

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) 擷取後移除
 throw QueueException

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

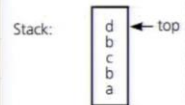
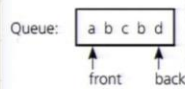
- A string of characters that reads *the same* from left to right as it does from right to left

To recognize a palindrome, you can use a queue in conjunction with a stack

- 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

String: abcbd



```

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
  
```

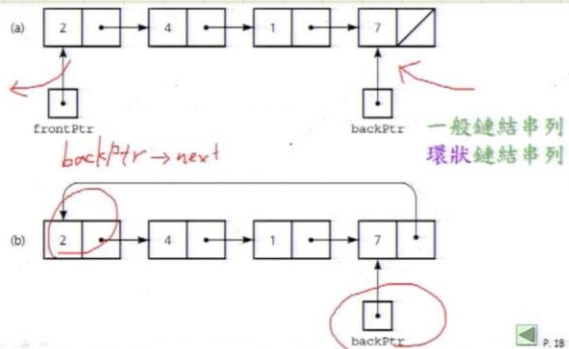
P. 15

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
 - A reference to the back
 - A circular linked list with one external reference
 - Only a reference to the back

前端
後端

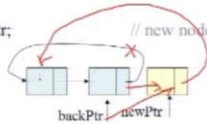
環狀：只有後端



1. A Pointer-based Implementation

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

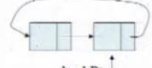
環狀佇列
新增



P. 28

```
void Queue::dequeue() throw (QueueException)
{
    if (isEmpty())
        throw ...;
    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
```

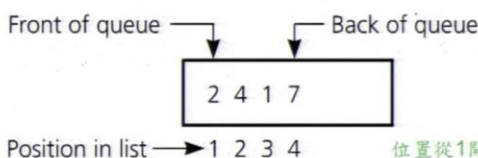
環狀佇列
移除



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 ()
    aList.insert (aList.getLength ()+1,
        newItem)
□ dequeue ()
    aList.remove (1)
□ getFront (queueFront)
    aList.retrieve (1, queueFront)
```

新增

移除

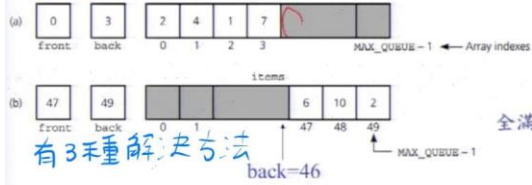
擷取

P. 43

3. An Array-based Implementation

全滿和全空
條件相同

$back + 1 = front$



環狀陣列

$back + 1 = front?$

全空

全滿

有3種解法方法

$back = 46$

P. 32

Inserting into a queue 1. 設 count

$back = (back + 1) \% MAX_QUEUE;$ 第一次新增 items[0]

items[back] = newItem;

$++count;$

Deleting from a queue

$front = (front + 1) \% MAX_QUEUE;$

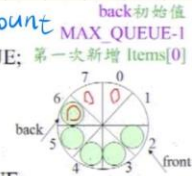
$--count;$

全空的條件?

$count == 0$

全滿的條件?

$count == MAX_QUEUE$



front 初始值 0
第一次移除
front 0 → 1

P. 36

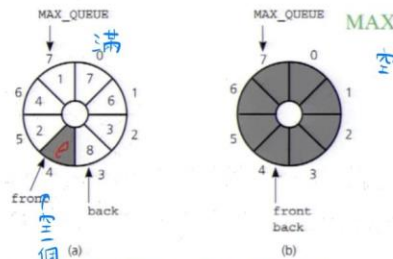
2. 多宣告一個空間

Variations of the array-based implementation

2. Declare $MAX_QUEUE + 1$ locations for the array items, but use only MAX_QUEUE of them for queue items

3. Use a flag **isFull** to distinguish between the full and empty conditions

設定旗標：是否全滿



全滿: $front = (back + 1) \% (MAX_QUEUE + 1)$

P. 38

Queue::Queue(): back(MAX_QUEUE-1), front(0); isFull(FALSE);

bool Queue::isEmpty() const
{ return (!isFull) && (front == (back + 1) \% MAX_QUEUE); }

void Queue::enqueue(const QueueItem& newItem) throw(QueueException)
{ if (isFull == TRUE) throw; else { back = (back + 1) \% MAX_QUEUE; items[back] = newItem; if (front == (back + 1) \% MAX_QUEUE) isFull = TRUE; } }

3. isFull Flag

P. 40

void Queue::dequeue() throw(QueueException)

{ if (isEmpty()) throw; else { front = (front + 1) \% MAX_QUEUE; if (isFull == TRUE) isFull = FALSE; } }

P. 41

Comparing Implementations

Fixed size versus dynamic size

固定大小

- A statically allocated array-based implementation

■ Fixed-size queue that can get full 動態配置

■ Prevents the enqueue operation from adding an item to the queue, if the array is full

- A dynamically allocated array-based implementation or a pointer-based implementation

■ No size restriction on the queue

P. 44

A pointer-based implementation vs. one that uses a pointer-based implementation of the ADT list

- Pointer-based implementation is more efficient

- ADT list approach reuses an already implemented class

程式執行效率

■ Much simpler to write

■ Saves programming time

程式撰寫效率

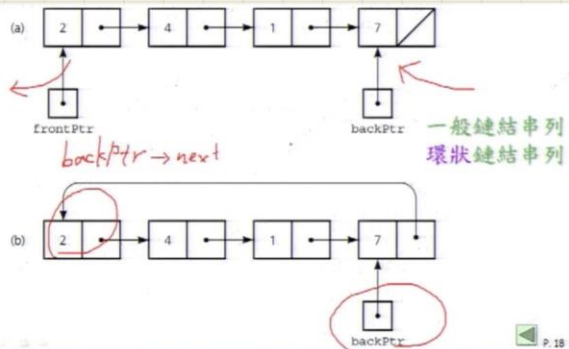
P. 45

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
 - A reference to the back
 - A circular linked list with one external reference
 - Only a reference to the back

前端
後端

環狀：只有後端



1. A Pointer-based Implementation

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

環狀佇列
新增

P. 28

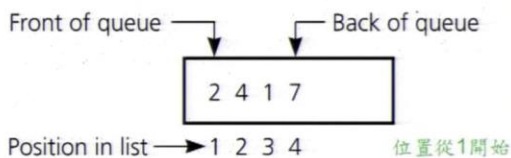
```
void Queue::dequeue() throw (QueueException)
{
    if (isEmpty())
        throw ...;
    else
    {
        QueueNode *tempPtr = backPtr->next; // the front
        if (backPtr == backPtr->next)
            backPtr = NULL; // one node → empty
        else
            backPtr->next = tempPtr->next;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

環狀佇列
移除

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 ()
    aList.insert (aList.getLength ()+1,
                  newItem)
□ dequeue ()
    aList.remove (1)
□ getFront (queueFront)
    aList.retrieve (1, queueFront)
```

新增
移除
擷取

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		

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*.

AWT = 1/7

Events (input file)

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

兩個佇列

P. 62

Measuring the Efficiency of Algorithms

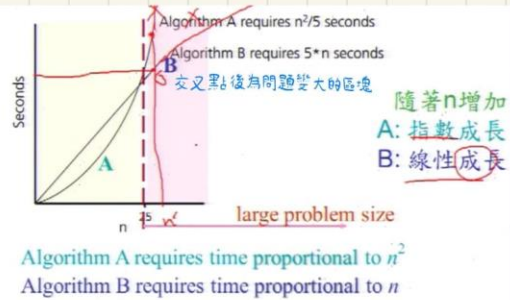
How many time units does the nested loop take?

```

- 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;
  
```

Handwritten notes: $n=1, b=1, 5 \times 1$; $n=2, b=1, 2, 5 \times 2$; $n=3, b=1, 2, 3, 5 \times 3$. The innermost loop is circled and labeled n . Below the code, the calculation is shown: $\Sigma(t \cdot 5 \cdot a) \text{ for } a=1 \text{ to } n \rightarrow 5 \cdot t \cdot \frac{n(n+1)}{2} \rightarrow t \cdot (2.5n^2 + 2.5n) \rightarrow n=10: 275t, n=100: 25250t$.

P. 10



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 $k \cdot f(n)$ time units to solve a problem of size $n \geq n_0$

Examples

Practice 7-1 $2.5n^2 - 2.5 \cdot n$ is $O(?)$, $k=?$, $n_0=?$
 $\forall n \geq n_0, (2.5n^2 - 2.5 \cdot n) \leq k \cdot f(n)$
 $\forall n \geq 10, (2.5n^2 - 2.5 \cdot n) \leq 1 \cdot n^{10} \rightarrow O(n^{10})$
 $\forall n \geq n_0, (2.5n^2 - 2.5 \cdot n) \leq k \cdot n^2 \rightarrow O(n^2)$
 $\forall n \geq 1, (2.5n^2 - 2.5 \cdot n) \leq 3 \cdot n^2 \rightarrow O(n^2)$

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 $k \cdot f(n)$ time units to solve a problem of size $n \geq n_0$

Examples

Example 1. $(n+1) \cdot (c+a) + n \cdot w$ is $O(?)$, $k=?$, $n_0=?$
 $\forall n \geq 1, (n+1) \cdot (c+a) + n \cdot w \leq k \cdot f(n)$
 $\forall n \geq 1, (n+1) \leq 2n \rightarrow k=2$
 $(n+1) \cdot (c+a) + n \cdot w \leq n \cdot (2 \cdot (c+a) + w) \leq k \cdot n$

P. 16

Worst-case analysis

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

Average-case analysis

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

Best-case analysis

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

P. 26

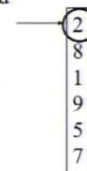
Sequential search

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

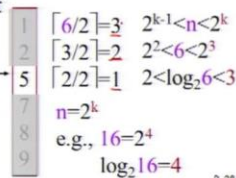
二元搜尋

– Strategy

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

– Efficiency

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



□ 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！

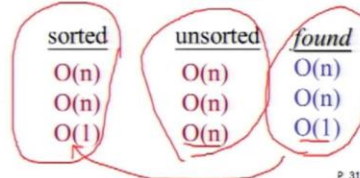
□ Consider a sequential searching of n data items

- 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？

Worst case
 Average case
 Best case



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

- The collection of data will **not** fit in the computer's main memory all at once, but must reside in **secondary storage**

因內容太大！下學期會學到
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

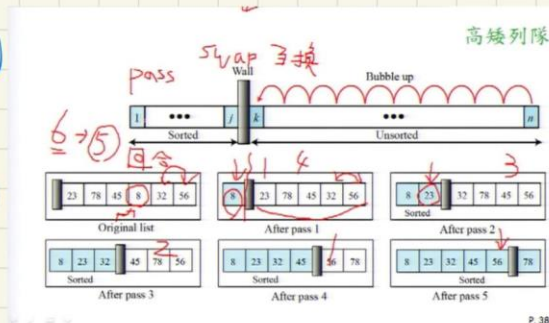
Sort Algorithms comparisons

Bubble sort

best $O(n^2)$

$O(n^2)$

worst $O(n^2)$



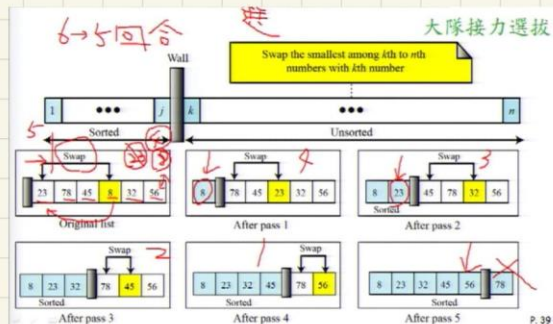
swap 次數多

selection sort

best $O(n^2)$

or $O(n^2)$

worst $O(n^2)$



swap 次數少

最差的 sort

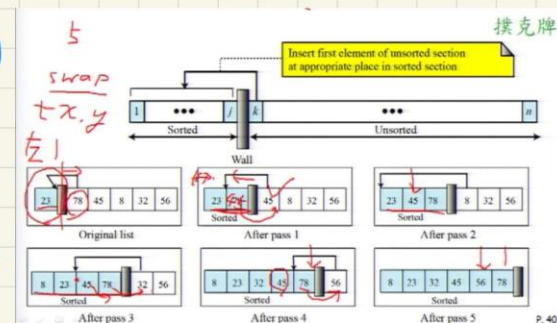
只適合應用在

單筆資料太大的情況

insertion sort

best $O(n^2)$

worst $O(n^2)$



無 swap

僅將元素

往右移後

插入

□ Sort the following data by using each algorithm:

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

1. bubble sort

2. selection sort unstable!

Stable

3. insertion sort

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

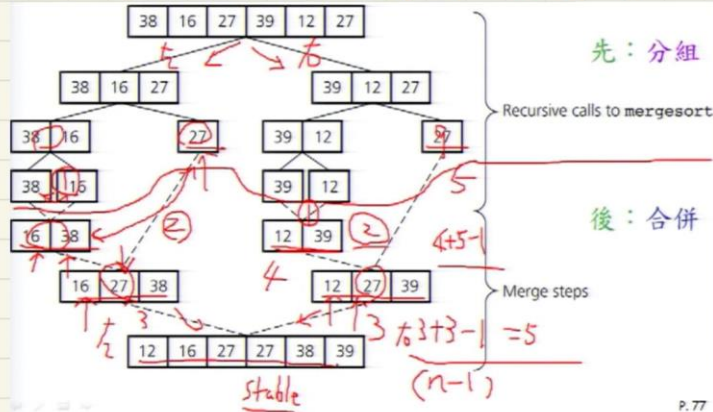
IS: 2 5_a 5_b 6 8_a 8_b 14 26 28 29

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
```

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
}
```

Worst case: $n-1$ comparisons

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

```
...
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
```

寫回資料!

theArray <=> tempArray: $2n$ moves

$\therefore (n-1) + 2n = 3 * n - 1$ major operations $\rightarrow O(n)$

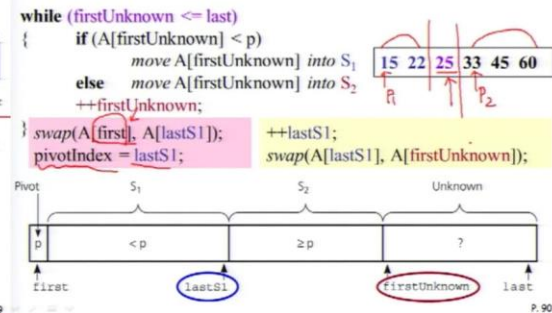
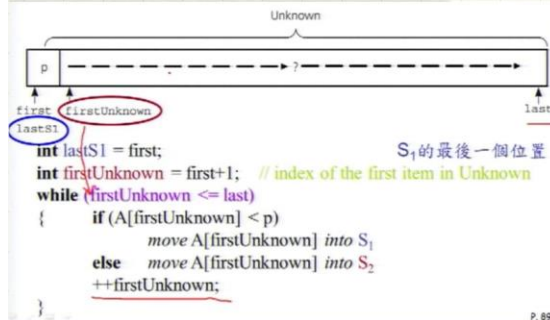
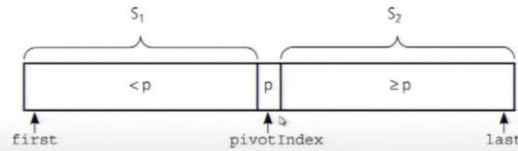
Quick Sort

Another divide-and-conquer algorithm

Strategy

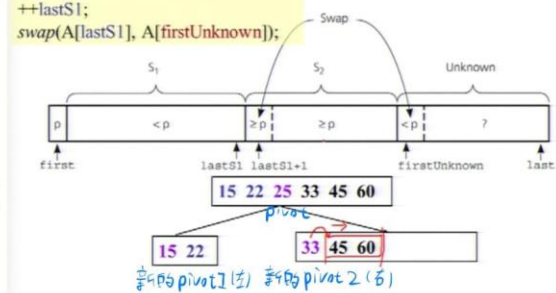
- Choose a **pivot** 找 pivot 分左右組 樞紐、軸
- Partition the array about the pivot
 - items < pivot 先：分組（軸的位置）
 - items ≥ pivot
 - Pivot is now in **correct** sorted position
- Sort the **left** section 後：遞迴呼叫
- Sort the **right** section

If pivot is placed at the **correct** sorted position...



```
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
} // end quickSort
```

先：依軸分組
後：遞迴呼叫



average $O(n \log n)$

unstable!

Worst $O(n^2)$

Radix Sort

取一個索引作為分組的radix

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

左邊補零

Original integers

Combined

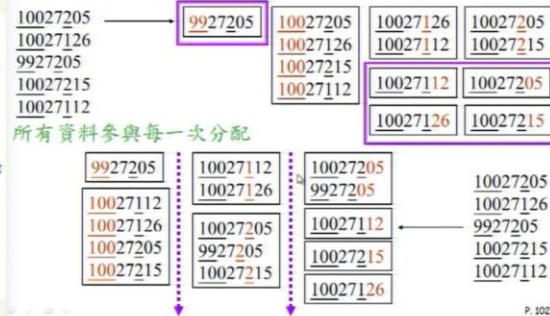
Combined

Grouped by second digit

Combined

Grouped by first digit

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)

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

□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)

□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！

Radix Implementation

```
void radixSort(int A[], int first, int last)
{ // the maximum in A is a d-digit integer
  for (j = d down to 1) 從最右側開始!
  {
    Initialize 10 groups with counters reset;
    for (i = first to last)
    {
      k = jth 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
    replace A with the sequence of group 1, ... group 10
  } // end for
} // end radixSort
```

先：分組
後：串接

P. 108

```
void radixSort(int A[], int first, int last)
{ int temp[MAX_SIZE], maxData;
  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
```

先：分組
後：串接

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) - 0
(1) (1061, 1560) - 4
(2) (2150, 2154) - 6
(3) 0 - 2

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
  } // end radixSort
```

LSD即代表分組

P. 111

```
for (base=1; (maxData / base) > 0; base*=10)
{
  int k=0;
  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]--;
      } // end if
    } // end for
  } // end for
```

$O(2 \cdot n \cdot d) \rightarrow O(n)$

P. 112