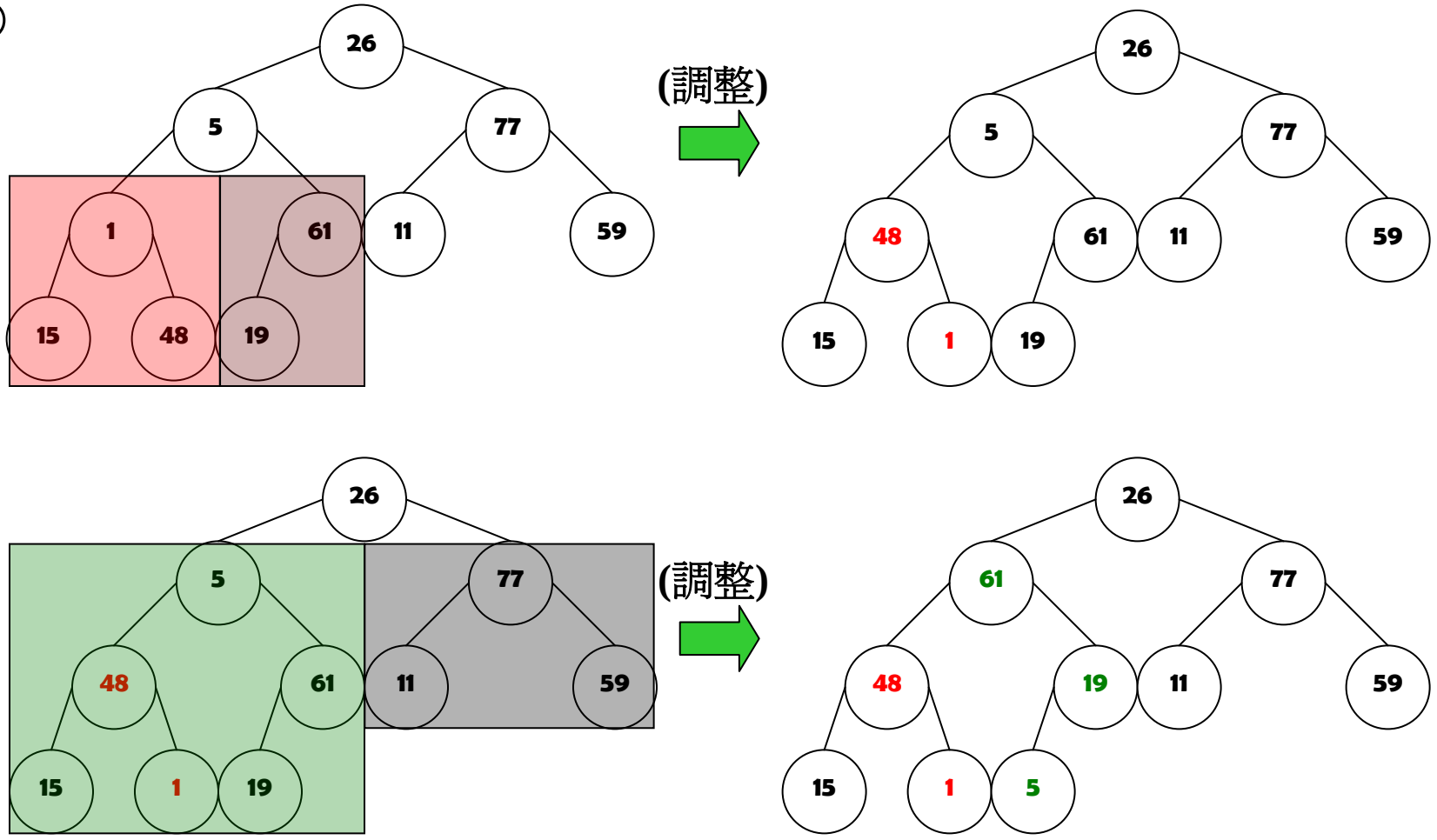
Bottom-Up

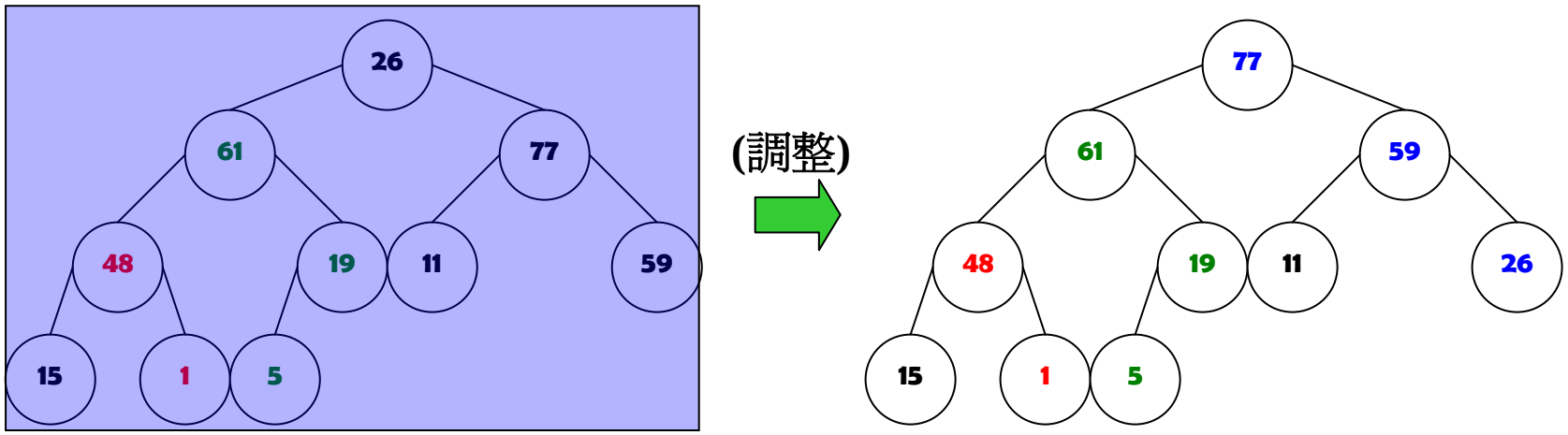
- Step:
  - ① 先將資料建成**Complete B.T.**
  - ② 從 "Last Parent" 往 "Root" 方向，逐次調整每棵子樹成爲Max-Heap
- Stpe ①之所以將之建成Complete B.T.，是因爲真正在寫程式時，可用**Array**儲存，會**較易搜尋子節點及父節點**。**(Course 0的Slide48)**
- 例: 給予 26, 5, 77, 1, 61, 11, 59, 15, 48, 19以Top-Down的方式建立Heap。

Sol: ①



②


26
5
77
1
61
11
59
15
48
19
(調整)
26
5
77
48
61
11
59
15
1
19
26
5
77
48
61
11
59
15
1
19
(調整)
26
61
77
48
19
11
59
15
1
5



- 同樣資料，用Top-Down及Bottom-up所建立出來的Max-Heap不一定相同。
- 通常Bottom-up的實際執行速度較快!! (但兩者的Time Complexity相同)

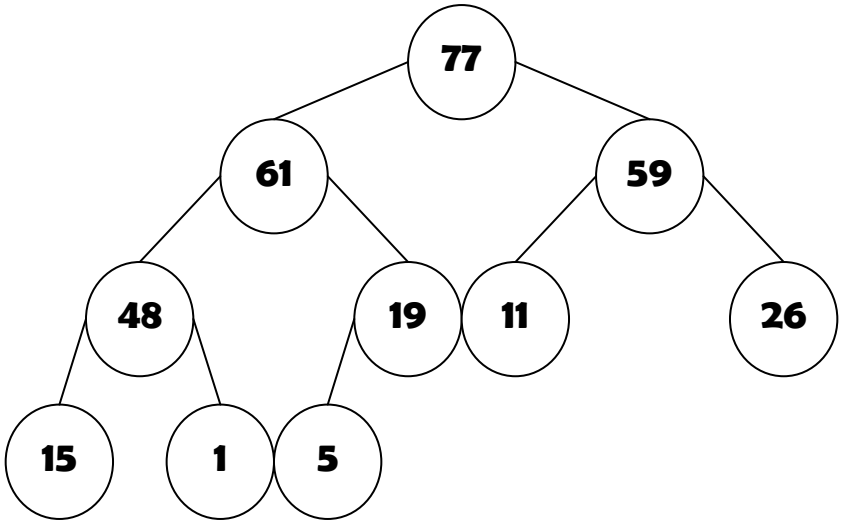
Heap操作	Time Complexity
Insert x	O(log n)
Delete Max	O(log n)
Search for Max	O(1)
建立Heap (n筆資料)	O(n) (補充 3)

※ Heap Sort

- Step:
  - ① 將資料先以 "**Bottom-up**" 的方式建立Max-Heap
  - ② 執行 n-1 回合的 "**Delete Max.**" 動作
- 給予 26, 5, 77, 1, 61, 11, 59, 15, 48, 19，寫出Heap Sort的過程

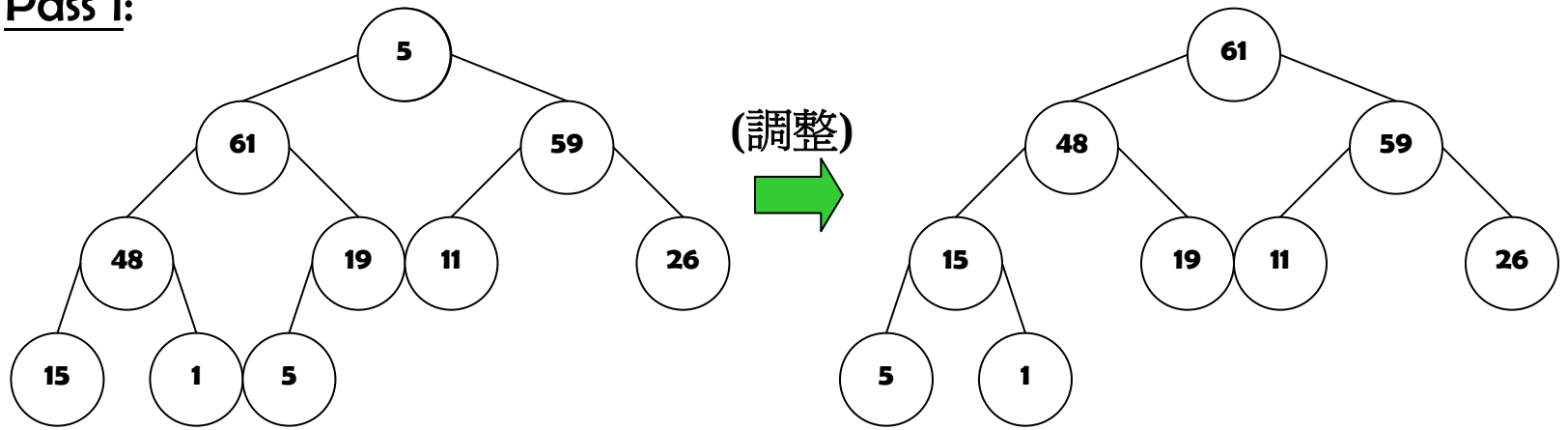
Sol:

①先以 "Bottom-up" 的方式建立Max-Heap

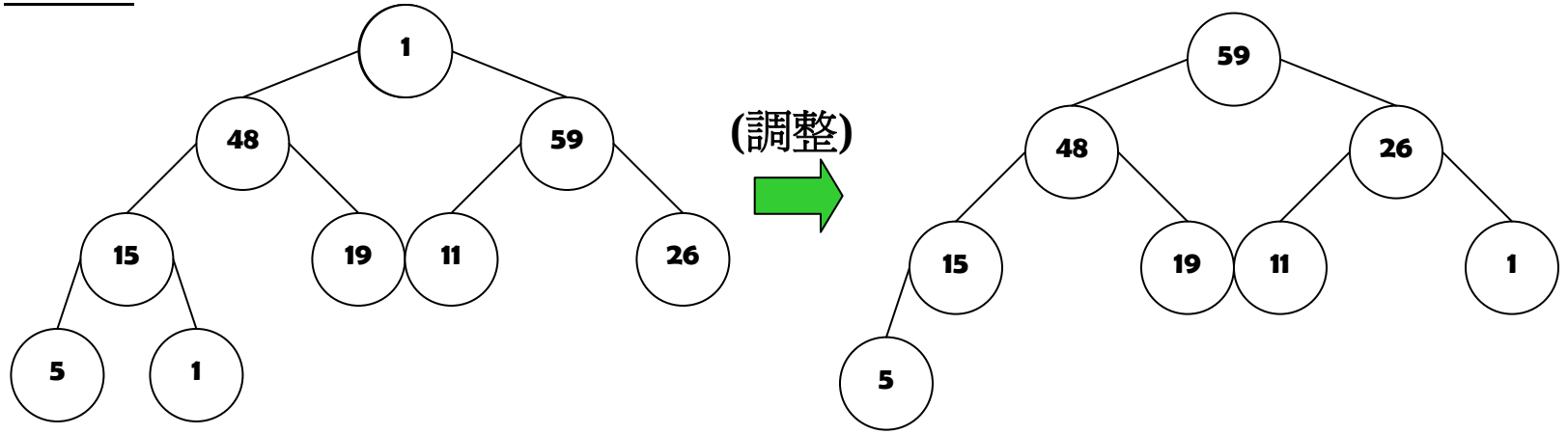


② 執行 n-1 回合的 "Delete Max." 動作

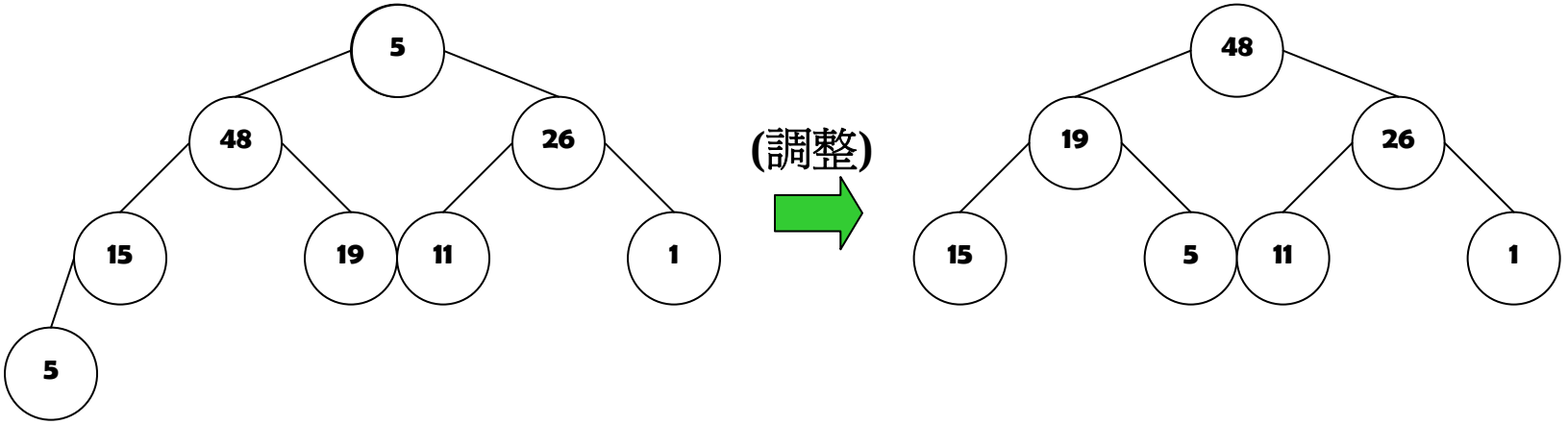
Pass 1:



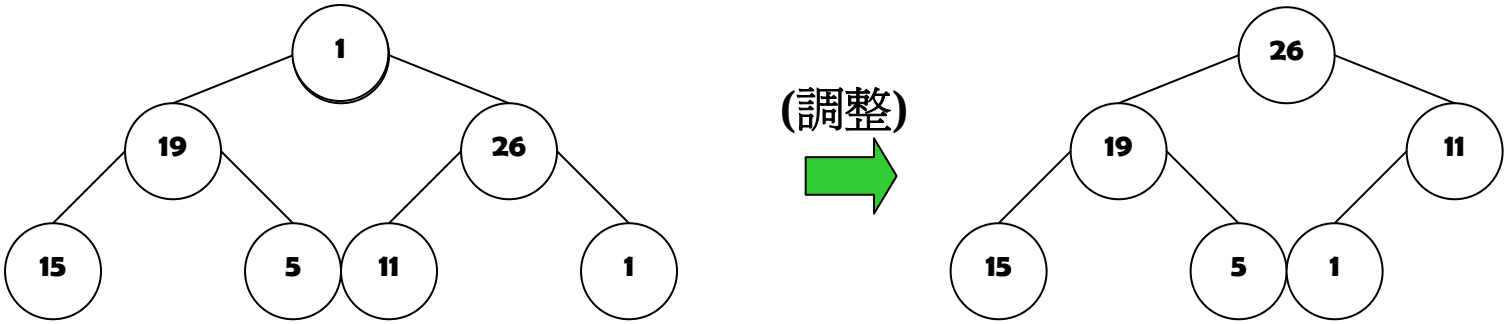
Pass 2: 77



Pass 3: 77, 61



Pass 4: 77, 61, 59



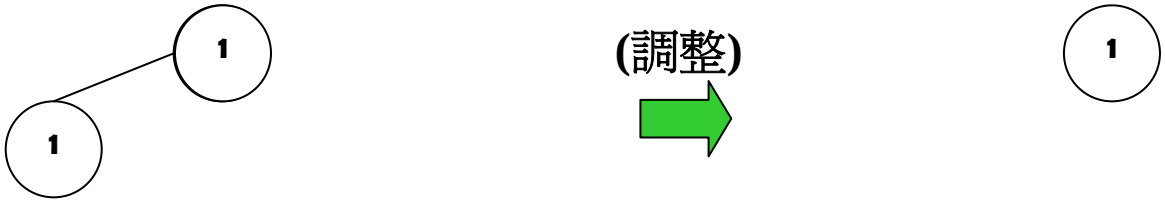
Pass 5: 77, 61, 59, 48

Pass 6: 77, 61, 59, 48, 26

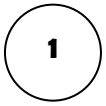
Pass 7: 77, 61, 59, 48, 26, 19

Pass 8: 77, 61, 59, 48, 26, 19, 15

Pass 9: 77, 61, 59, 48, 26 , 19, 15, 11



Pass 10: 77, 61, 59, 48, 26, 19, 15, 11, 5



- **How to 小⇨大**
  - 將Max-Heap Sort的結果push到一個stack，最後再pop。
  - 使用Min-Heap Sort輸出即是。

分析

- Time Complexity
  1) Best Case
  2) Worst Case
  3) Average Case
- Space Complexity
- Stable / Unstable

Time-Complexity

- Avg. / Worst / Best Case: **O(n log n)**
- 以Heap Sort的執行步驟 (Algorithm) 來說明:
  - Step:

    ① 將資料先以 "**Bottom-up**" 的方式建立Max-Heap

    ② 執行 n-1 回合的 "**Delete Max.**" 動作
  - Step ①: 建立Max-Heap會花費 **O(n)** 時間
  - Step ②: 需執行 (n-1) 回合的 Delete Max 動作，而每一次的 Delete Max 動作需花費 O(log n) 時間

    ⇨ Step ② 共花費 **O(n log n)**
- ∴ 整個Heap-Sort 花費 O(n) + O(n log n) ≅ **O(n log n)** 時間

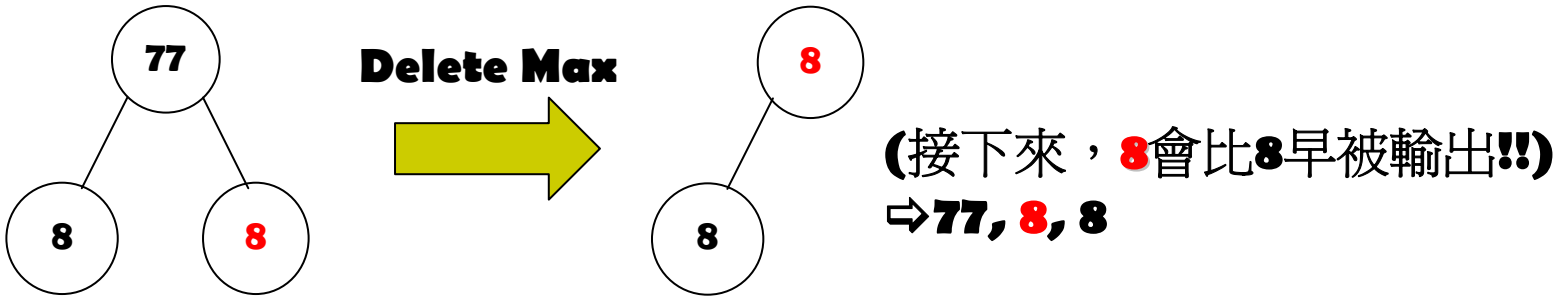
Space-Complexity

- 主程式有一個Simple variable (一般變數) 與Structure variable (即: **Array**，存放構成**Heap**的**Complete Binary Tree**)
- 由以上分析，可以得知:
  - S(P) = C + SP(I)

    = C + 0 (或一常數)
  - 因此，除了存放輸入資料之外，額外的空間需求(Extra space)是固定的。
    - The algorithm is called an ***in-place sort (原地置換)***. --額外的空間需求不會隨著要被排序的資料個數n而增加。
- ∴ Space Complexity: **Θ(1)** (或Θ(C), C爲一常數)

Stable / Unstable

- Unstable (不穩定的)
- 說明:
  - 有一組資料: 8, 8, 77, 其Max-Heap如下。若進行Heap Sort時，執行一回合的Delete Max:



(接下來，8會比8早被輸出!!)
⇨77, 8, 8

∵相同鍵值的記錄在排序後，其相對位置有改變，亦即有不必要的Swap發生,
∴Unstable

- 皆採用 **Comparison & Swap** 技巧
  - 即: 利用鍵值 (Key) 來與欲排序的數字做比較，合乎某種條件就將Key與被比較的數字做交換的動作

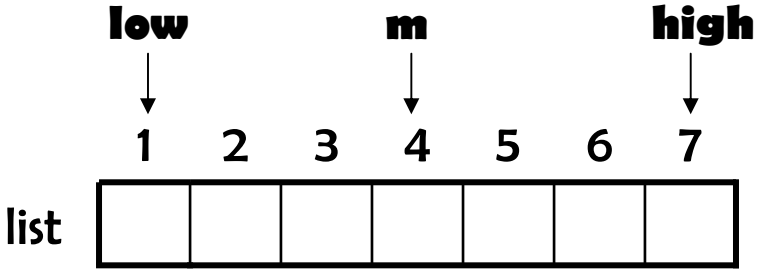
	Time Complexity				
	Best	Worst	Avg.	Space Complexity	Stable/Unstable
Insert Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Stable
Select Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	Unstable
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Stable
Quick Sort	O(n log n)	$O(n^2)$	O(n log n)	O(log n) ~ O(n)	Unstable
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Stable
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	Unstable

  - 已有科學家証明，若採用此技巧所開發出來的演算法，**θ(n log n)** 的時間已是最好的，不會出現有比該時間更有效率的演算法。**(補充 2)**

補　充

補 1: 改善Quick Sort在Worst Case下的執行時間

- 避免挑到最小值或最大值作爲Pivot Key
- 作法: 使用 "middle-of-three"
- 假設:

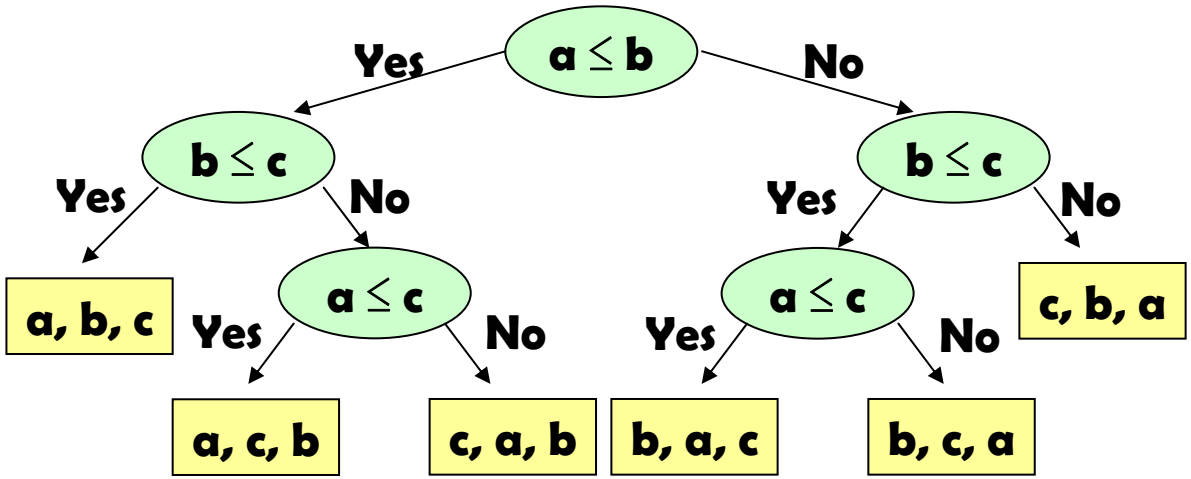


  - 步驟:
    1) m = [(low+high)/2]
    2) 找出 list[low], list[m], list[high]這三筆記錄的中間鍵值 (即: 誰第二大)
    3) 將此筆記錄與list[low]交換
    4) Apply "Quick Sort"
- 可保証第一筆記錄絕對不是最小值或最大值

補 2: 排序方法能達到多快?

- 假設排序方法的設計是採用 **"Comparison & Swap"** 技巧
- 利用**決策樹 (Decision Tree)** 來判斷:
  - Decision Tree: 描述Sort過程中，各種狀況的比較過程
    - **Non-leaf Node**: 表示 "Comparison"
    - 左、右分枝: 表示 "Yes" or "No"
    - **Leaf**: 排序結果
- 例: 試說明3個資料 a，b，c 排序之Decision Tree.

Sol:

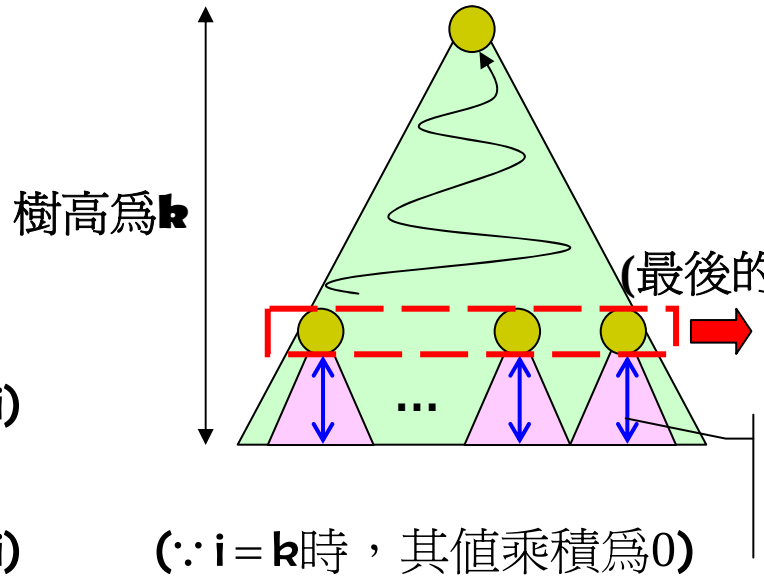


[說明]:

- n 個資料做Sort，有 n! 個可能的排序結果。因此，Sort 的 Decision Tree有 **n! 個 Leaf nodes**。
- 根據二元樹之三個基本定理的 [定理一]，我們可以知道 $2^{i-1} = n! \Rightarrow i-1 = \lceil \log_2 n! \rceil$，$\therefore i = \lceil \log_2 n! \rceil + 1$，表示此Tree的高度至少爲 $\lceil \log_2 n! \rceil + 1$ 。
- 比較次數爲 $\geq \lceil \log_2 n! \rceil$
- 又 $n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$ ，**(Course 1)**

$$\therefore \lceil \log_2 n! \rceil \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = (n/2) \log (n/2) \leq O(n \log n)$$

補 3: 建立Heap花費 O(n) 時間



$$\text{Time} = \sum_{i=1}^{k} 2^{i-1}(k-i)$$

$$= \sum_{i=1}^{k-1} 2^{i-1}(k-i) \qquad (\because i=k\text{時，其值乘積爲}0)$$

$$= \sum_{i=1}^{k-1} \frac{2^i}{2}(k-i)$$

$$= \sum_{i=1}^{k-1} \left(\frac{2^i \times k}{2} - \frac{2^i \times i}{2}\right)$$

$$\le n\sum_{i=1}^{k-1} \frac{i}{2} \qquad (\because k = \lceil \log_2(n+1) \rceil)$$

$$< 2n$$

⇨O(n)

補 4: Iterative Merge Sort 的演算法說明

- Algorithm主要由3個副程式組成:
  - **Merge1副程式**
    - 將兩個Run的記錄 (即: 兩筆已排序的記錄)，合併成一筆已排序的記錄 U (即: 合併成一個Run)
  - **MergePass副程式**
    - 在每一回合 (Pass) 中，會處理多次的 "合併兩個Run" 之工作
  - **MergeSort副程式** (可當作主程式)
    - 整個非遞迴的合併排序副程式需執行 $\lceil \log_2 n \rceil$ 回合 (Pass)

Merge1副程式

```
問題: 將兩筆已排序的記錄，合併成一筆已排序的記錄 U
輸入: 索引值 low, mid 與 high, 待合併之子陣列 S[ ], 合併好之子陣列 U[ ]
輸出: 合併好之子陣列 U[ ]

void merge1(S[ ], U[ ], int low, mid, high)
{
  i = low;
  k = low;
  j = mid+1;
  while (i ≤ mid && j ≤ high)
        {
          if (S[i] ≤ S[j])
            {
              U[k] = S[i];
              i++;
            }
          else
            {
              U[k] = S[j];
              j++;
            };
          k++;
        };
  if (i > mid)
     將 S[j...high] 的記錄移動到 U[k...high];
  else
     將 S[i...mid] 的記錄移動到 U[k...high];
}
```

- Run 1的長度爲 m，Run 2的長度爲 n，則合併兩個Run的最多比較次數爲 m+n-1 次

MergePass副程式

問題: 處理每一回合中，所有 Run 的兩兩合併動作
輸入: 待合併之子陣列 S[ ], 合併好之子陣列 U[ ], 記錄的總數 n, 某一回合中最大的 Run 之長度 L
輸出: 每一回合合併好之陣列 U[ ]

```
void MergePass(S[ ], U[ ], int n, L)
{
  for (int i = 1; i ≤ n − 2×L+ 1; i += 2×L)
       merge1(S[ ], U[ ], i, i + L - 1, i + 2×L− 1);
  if ((i + L − 1) < n)
       merge1(S[ ], U[ ], i, i+L−1, n);
  else for (int t = i; t <= n; t++)
            U[t] = S[t];
}
```

- 範例 1:

  執行 **Pass 2**: (n = 10, L = 2)

  - i = 1 時:

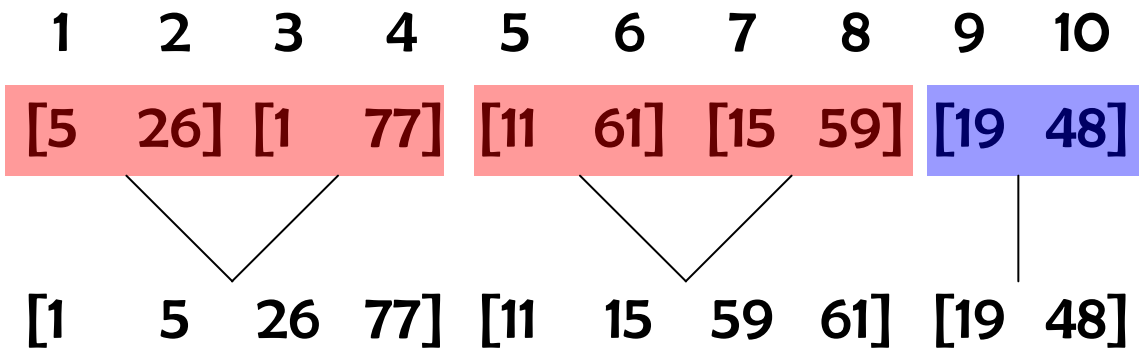
    做 merge1(S[ ], U[ ], 1, 2, 4)

  - i = 5 時:

    做 merge1(S[ ], U[ ], 5, 6, 8)

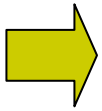
  - i = 9 時:

    做 for 迴圈



- MergePass副程式:

```
for (int i = 1; i ≤ n − 2×L+ 1; i += 2×L)
    merge1(S[ ], U[ ], i, i + L - 1, i + 2×L− 1);
if ((i + L − 1) < n)
    merge1(S[ ], U[ ], i, i+L−1, n);
else for (int t = i; t <= n; t++)
        U[t] = S[t];
```



```
for (int i = 1; i ≤ 7; i += 4)
        merge1(S[ ], U[ ], i, i + 1, i + 3);
if ((i + 1) < 10)
   merge1(S[ ], U[ ], i, i+1, 10);
else for (int t = i; t ≤ 10; t++)
        U[t] = S[t];
```

- 範例 2:

  執行 **Pass 4**: (n = 10, L = 8)

  - i = 1 時:

    做 if條件後的

    merge1(S[ ], U[ ], 1, 8, 10)

1	2	3	4	5	6	7	8	9	10
[1	5	11	15	26	59	61	77]	[19	48]

[1	5	11	15	19	26	48	59	61	77]

- MergePass副程式:

```
for (int i = 1; i ≤ n − 2×L+ 1; i += 2×L)
    merge1(S[ ], U[ ], i, i + L - 1, i + 2×L− 1);
if ((i + L − 1) < n)
    merge1(S[ ], U[ ], i, i+L−1, n);
else for (int t = i; t <= n; t++)
        U[t] = S[t];
```

```
for (int i = 1; i ≤ -5; i += 16)
        merge1(S[ ], U[ ], i, i + 7, i + 15);
if ((i + 7) < 10)
   merge1(S[ ], U[ ], i, i+7, 10);
else for (int t = i; t ≤ 10; t++)
        U[t] = S[t];
```

MergeSort副程式

問題: 執行 $\lceil \log_2 n \rceil$ 回合 (Pass)之合併排序

輸入: 待合併之陣列 S[ ], 記錄的總數 n

輸出: 合併好之陣列 S[ ]

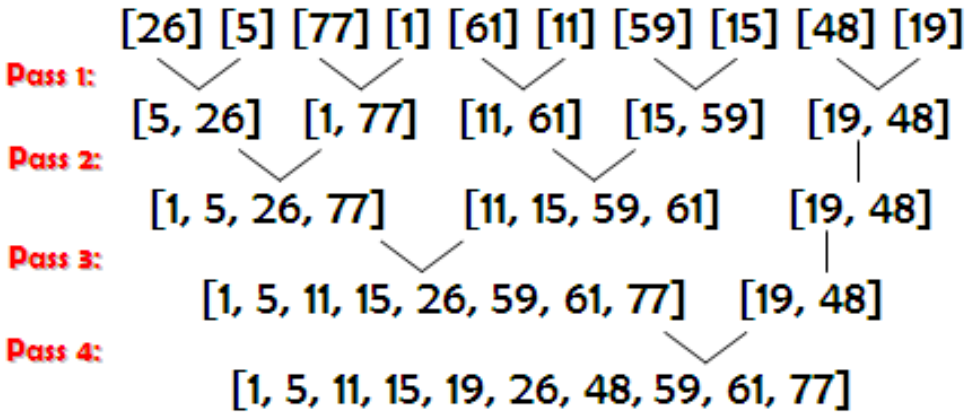
```
void MergeSort(S[ ], int n)
{
  keytype Temp[ ];
  for (int i = 1; i < n; i *= 2)
  {
      MergePass(S[ ], Temp[ ], n, i);
      i *= 2;
      MergePass(Temp[ ], S[ ], n, i);
  }
  delete Temp[ ];
}
```

i = 1

i = 2

i = 4

i = 8



- 執行 $\lceil \log_2 10 \rceil = \lceil 3.\times\times \rceil = 4$ 個回合(Pass)

補 5: Radix Sort (基數排序)

- 採取 "**Distribution & Merge**" 技巧來Sort。
  (分配、分派) (合併)
- 又稱Bin Sort或Bucket Sort
- 常用於卡片或信件的分類機
- 可分爲兩種:
  - LSD: Least Significant Digit First
  - MSD: Most Significant Digit First (✗)

LSD Radix Sort

- 做法:
  - 假設 **r** 爲基底 (Base, 或稱進制)，則準備 **r** 個 **Buckets**，其編號爲 0 ~ r-1
  - 假設 **d** 爲最大鍵值的位數個數，則須執行 **d** 回合才能完成 sort 工作
  - 從最低位數到最高位數執行d個回合，每一回合做:
    - 依位數值，將資料分配到對應的 Bucket 中 ... Distribution
    - 合併 r 個 Buckets (從 0 ~ r-1) ... Merge

範例

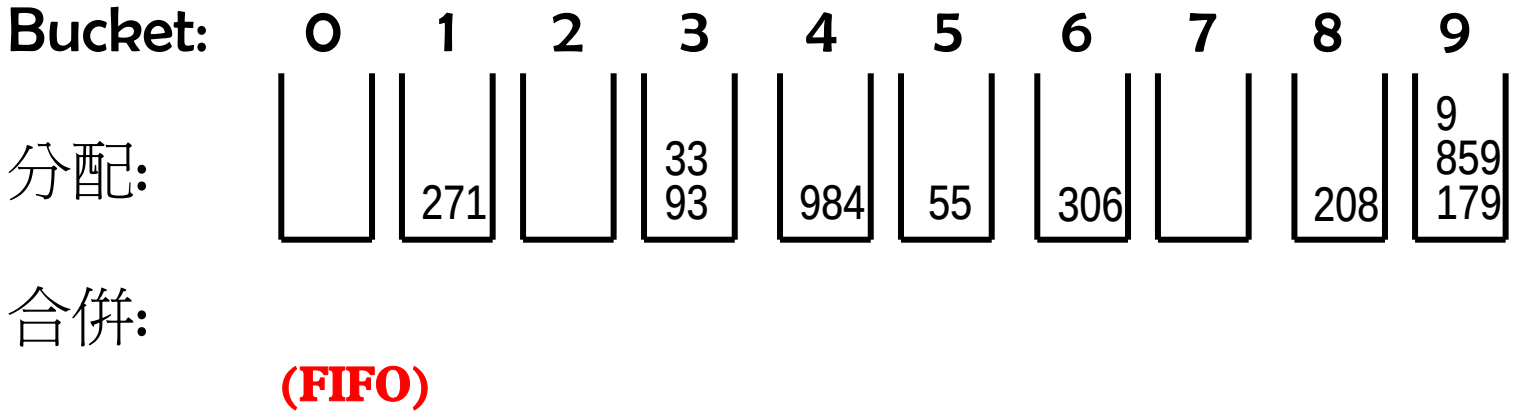
- 將下列數字以LSD Radix Sort加以排序 (Base = 10)

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

Sol:

- Base = 10, ∴準備 **10個桶子**，編號爲 0 ~ 9
- 最大的數值是984，有三個位數，∴d = 3，同時可知道需**執行 3 個回合**才會完成 Sort 工作
- 從**最低位數 (個位數)** 開始執行各回合:

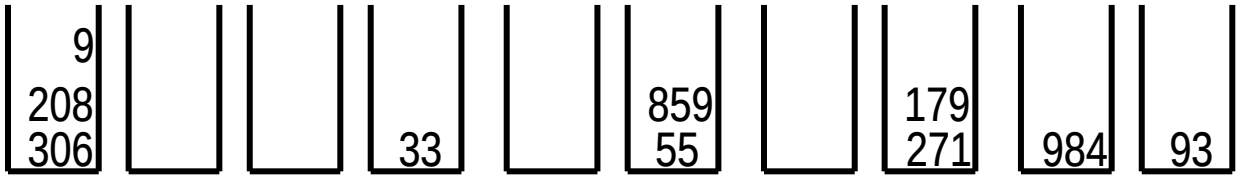
Pass 1: (針對個位數)



Pass 2: (針對十位數) (以Pass 1 的結果做爲Pass 2 的輸入)

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

分配:

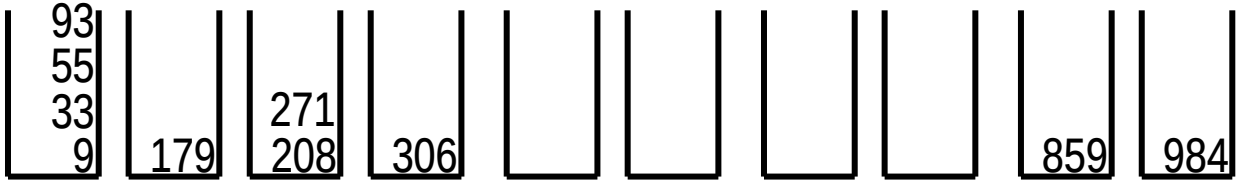


合併: 306, 208, 9, 33, 55, 859, 271, 179, 984, 93

Pass 3: (針對百位數) (以Pass 2 的結果做爲Pass 3 的輸入)

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

分配:



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

分析

- Time Complexity
- Space Complexity
- Stable / Unstable

Time-Complexity

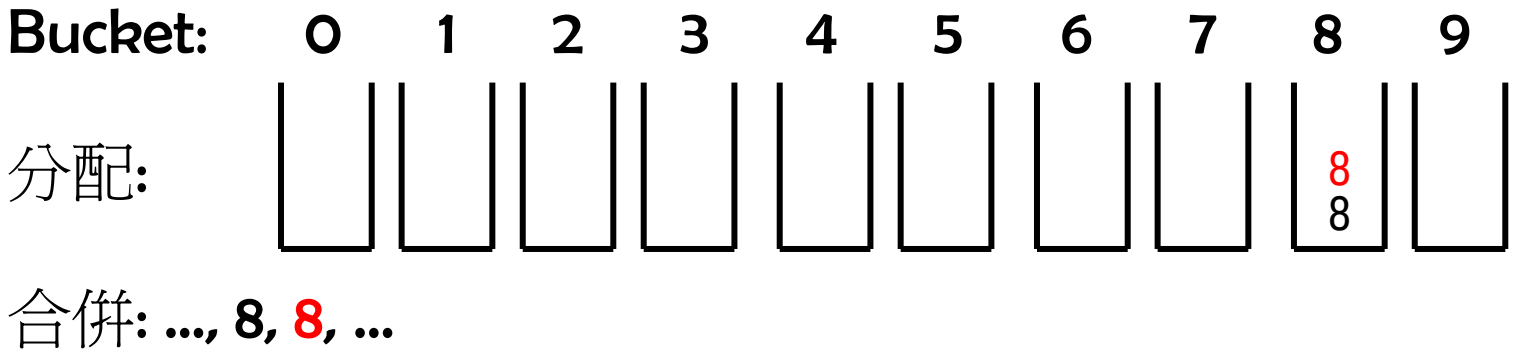
- 說明:
  - **d:** 回合數，**r:** 基底，**n:** 資料數目
  - ∵總共須執行 **d** 回合，而每一回合花費 **O(n+r)** 時間，其中:
    - 分配 **n** 個資料的時間: **O(n)**
    - 合併 **r** 個 **Buckets**的時間: **O(r)**
  - ∴總共花費 **O(d×(n+r))** 時間

Space-Complexity

- 說明:
  - ∵準備 r 個 Buckets，而每個 Buckets的Size最多爲 n
  - ∴ **O(n×r)**

Stable / Unstable

- Stable (穩定的)
- 說明: ..., 8, ..., 8, ...



∵相同鍵值的記錄在排序後，其相對位置沒有改變，亦即沒有不必要的Swap發生,

∴Stable

	Time Complexity				
	Best	Worst	Avg.	Space Complexity	Stable/Unstable
Insert Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Stable
Select Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	Unstable
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Stable
Quick Sort	O(n log n)	$O(n^2)$	O(n log n)	O(log n) ~ O(n)	Unstable
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Stable
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	Unstable
Radix Sort	**O(d×(n+r))**			**O(r×n)**	**Stable**