Queue

The Abstract Data Type Queue

Operation Contract for the ADT *Queue*

isEmpty():boolean {query} 是否為空
enqueue(in newItem:QueueItemType) 新增
throw QueueException
dequeue() throw QueueException 移除
getFront(out queueFront:QueueItemType) 擷取
{query} throw QueueException
dequeue(out queueFront:QueueItemType)
throw QueueException 擷取後移除

P. 6

ADT *queue* operations

- Create an empty queue 建構
- Destroy a queue 解構
- Determine whether a queue is empty 是否為空
- Add a new item to the queue 新增
- Remove the item that was added earliest 移除
- Retrieve the item that was added earliest 擷取

P. 5

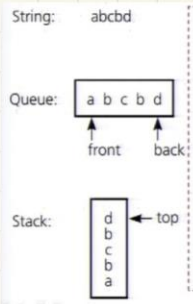
Recognizing Palindrome 回文

A palindrome (e.g.,, dad, noon, civic, level, …) 迴文
- A string of characters that reads *the same* from left to right as its does from right to left

To recognize a palindrome, you can use a queue in conjunction with a stack 堆疊
- A stack reverses the order of occurrences 顛倒次序
- A queue preserves the order of occurrences 佇列 保持次序

利用堆疊和次序相反的性質
比較 front of queue & top of stack

P. 11



isPal(in str: string): boolean
aQueue.createQueue()
aStack.createStack()
for (the next character ch in str)
{ *store ch into aQueue & aStack*
} // end for
while (aQueue is not empty)
{ *compare front & top*
} // end while 辨識迴文

P. 13

```
while (!aQueue.isEmpty()
   && charEqual)
{  aQueue.dequeue(front)
   aStack.pop(top)
   if (front != top)
       charEqual = FALSE
}  // end while
```

```
isPal(in str: string): boolean
aQueue.createQueue()
aStack.createStack()
for (the next character ch in str)
{  aQueue.enqueue(ch)
   aStack.push(ch)
}  // end for
charEqual = TRUE
```

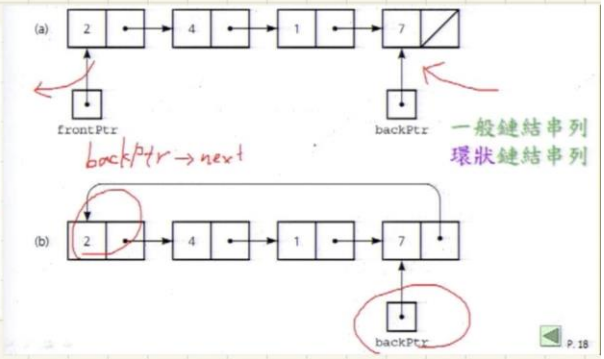
辨識迴文

P. 15

Implementation of the ADT Queue

- An *array*-based implementation (later)
- Possible implementations of a *pointer*-based queue
  - A *linear* linked list with two external references
    - A reference to the front — 前端
    - A reference to the back — 後端
  - A *circular* linked list with one external reference
    - Only a reference to the back — 環狀：只有後端

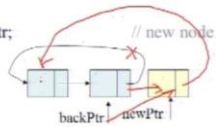
P. 17



P. 18

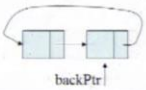
1. A Pointer-based Implementation

```
void Queue::enqueue(const QueueItemType& newItem)
{  QueueNode *newPtr = new QueueNode;                    環狀佇列
   newPtr->item = newItem;                                新增
   if (isEmpty())                          // 0 → 1 node
      newPtr->next = newPtr;               // point to itself
   else                                    // k → k+1 nodes, k > 0
   {  newPtr->next = backPtr->next;        // point to the front
      backPtr->next = newPtr;              // put behind the back
   }  // end else
   backPtr = newPtr;                       // new node at the back
}  // end enqueue
```



P. 28

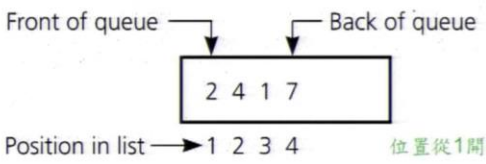
```
void Queue::dequeue() throw(QueueException)
{  if (isEmpty())                                        環狀佇列
      throw …;                                           移除
   else
   {  QueueNode *tempPtr = backPtr->next;      // the front
      if (backPtr == backPtr->next)
            backPtr = NULL;                   // one node → empty
      else  backPtr->next = tempPtr->next;    // the next front
      tempPtr->next = NULL;                   // defensive strategy
      delete tempPtr;                         // release space
   }  // end else
}  // end dequeue
```



P. 30

2. An Implementation That Uses the ADT List

The front of the queue is at position 1 of the list;
The back of the queue is at the end of the list



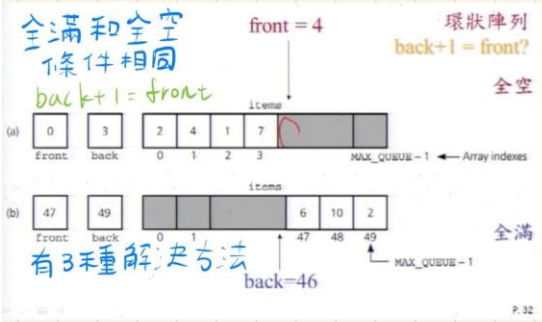
位置從1開始

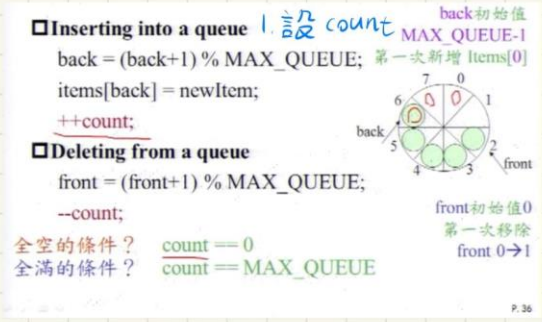
P. 42

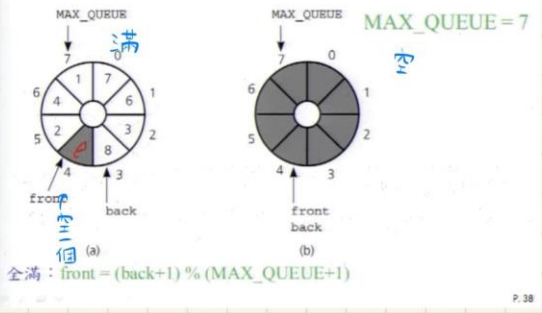
- `enqueue()` — 新增
  ```
  aList.insert(aList.getLength()+1,
               newItem)
  ```
- `dequeue()` — 移除
  ```
  aList.remove(1)
  ```
- `getFront(queueFront)` — 擷取
  ```
  aList.retrieve(1, queueFront)
  ```

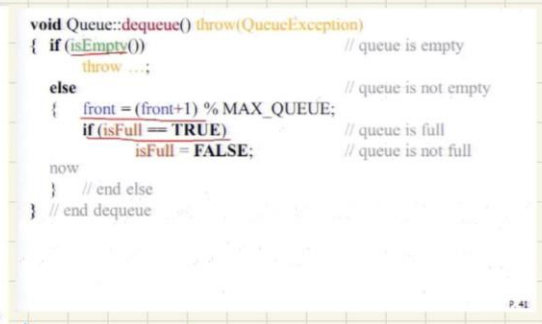
P. 43

3. An Array-based Implementation

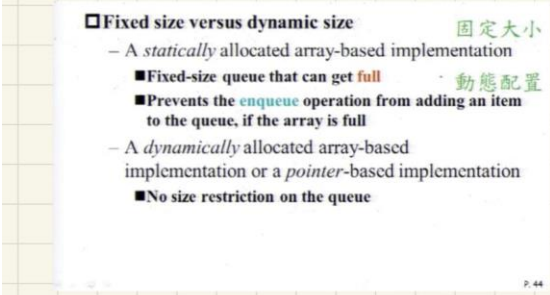


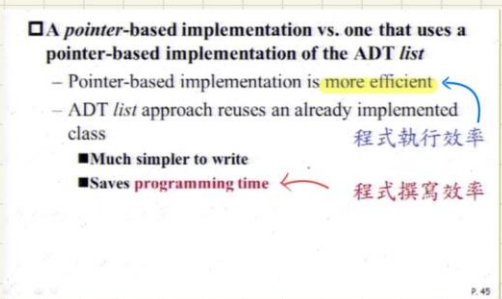






Comparing Implementations

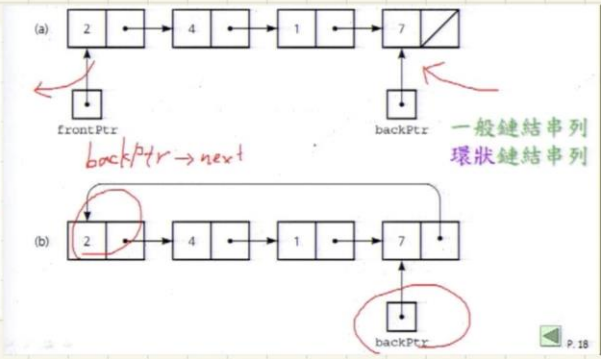




Implementation of the ADT Queue

- An *array*-based implementation (later)
- Possible implementations of a *pointer*-based queue
  - A *linear* linked list with two external references
    - A reference to the front — 前端
    - A reference to the back — 後端
  - A *circular* linked list with one external reference
    - Only a reference to the back — 環狀：只有後端

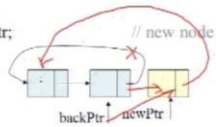
P. 17



P. 18

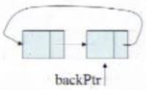
1. A Pointer-based Implementation

```
void Queue::enqueue(const QueueItemType& newItem)
{  QueueNode *newPtr = new QueueNode;                    環狀佇列
   newPtr->item = newItem;                                新增
   if (isEmpty())                          // 0 → 1 node
      newPtr->next = newPtr;               // point to itself
   else                                    // k → k+1 nodes, k > 0
   {  newPtr->next = backPtr->next;        // point to the front
      backPtr->next = newPtr;              // put behind the back
   }  // end else
   backPtr = newPtr;                       // new node at the back
}  // end enqueue
```



P. 28

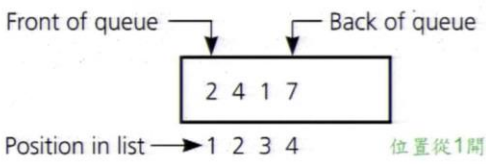
```
void Queue::dequeue() throw(QueueException)
{  if (isEmpty())                                        環狀佇列
      throw …;                                           移除
   else
   {  QueueNode *tempPtr = backPtr->next;      // the front
      if (backPtr == backPtr->next)
            backPtr = NULL;                   // one node → empty
      else  backPtr->next = tempPtr->next;    // the next front
      tempPtr->next = NULL;                   // defensive strategy
      delete tempPtr;                         // release space
   }  // end else
}  // end dequeue
```



P. 30

2. An Implementation That Uses the ADT List

The front of the queue is at position 1 of the list;
The back of the queue is at the end of the list



位置從1開始

P. 42

- `enqueue()` — 新增
  ```
  aList.insert(aList.getLength()+1,
               newItem)
  ```
- `dequeue()` — 移除
  ```
  aList.remove(1)
  ```
- `getFront(queueFront)` — 擷取
  ```
  aList.retrieve(1, queueFront)
  ```

P. 43

Simulation by Queue

☐ **Use the following event list to simulate a single bank queue and calculate the *average waiting time*.**

Events (input file)

Arrival	transaction	Departure	waiting
5	9	14	0
7	5	19 (14+5)	7 (14−7)
14	5	24	5
30	5	35	0
32	5	40	3
34	5	45	6
38	3		

單一佇列

P. 59

Multi-queue Simulation

☐ **Use the following event list to simulate two bank queues with bankQueue 1 first selection strategy and calculate the *average waiting time*.**

Events (input file)

Arrival	transaction	Departure	waiting
5	9	14	0
7	5	—	—
14	5	—	—
30	5	—	—
32	5	—	—
34	5	—	—
38	3	—	—

P. 61

☐ **Use the following event list to simulate two bank queues with bankQueue 1 first selection strategy and calculate the *average waiting time*.**

Events (input file)

AWT = 1/7

兩個佇列

Arrival	transaction
5	9
7	5
14	5
30	5
32	5
34	5
38	3

anEventList: D 40 | D 41

bankQueue 1: 34 5

bankQueue 2: 38 3

P. 62

Measuring the Efficency of Algorithms

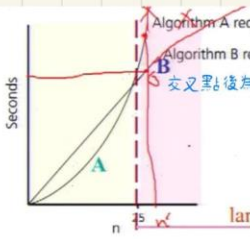
How many time units does the nested loop take?

- n = 10
- n = 100

```
for (a = 1; a <= n; a++)
  for (b = 1; b <= a; b++)
    for (c = 1; c <= 5; c++)
        cout << a << b << c << endl;   //t ms
```

Σ(t*5*a) for a = 1 to n → 5 * t * n * (n+1) / 2
→ t * (2.5n² + 2.5n) → n = 10: 275t, n = 100: 25250t

P. 10



交叉點後為問題變大的區塊

隨著n增加
A: 指數成長
B: 線性成長

large problem size

Algorithm A requires time proportional to n^2
Algorithm B requires time proportional to n

P. 12

Algorithm A is order *f*(*n*) – denoted O (*f*(*n*))

if constants *k* and n_0 exist such that *A* requires no more than *k* * *f*(*n*) time units to solve a problem of size $n \geq n_0$

大小位階

Examples

Practice 7-1 $2.5n^2 - 2.5*n$ is O(?), k=? n_0=?

$\forall n \geq n_0, (2.5n^2 - 2.5*n) \leq k * f(n)$

$\forall n \geq 10, (2.5n^2 - 2.5*n) \leq 1 * n^{10}$

$\forall n \geq n_0, (2.5n^2 - 2.5*n) \leq k * n^2$

$\forall n \geq 1, (2.5n^2 - 2.5*n) \leq 3 * n^2$

P. 15

Algorithm A is order *f*(*n*) – denoted O (*f*(*n*))

if constants *k* and n_0 exist such that *A* requires no more than *k* * *f*(*n*) time units to solve a problem of size $n \geq n_0$

Examples 鏈結串列 假設c, a, w皆為常數

Example 1. (n+1)*(c+a) + n*w is O(?), k=? n_0=?

$\forall n \geq n_0, (n+1)*(c+a) + n*w \leq k * f(n)$

$\forall n \geq 1, (n+1) \leq 2n$ →

$(n+1)*(c+a) + n*w \leq n * (2*(c+a) + w) \leq k * n$

P. 16

Worst-case analysis 最多

- A determination of the maximum amount of time that an algorithm requires to solve problems of size *n*

Average-case analysis 平均

- A determination of the average amount of time that an algorithm requires to solve problems of size *n*

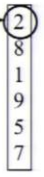
Best-case analysis 最少

- A determination of the minimum amount of time that an algorithm requires to solve problems of size *n*

P. 26

Sequential search 循序搜尋

- Strategy
  - Look at each item in the data collection in turn
  - Stop when the desired item is *found*, or the *end* of the data is reached
- Efficiency
  - Worst case: O(*n*)
  - Average case: O(*n*)
  - Best case: O(1)



P. 27

Efficiency of Sorting Algorithms

Binary search of a sorted array 二元搜尋

– Strategy
- Repeatedly divide the array in half
- Determine which half could contain the item, and discard the other half

– Efficiency
- Worst case: $O(\log_2 n)$

1	$\lceil 6/2 \rceil = 3$	$2^{k-1} < n < 2^k$
2	$\lceil 3/2 \rceil = 2$	$2^2 < 6 < 2^3$
5	$\lceil 2/2 \rceil = 1$	$2 < \log_2 6 < 3$
7	$n = 2^k$	
8	e.g., $16 = 2^4$	
9	$\log_2 16 = 4$	

P. 28

Binary search of a sorted array

– Efficiency
- Worst case: $O(\log_2 n)$
- For large arrays, the binary search has an enormous advantage over a sequential search
  - At most 20 comparisons to search one million items
  - $\log_2 10^6 = 19.9$

一百萬筆資料只需要做二十次比較！

P.

Consider a sequential searching of *n* data items

2. What is the order of the sequential search algorithm when the desired item is not in the data collection?
   – **Sorted vs. unsorted**
   – **Worse vs. average vs. best**

不同狀況下的位階？

	sorted	unsorted	*found*
Worse case	O(n)	O(n)	O(n)
Average case	O(n)	O(n)	O(n)
Best case	O(1)	O(n)	O(1)

P. 31

Categories of sorting algorithms

Categories of sorting algorithms 內部排序

– An internal sort 在記憶體內部做排序
- Requires that the collection of data fit entirely in the computer's main memory

– An external sort 在記憶體外做排序 外部排序
因內容太大！下學期會學到
- The collection of data will not fit in the computer's main memory all at once, but must reside in secondary storage

速度慢

P. 36

Stable Sort

Stable Sort vs. Unstable sort

1. bubble
2. insertion
3. merge
4. radix

1. quick
2. heap

相同值能夠/不能維持不變的排序

Ex.

8 28 14 5 8 26 2 6 29 5

↓↓

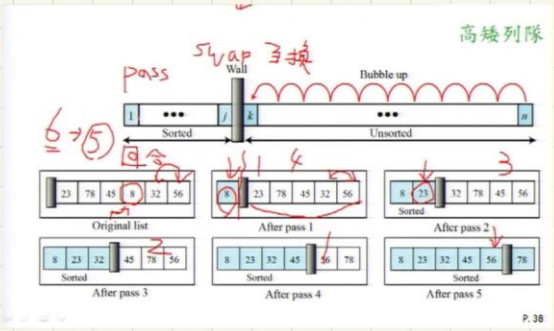
2 5 5 6 8 8 14 26 28 29

Sort Algorithms comparisons

Bubble sort

base $O(n^2)$ or $O(n)$

worst $O(n^2)$

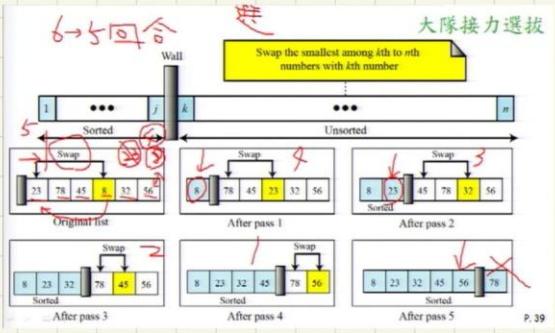


swap 次數多

selection sort

base $O(n^2)$ or $O(n)$

worst $O(n^2)$



swap 次數少

最差的 sort

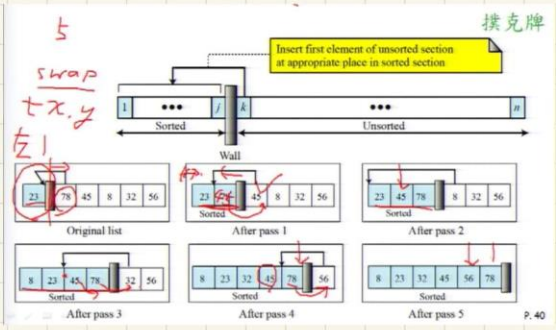
只適合應用在

單筆資料太大的情況

insertion sort

base $O(n)$

worst $O(n^2)$



無 swap

僅將元素

往右移後

插入

- **Sort the following data by using each algorithm:**

8_a 28 14 5_a 8_b 26 2 6 29 5_b

1. bubble sort
2. selection sort unstable!
3. insertion sort

Stable

BS: 2 5_a 5_b 6 8_a 8_b 14 26 28 29
SS: 2 5_a 5_b 6 8_b 8_a 14 26 28 29
IS: 2 5_a 5_b 6 8_a 8_b 14 26 28 29

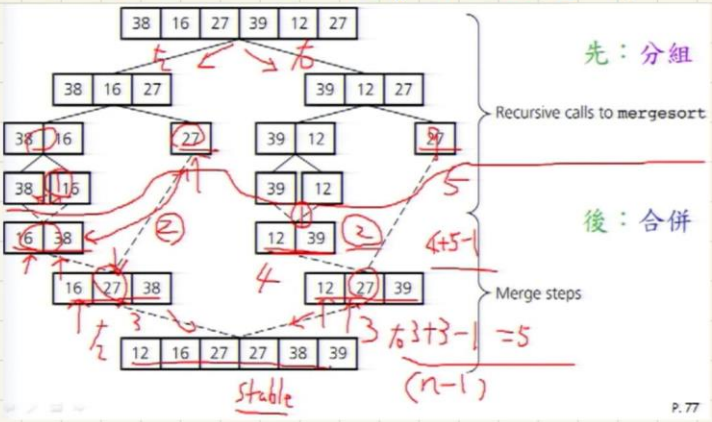
shellSort

```
void shellSort(int A[], int n)
{  for (int h = n/2; h > 0; h = h/2)
   for (int unsorted = h; unsorted < n; ++unsorted)
   {    int loc = unsorted;
        int nextItem = A[unsorted];
        for (; (loc >= h)&&(A[loc-h] > nextItem); loc=loc-h)
              A[loc] = A[loc-h];
        A[loc] = nextItem;
   }    // end for
}  // end shellSort
```

P. 71

Merge Sort

先分組，再排序，後合併




```
void mergeSort(DataType theArray[], int first, int last)
{ if (first < last)
  {   int mid = (first + last)/2;          // middle point
      mergeSort(theArray, first, mid);     // sort the left half
      mergeSort(theArray, mid+1, last);    // sort the right half
      merge(theArray, first, mid, last);   // merge the two halves
  }   // end if
} // end mergeSort
```

先：分組（遞迴呼叫）
後：合併

```
void merge(DataType theArray[], int first, int mid, int last)
{  DataType tempArray[MAX_SIZE];        // temporary array
   int first1 = first, last1 = mid;     // the left half [first…mid]
   int first2 = mid+1, last2 = last;    // the right half [mid+1...last]
   int index = first;                   // next available location
   for (; (first1 <= last1) && (first2 <= last2); ++index)
       if (theArray[first1] < theArray[first2])
       {     tempArray[index] = theArray[first1];
             ++first1;
       } else
       {     tempArray[index] = theArray[first2];
             ++first2;
       }     // end if-else
   ...
   for (; first1 <= last1; ++first1, ++index)   // finish the left half
       tempArray[index] = theArray[first1];
   for (; first2 <= last2; ++first2, ++index)   // finish the right half
       tempArray[index] = theArray[first2];

   for (index = first; index <= last; ++index) // copy the result back
       theArray[index] = tempArray[index];
} // end merge
```

10 33 60

21 43 57 72

兩組中最小的優先

Worst case: n-1 comparisons

寫回資料！

theArray <==> tempArray: 2n moves

∴ (n-1) + 2n = 3 * n – 1 major operations ➔ O(n)

P. 79 P. 80

Analysis

- Worst case: $O(n * \log_2 n)$
- Average case: $O(n * \log_2 n)$

Advantage

- Mergesort is an extremely fast algorithm

Disadvantage

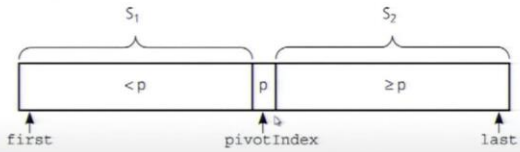
- Mergesort requires a second array as large as the original array

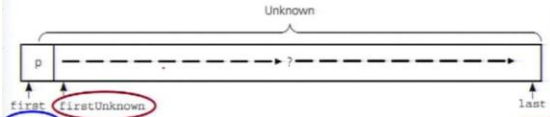
Quick Sort

- **Another divide-and-conquer algorithm**
- **Strategy** 找 pivot 分左右組 後排序
  - Choose a pivot 樞紐、軸
  - *Partition* the array about the pivot
    - **items < pivot** 先：分組（軸的位置）
    - **items >= pivot**
    - **Pivot is now in correct sorted position**
  - *Sort* the left section 後：遞迴呼叫
  - *Sort* the right section

P. 86

- **If pivot is placed at the correct sorted position…**



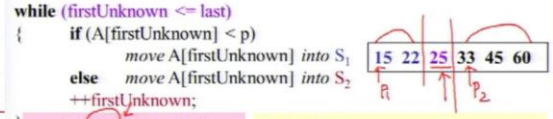


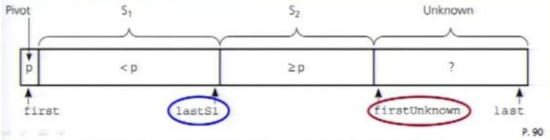

```
int lastS1 = first;                                  S1的最後一個位置
int firstUnknown = first+1;   // index of the first item in Unknown
while (firstUnknown <= last)
{       if (A[firstUnknown] < p)
                move A[firstUnknown] into S1
        else    move A[firstUnknown] into S2
        ++firstUnknown;
}
```

P. 89

```
while (firstUnknown <= last)
{       if (A[firstUnknown] < p)
                move A[firstUnknown] into S1
        else    move A[firstUnknown] into S2
        ++firstUnknown;
}
swap(A[first], A[lastS1]);
pivotIndex = lastS1;
```

```
++lastS1;
swap(A[lastS1], A[firstUnknown]);
```



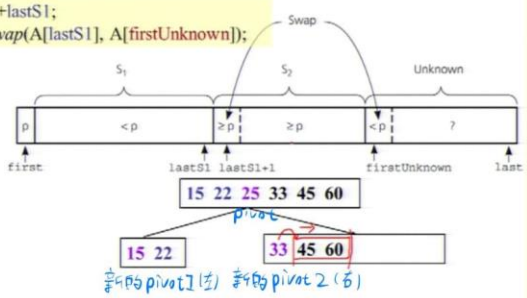


P. 90

```
void quickSort(DataType theArray[], int first, int last)
{  int pivotIndex;
   if (first < last)       // create the partition: S1, pivot, S2
   {    partition(theArray, first, last, pivotIndex);
        quickSort(theArray, first, pivotIndex-1);
        quickSort(theArray, pivotIndex+1, last);
   }    // end if                                  先：依軸分組
} // end quickSort                                 後：遞迴呼叫
```

P. 92

```
++lastS1;
swap(A[lastS1], A[firstUnknown]);
```



P. 91

average $O(n \log n)$ unstable!

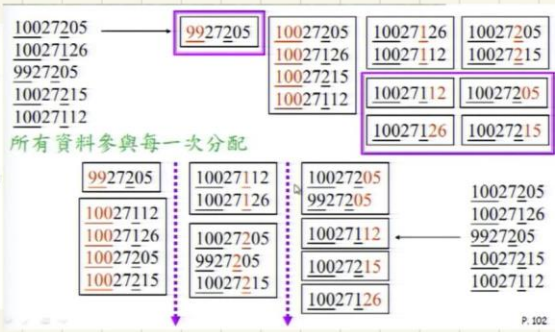
Worst $O(n^2)$

Radix Sort

取一個索引作為分組的radix

左邊補零

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150	Original integers
(1560, 2150) (1061) (0222) (0123, 0283) (2154, 0004)	Grouped by fourth digit
1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004	Combined
(0004) (0222, 0123) (2150, 2154) (1560, 1061) (0283)	Grouped by third digit
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Combined
(0004, 1061) (0123, 2150, 2154) (0222, 0283) (1560)	Grouped by second digit
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Combined
(0004, 0123, 0222, 0283) (1061, 1560) (2150, 2154)	Grouped by first digit
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Combined (sorted)



□LSD (*Least Significant Digit*) 依照最右側數字分組、串接

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150
- [0] (1560, 2150)
- [1] (1061)
- [2] (0222)
- [3] (0123, 0283)
- [4] (2154, 0004)

1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004
- [0] (0004)
- [2] (0222, 0123)
- [5] (2150, 2154)
- [6] (1560, 1061)
- [8] (0283)

P. 104

0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283
- [0] (0004, 1061)
- [1] (0123, 2150, 2154)
- [2] (0222, 0283)
- [5] (1560)

0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560
- [0] (0004, 0123, 0222, 0283)
- [1] (1061, 1560)
- [2] (2150, 2154)

0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154

P. 105

□MSD (*Most Significant Digit*) 改從最左側數字開始？

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150
- [0] (0123, 0222, 0004, 0283)
- [1] (1560, 1061)
- [2] (2154, 2150)

0123, 0222, 0004, 0283, 1560, 1061, 2154, 2150
- [0] (0004, 1061)
- [1] (0123, 2154, 2150)
- [2] (0222, 0283)
- [5] (1560)

P. 106

□MSD指排序時最重要的數(Most Significant Digit)
- 例：十進位的兩個數字X1X2X3和Y1Y2Y3（Xi和Yi都是0-9的數字，如356<412），X1和Y1是決定大小最重要的數字，故最左邊的數稱為MSD
- 字串排序以最左邊為MSD（如"john"<"mary"）

□以356和412為例，觀察每個回合的分配和串接：

從MSD開始做：最後一次會按照LSD分配後並依序串接

串接的次序是錯的！

P. 107

Radix Implementation

```
void radixSort(int A[], int first, int last)                    先：分組
{ // the maximum in A is a d-digit integer                      後：串接
  for (j = d down to 1)   從最右側開始！
  {  Initialize 10 groups with counters reset;
     for (i = first to last)
     {     k = j^th  digit of A[i];   第j位數字決定分組
           increase the counter of group k by 1;
           Append A[i] to group k;
     }     // end for                  依序串接分組
     replace A with the sequence of group 1,... group 10
  }  // end for
} // end radixSort
```

P. 108

```
void radixSort(int A[], int first, int last)                    先：分組
{ int temp[MAX_SIZE], maxData;                                  後：串接
  int bucket[10], i;
  for (maxData=A[first], i=first+1; i <= last; i++)
      if (maxData < A[i])
            maxData=A[i];          // d-digit integer
  for (int base=1; (maxData / base) > 0; base*=10)
  {   for (i=first; i <= last; i++)        // counting
            bucket[ (A[i] / base) % 10 + 1]++;
      …
  }   // end for
} // end radixSort
```

P. 109

```
void radixSort(int A[], int first, int last)
{ …
  for (int base=1; (maxData / base) > 0; base*=10)
  {   … bucket[0] = 0;
      for (i=1; i < 10; i++)       // the start of each group
            bucket[i] += bucket[i-1];
      for (i=first; i <= last; i++)          依序串接分組
            temp[ bucket[ (A[i] / base) % 10 ]++ ] = A[i];
  }   // end for
} // end radixSort
```

[0] (0004, 0123, 0222, 0283)
[1] (1061, 1560)
[2] (2150, 2154)
[3] 0

P. 110

Implementation II

```
void radixSort(int A[], int first, int last)
{ int temp[MAX_D][MAX_SIZE], maxData;
  int counter[10]={0}, i, j;
  for (maxData=A[first], i=first+1; i <= last; i++)
      if (maxData < A[i])          maxData=A[i];
  for (int base=1; (maxData / base) > 0; base*=10)
  {   for (i=first; i <= last; i++)        // counting
      {     int LSD = (A[i] / base) % 10;
            temp[LSD][ counter[LSD] ] = A[i];
            counter[LSD]++;
      }     // end for
      …                                   LSD即代表分組
```

[0]2 (1560, 2150)
[1]1 (1061)
[2]1 (0222)
[3]2 (0123, 0283)
[4]2 (2154, 0004)

P. 111

```
  for (base=1; (maxData / base) > 0; base*=10)
  {   …
      int k=0;                                  依序串接分組
      for (i=0; i < 10; i++)       // concatenate the groups
            if (counter[i] > 0)
            {     for (int j=0; j < counter[i]; j++, k++)
                        A[k]= temp[i][j];
                  counter[i] = 0;
            }     // end if
  }   // end for
```

[0]2 (1560, 2150)
[1]1 (1061)
[2]1 (0222)
[3]2 (0123, 0283)
[4]2 (2154, 0004)

O(2*n*d) → **O(n)**

P. 112