# Sorting 排序

林劭原老師

# Motivation動機

- 一個檔案(record)通常包含多個欄位(field)，**Keys**是其中可以用來區分 records的欄位，例如:身分證號碼、學號
- 假設Keys are integers，習慣上若搜尋成功(successful)，則回傳key value 是 k 的 a[i] 的 index i( 即 回 傳 a[i]=k 的 i) ； 若 搜 尋 失 敗 (unsuccessful)，則回傳0(也因為這原因，資料由a[1]開始儲存)
- 因此 Sorting 有兩個應用:
- 1. Searching 搜尋
- 2. List verification 表單驗證

# Searching

- 方法1:sequential search 循序搜尋

```
int SeqSearcj(int *a, const int n, const int k){
    int i;
    for(i = 1;i <= n && a[i] != k;i++);
    if(i > n) return 0;
    return i;
}
```

- In the worst case, 花 O(n) time

- 註: $f(n) = O(g(n))$ 的意思是:當 n 足夠大時，$g(n)$ is an asymptotic upper bound for $f(n)$, to within a constant factor.

# Searching

- 方法2:binary search 二元搜尋

```
int BinarySearch(int *a, const int n, const int k){
    int left = 1,right = n;
    while(left <= right){
        int middle = (left + right)/2;
        if(x < a[middle]) right = middle - 1;
        else if(x > a[middle]) left = middle + 1;
        else return middle;
    }
    return 0;
}
```

- In the worst case, 花O(lgn)time. 很顯然, better than sequential search.
- 但是要搜尋的資料必須sorted(已排序)，這是為何要有sorting的原因之一

# List verification

- List verification:Compare two lists to verify that they are identical 完全相同的.若不 identical, 則指出不同處.
- 例如：國稅局收到 two lists.

  from employer : employee 的薪水

  from employees : 自己的薪水
- 這兩個 list $L_1, L_2$ 應該 identical. Let $m = |L_1|$ and $n = |L_2|$
- 不使用 sorting 花 O(mn) time
- 使用 sorting 花 O(sorting+m+n) time,
- 其中 sorting 可以只花O(mlgm)+O(nlgn) time.

# Sorting

- 有研究指出，超過25%(甚至50%)的computing time是花在sorting上的，可見sorting之重要。
- 但是至今沒有任何一個sorting方法是在所有狀況下都最佳的。
- 因此我們將介紹好幾個sorting方法，並指出在什麼情況下，此方法將優於其他方法。

# Sorting problem

- 正式定義：Given a llist of records $R_1, R_2, \cdots, R_n$($R_i$ has key $K_i$), find a permutation $\sigma$ such that $R_{\sigma(1)}, R_{\sigma(2)}, \cdots, R_{\sigma(n)}$ has the property that $K_{\sigma(1)} \leq K_{\sigma(2)} \leq \cdots \leq K_{\sigma(n)}$.
- 當 keys 允許 identical，則 $\sigma$ 不唯一
- 例如：$K_1 = 10, K_2 = 5, K_3 = 10$ 可排序成$R_2, R_1, R_3$ 或 $R_2, R_3, R_1$
- 此時我們稱前面的 $\sigma$ 為 stable（keys相同時，原本在前面的，排序完依然在前），後面的 $\sigma$ is not stable.
- 有些排序方法很難做到 stable，quicksort 即一例。

# 這次要介紹的排序法

| Method | Worst | Average |
| --- | --- | --- |
| 氣泡排序法 Bubble Sort | $O(n^2)$ | $O(n^2)$ |
| 選擇排序法 Selection Sort | $O(n^2)$ | $O(n^2)$ |
| 插入排序法 Insertion Sort | $O(n^2)$ | $O(n^2)$ |
| 快速排序法 Quick Sort | $O(n^2)$ | $O(nlogn)$ |
| 合併排序法 Merge Sort | $O(nlogn)$ | $O(nlogn)$ |

# 氣泡排序法 Bubble Sort

- pseudo code：
- 
```
function bubble_sort (array, length) {
    for(i from 0 to length-1){
        for(j from 0 to length-1-i){
            if (array[j] > array[j+1]){
                swap(array[j], array[j+1])
            }
        }
    }
}
```
- 思考：這樣做為什麼可以進行排序？

# 實作氣泡排序法 Bubble Sort

- 看著上一頁的pseudo code，嘗試練習 Bubble Sort
- input:75 89 33 9 6 7 127
- output:6 7 9 33 75 89 127
- 如果還不會讓陣列進函式，可以直接寫在main裡面

# 選擇排序法 Selection Sort

- pseudo code：
- function Selection_sort (array, length) {
  for(i from 0 to length-1){
  j=i;
  for(k from i+1 to length-1)
  if (array[k] < array[j]) j=k;
  swap($a[i], a[j]$);
  }
  }
- 思考：這樣做為什麼可以進行排序？

# 實作選擇排序法 Selection Sort

- 看著上一頁的pseudo code，嘗試練習 Selection Sort
- input:75 89 33 9 6 7 127
- output:6 7 9 33 75 89 127
- 如果還不會讓陣列進函式，可以直接寫在main裡面

# 插入排序法 Insertion Sort

- Idea:設 $R_1, R_2, \cdots, R_i$ 為已經 sort 好的 records.

  Insert $R_{i+1}$ into $R_1, R_2, \cdots, R_i$ 中之適當位置

  Initially, $R_1$ 是已經 sort 好的，因此由 $R_2$ 開始做 sort 即可。

# 插入排序法 Insertion Sort

- void **Insert**(int temp,int *a,int i){
  a[0] = temp;//擋土牆
  while(temp < a[i]){
  a[i+1] = a[i];
  i--;
  }
  }
- void **InsertionSort**(int*a,const int n){
  for(int j = 2;j <= n;j++){
  int temp = a[j];
  **Insert**(temp,a,j-1);
  }
  }

- 特別小心：
- data in a[1],a[2],…,a[n]
  不是 a[0],a[1],…,a[n-1]

- 傳統：
- while(i>=1 && temp < a[i])
- 改成用擋土牆，不用每次while都判斷

# Insertion Sort 之結論

- Stable
- Best case:$O(n)$ time
- Worst case:$O(n^2)$ time
- Average case:$O(n^2)$ time
- 有研究指出，它是$n \leq 30$ 時之 fastest sorting method
- Variations(變型):
- 1. Binary Insertion Sort(insert時改用binary search而非sequential search)
- 2. Linked Insertion Sort(資料儲存在鏈結串列中而非陣列中)

# 實作 Insertion Sort

- 看著前兩頁的 code，練習 Insertion Sort
- input:75 89 33 9 6 7 127
- output:6 7 9 33 75 89 127

# 快速排序法 Quick Sort

- Idea:
  和 insertion sort 不同的是
  insertion sort 只是將 $R_i$ insert 到針對 $R_1, R_2, \cdots, R_{i-1}$ 來看正確位置;
  quick sort 則是將 $R_i$ 放置到針對 the whole list 來看正確位置(*)
  quick sort 是利用 pivot 以及 partition 來做到 (*)
- 習慣上，稱 $R_i$ 的 key $K_i$ 為 pivot(參考值)。常見的 pivot 有:
  (1) $K_l$ 最左的 key
  (2) $K_r$ 最右的 key
  (3) Median of $\{K_l, K_{\frac{l+r}{2}}, K_r\}$，又叫做 median of three.

# 快速排序法 Quick Sort

- 書上選用 $K_l$ 為 the pivot.
- 若它應該放在 p 位置上，則 quick sort will partition a[1..n] into:

p

| ≤pivot | pivot | ≥pivot |
|---|---|---|

- 此時左半邊和右半邊可以 independently 做排序.
- 如果找出 the index p?利用 i,j 兩 indices(指標).
- i 初值為 1 ，向右走，一開始就 i++，直到找到 ≥ pivot 者.
- j 初值為n+1，向左走，一開始就 i-- ，直到找到 ≤ pivot 者.

# 快速排序法 Quick Sort

- 例如:pivot是26.

| $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ | $R_6$ | $R_7$ | $R_8$ | $R_9$ | $R_{10}$ | |
|---|---|---|---|---|---|---|---|---|---|---|
| 26 | 5 | 37 | 1 | 61 | 11 | 59 | 15 | 48 | 19 | |
| i→ | | | | | | | | | ←j | |
| | | i↑ | | | | | | ↑j | | ∵ $i < j$,swap(37,19) |
| | | | | i↑ | | | ↑j | | | ∵ $i < j$,swap(61,15) |
| | | | | | ↑j | i↑ | | | | i 和 j 過頭了,swap(pivot,a[j]) |

- After partition, array a becomes:
- [11 5 19 1 15] 26 [59 61 48 37]

# 快速排序法 Quick Sort

- 完整的例子

| $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ | $R_6$ | $R_7$ | $R_8$ | $R_9$ | $R_{10}$ | left | right |
|---|---|---|---|---|---|---|---|---|---|---|---|
| [26 | 5 | 37 | 1 | 61 | 11 | 59 | 15 | 48 | 19] | 1 | 10 |
| [11 | 5 | 19 | 1 | 15] | 26 | [59 | 61 | 48 | 37] | 1 | 5 |
| [1 | 5] | 11 | [19 | 15] | 26 | [59 | 61 | 48 | 37] | 1 | 2 |
| 1 | 5 | 11 | [19 | 15] | 26 | [59 | 61 | 48 | 37] | 4 | 5 |
| 1 | 5 | 11 | 15 | 19 | 26 | [59 | 61 | 48 | 37] | 7 | 10 |
| 1 | 5 | 11 | 15 | 19 | 26 | [48 | 37] | 59 | [61] | 7 | 8 |
| 1 | 5 | 11 | 15 | 19 | 26 | 37 | 48 | 59 | [61] | 10 | 10 |
| 1 | 5 | 11 | 15 | 19 | 26 | 37 | 48 | 59 | 61 | | |

# pseudo code of Quick Sort

```
• void QuickSort(int *a,const int left,const int right){
        if(left <= right){
                int i = left,j = right +1 ,pivot = a[left]
                do{
                        do i++; while(a[i] < pivot)      //亦即do {i++;} while(a[i] < pivot)
                        do j--; while(a[j] > pivot)      //亦即do {j--;} while(a[j] > pivot)
                        if(i < j) swap(a[i],a[j])        //swap要另外寫
                }while(i < j);
                swap(a[left],a[j]);//the pivot針對the whole list來看的正確位置是at index j

                QuickSort(a,left,j-1);
                QuickSort(a,j+1,right);
        }
  }
```

# Quick Sort 之結論

- Not Stable
- Best case:O($nlgn$) time
- Worst case:O($n^2$) time (例如:list原本就in sorted order)
- Average case:O($nlgn$) time
- 有研究指出，它是考慮average computing time時，最快的sorting method
- 因為用到recursion，用到了system stack，因此比insertion sort多花記憶體

# 實作 Quick Sort

- 看著前兩頁的 pseudo code，練習 Quick Sort
- input:75 89 33 9 6 7 127
- output:6 7 9 33 75 89 127

# 合併排序法 Merge Sort

- 對，本來要講，但它的程式碼好複雜，所以我打算跟大家聊一下它的精神就好，有興趣知道細節的自己看課本。
- 假設你要排序一個數列，但一個人排序太累，所以你找了一隻會打字的猴子幫助你(例如隔壁同學)(會打莎士比亞全集的猴子)。你們將數列分成兩半，分別排序完成之後，該怎麼合併成一個排序好的數列呢?

# How fast can we sort?

- 雖然有best case, worst case, average case 三種，但是best case不太有代表性，average case 通常很難求出，因此，以worst case最常用，worst case反應出最差的情況所花時間的上界(upper bound)。

- 如果在做sorting之前，就對要sort的東西已有一些了解，例如，知道$0 \leq K_i \leq 999\ \forall i$,則可以用特殊的方法來sort，時間可以用的很少。例如:radix sort,counting sort.

# 基數排序 Radix Sort

- Each element has d digits, each digit has k possible values, and digit 1 is the least significant digit.
- 每個數都有d位數字，每位數字都有k可能值，digit 1是最不重要的digit

- Assume that the data are in array A[1..n]
- Radix sort:sort from LSD(least significant digit) to MSD(most significant digit)
- 由 LSD 做到 MSD ，每次依照"正在考慮的那個digit"排序

# ix Sort

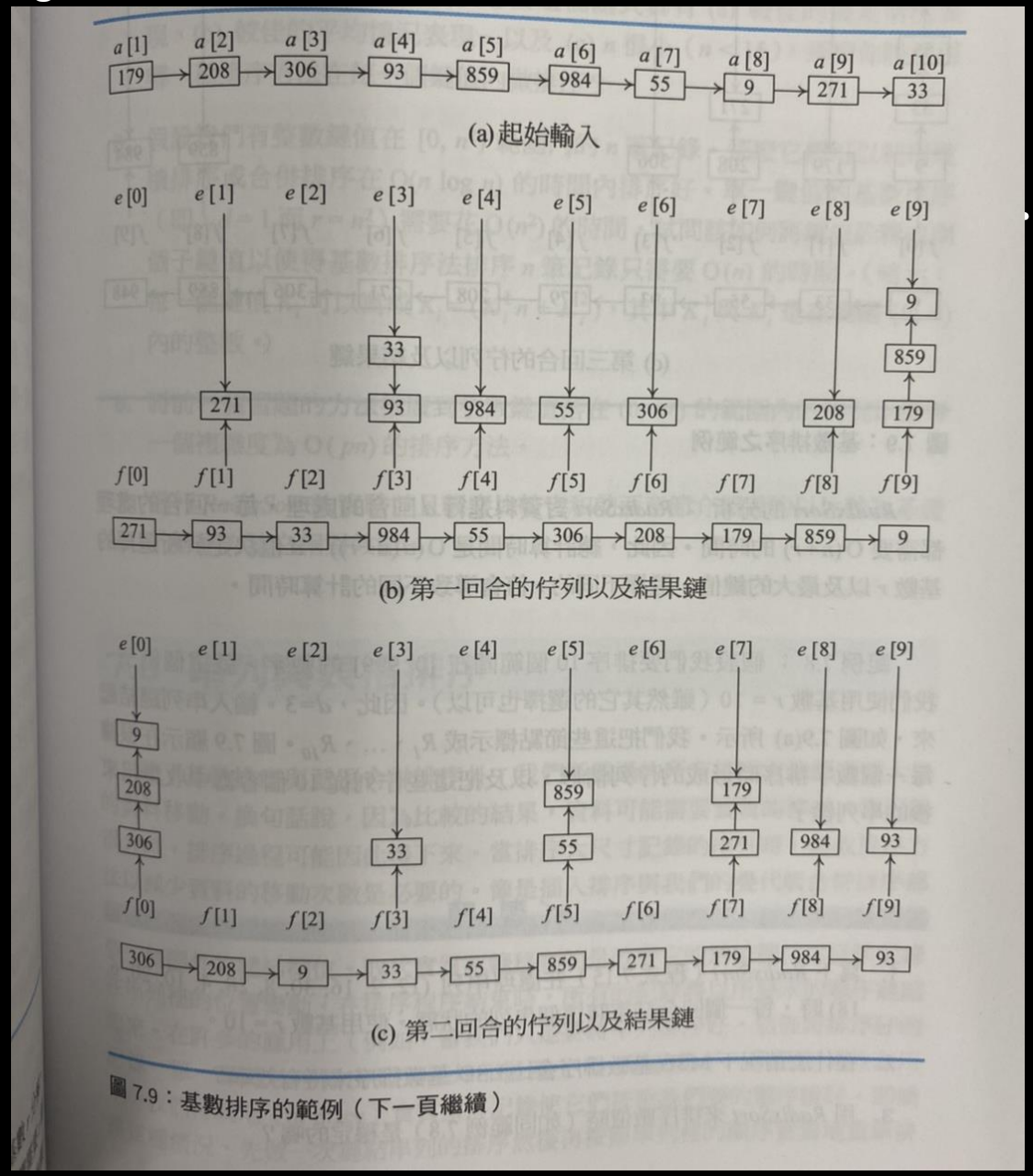



圖 7.9：基數排序的範例（下一頁繼續）

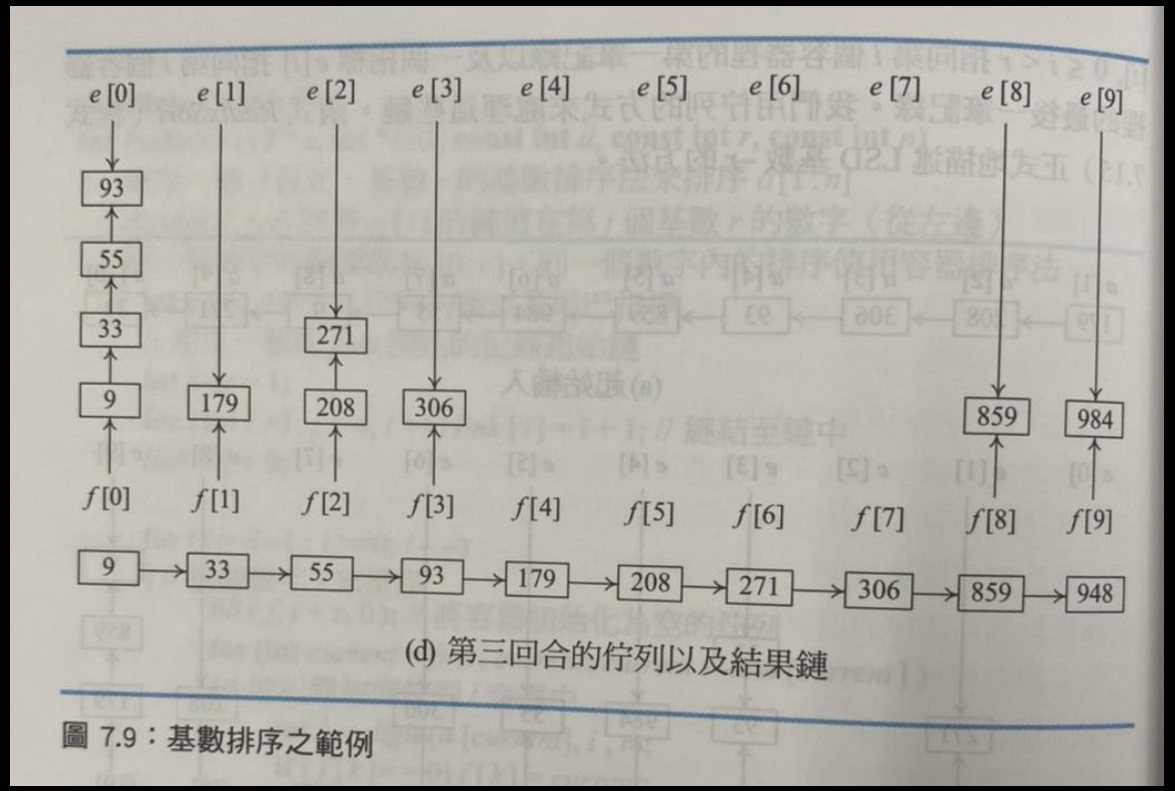
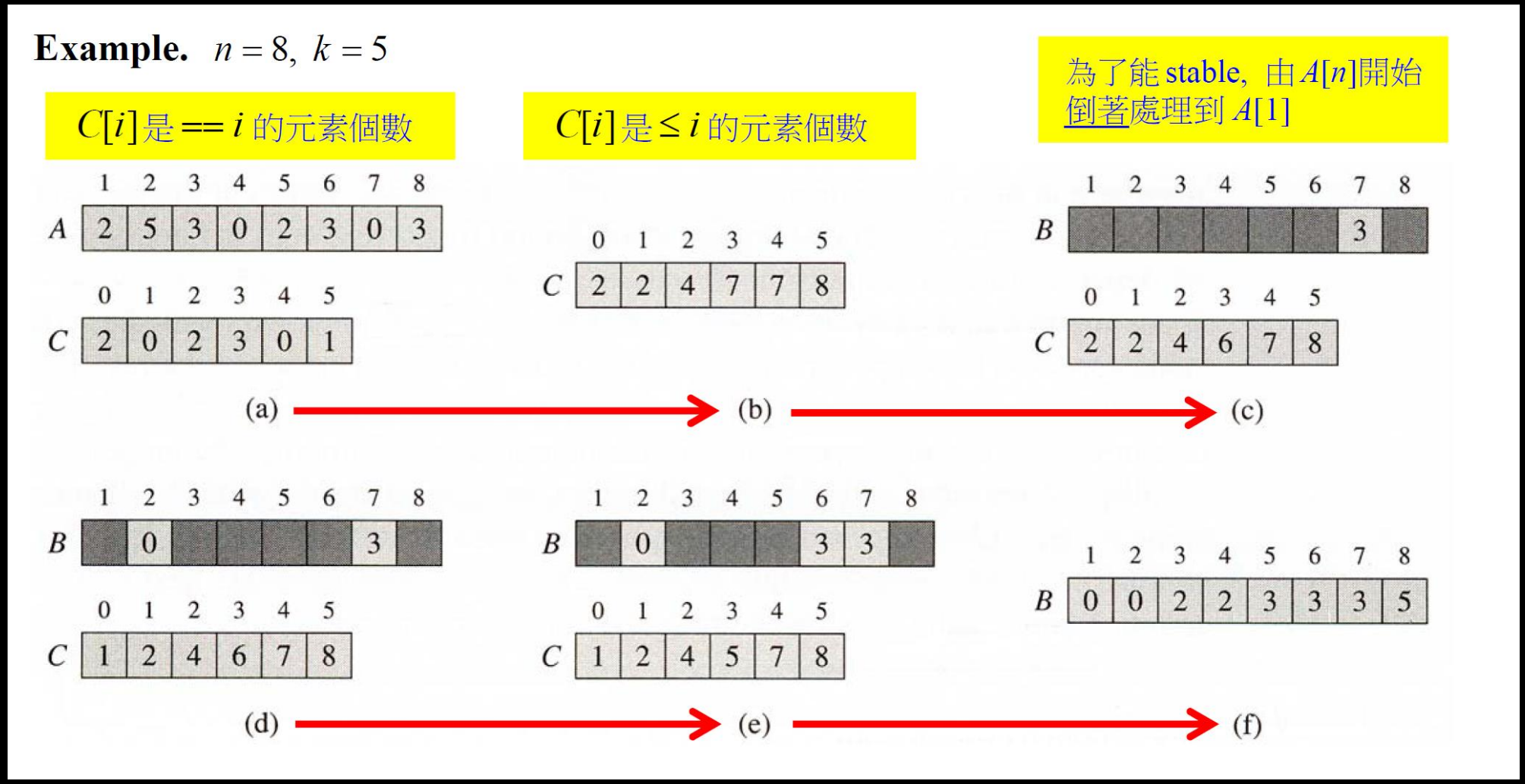



圖 7.9：基數排序之範例

# Counting sort

- Each of the n input numbers is an integer in the range 0 to k, for some integer k.
- Hence we can use O(n) time to count the number of times that each integer appears.可以先用O(n) time,數一數0到k中，每個數出現了幾次。

# Example of Counting sort



# How fast can we sort?

- 如果對於要sort的東西，並沒有一些了解，只知道每個東西都有key，只知道keys有a total order，則這時除了用comparisons of keys，別無它法。這樣的sorting method稱為是comparison-based sorting methods.

- Assume that input numbers are $a_1, a_2, \cdots, a_n$ and they are distinct.
- Any comparison sort can be described by a decision tree.

# How fast can we sort?

- **Theorem.** Any comparison-based sortin method requires $\Omega(nlgn)$ comparisons in the worst case.
- **Proof.**
  任何一個comparison-based sorting method都可以使用decision tree描述，這個decision tree的高度就是worst-case所需的comparisons個數。
  令$h$表示decision tree 的高度，$l$表示decision tree的樹葉個數
  (1)如果一個排序方法是正確的，則它必須能夠得到$n$個data的$n!$種排列中的任何一種，因此decision tree至少要有$n!$個樹葉，因此$l \geq n!$.
  (2)由於decision tree是二元樹，當它的高度是$h$，它最多只有$l \leq 2^h$個樹葉.

  由(1)及(2)，$n! \leq 2^h$，$n! = n(n-1)(n-2)\ldots(3)(2)(1) \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$

  因此$h \geq lg(n!) \geq \left(\frac{n}{2}\right) lg(\frac{n}{2}) = \Omega(nlgn)$，得證。

# 作業1:分數排序

- 第一次段考結束了，老師想將班上同學的數學成績由高排至低，方便統計觀察大家考的情況。每次輸入都是先輸入學生總人數，再輸入每位學生的座號及成績。輸出為依照分數排好的學生座號及其對應的分數。

| 測資一(20分) | 測資二(30分) | 測資三(50分) |
|---|---|---|
| 輸入：2 1 50 2 70 | 輸入：5 1 50 3 70 2 70 4 5 5 50 | 輸入：35人，分數隨機在0到100的整數 |
| 輸出：70 50 | 輸出(舉例)：70 70 50 50 5 | 輸出：排序後的分數及對應座號 |
| 2 1 | 2 3 1 5 4 | |

- 額外加分(20分)：學生的座號改為輸入英文名字

# 作業1:分數排序

- 注意事項：
- 1. 繳交期限為出作業後的隔週三 12:00 以前
- 2. 請將檔案上傳至 共用雲端硬碟\109多元選修_基礎資料結構與演算法\0. 個人作業\作業1_sorting
- 3. 檔名格式為 hw1_班級座號
- 4. 請上傳原始檔(.cpp檔)，不要只上傳執行檔
- 5. 同學之間可以互相討論，也可以來問我，但不要複製貼上，抄襲者以0分計