

递归

讓問題變得更簡單

提取字元 (substr)

e.g. `s.substr(size-1, 1);`

E.g.

① 輸出字串 \rightarrow 簡化 \rightarrow 提取字元 \rightarrow 迴圈停止條件

```
void writeBackward (string s, int size) {  
    if (size > 0) {  
        cout << s.substr(size-1, 1);  
        writeBackward (s, size-1); // 遞迴呼叫  
    } // if  
}
```

- ☆ 1. 遞迴定義
- 2. 問題簡化
- 3. 停止條件
- 4. 傳遞終止

求最大公因數 GCD

① `int gcd1 (int x, int y) {`

`if (y == 0) return x; // 若 $y=0$, x 是答案`

`(y > x) {`
`else if (y > x) return gcd1 (x, y % x); // $y \% x$ 後 y 變小`
`else return gcd1 (y, x % y); // 一直取余數`

\rightarrow 直到 $y=0$

② `int gcd2 (int x, int y) {`

`if (!x % y) return y; // 若 x 能被 y 整除, 答案是 y`
`else return gcd2 (y, x % y);`

}

如 第①算法

① 利用取余數把數字變小 \rightarrow 問題簡化

② 最後會 $=0$, 確能遞迴停止 (base case)

正確性

☆ 算法②會以步驟

\rightarrow 更有效率

gcd1 在整除後最後
 gcd2 整除後結束

如 第②算法

① 看有沒有余數 \rightarrow ④ 答案是 $y \rightarrow$ ⑤ 步驟 2

② 叫遞迴, `gcd2 (y, x % y)`; 左邊外層比較大

特殊情況

因沒有區分 x, y 大小
若 $x < y$
次數相同

二分法

① 看是不是答案 \rightarrow ② 分一半 $\text{mid} = (\text{first} + \text{last}) / 2$

② 分左半邊 $\text{mid} - 1$ 和右半邊 $\text{mid} + 1$

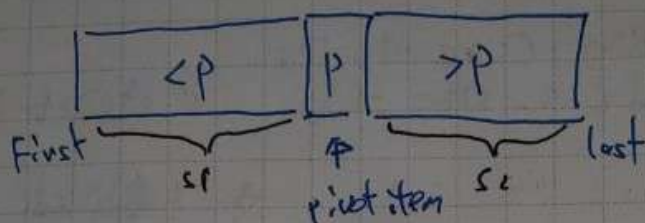
找第 k^{th} 小的 Item

一 用递归

- ① 設定特別的資料 P (pivot item)
- ② 以 P 為中心分為大於 P 和 小於 P

樞紐

※ 還可以排序
※ 只找取中一邊



河內塔 Hanoi

solve Towers (count, source, destination, spare)

if (count == 1)

// 把最下面的移动到目標

else {

solve Towers (count - 1, source, spare, destination)

solve Towers (1, source, destination, spare)

solve Towers (count - 1, spare, destination, source)

}

}

刻度尺 畫幾個虛線

drawTicks (length)

if (length > 0)

drawTicks (length - 1)

drawTick of the length

drawTicks (length - 1)

當 length = 4



void drawRuler (int inches, int majorLength) {

drawOneTick (majorLength, 0);

for (int i = 1; i < inches; i++) {

drawTick (majorLength - 1);

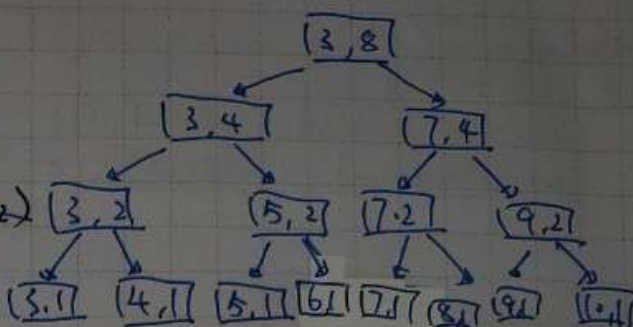
Binary Recursion

※ 431526

```
int sumB(int a, int n) {
    if (n == 1)
        return a;
    return sumB(a, n/2)
        + sumB(a*n/2, n-n/2);
}
```

起點

數量



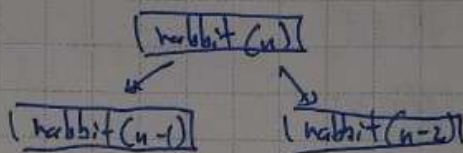
遞迴求子數量

方法①

※ 作業-寫法

$\text{rabbit}(n) = \text{rabbit}(n-1) + \text{rabbit}(n-2)$
 $\text{rabbit}(2) = 1$
 $\text{rabbit}(1) = 1$

★ Fibonacci sequence
(斐波那契數列)



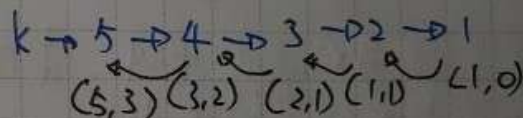
★ code

```
if (n == 1 or n == 2)
    return 1
else if (n > 2)
    return rabbit(n-1) + rabbit(n-2)
```

★ 效率不高

方法② (線性遞迴)

```
if k = 1 then
    return (k, 0) // base cases: k=1 -> (F1, F0)
else
    C(i, j) = linearFibonacci(C(k-1)) // (Fk-1, Fk-2)
    return C(i+j, i) // (Fk = Fk-1 + Fk-2, Fk-1)
```



費氏數列

$f(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } 2 \\ f(n-1) + f(n-2) & \text{if } n \geq 3 \end{cases}$

※

$\text{fib}[i] = \text{fib}[i-1] + \text{fib}[i-2]$

★ 尾遞迴 (tail recursion)

```
void writeBackward (string s, int size) {  
    if (size > 0) {  
        cout << s.substr(size-1, 1);  
        writeBackward (s, size-1);  
    }  
}
```

← Tail recursion
遞迴中最後執行的

Vector (STL) 用法

- vec.push_back() 新增元素到 Vector 的尾端
- vec.pop_back() 刪除最後一個元素
- vec.insert() 插入一個或多個元素
- vec.erase() 刪除 vector 中的元素

堆疊溢位

- 使用過多的記憶體時導致呼叫堆疊產生的溢位
- 產生過多的函式呼叫，導致使用的呼叫堆疊大小超出事先規畫的大小

尾遞迴好處

- ★ 函數的最後一個動作是返回一個函式的呼叫結果
- 執行效率被極大地最佳化 → 效能最佳化

Ch2

A class combines

■ Attributes of objects of a single type

- Typically data
- Called data members

屬性

■ Behaviors (operations)

- Typically operate on the data
- Called methods or member function

運算

principles of Object-Oriented Programming

- Three characteristics

■ Encapsulation

- objects combine data and operations
- Hides inner details

■ Inheritance

- classes can inherit properties from other classes
- Existing classes can be reused

■ polymorphism

- Objects can determine appropriate operations at execution time.

Abstract Data Type: motives

□ Modularity 模組化

- Easier to write
- " " read
- " " modify

Achieve a Better solution

- 高內聚
- 低耦合

Functional abstraction 功能性的抽象化

- 描述
- 實作

Information hiding

- 資訊隱藏 (實作的部分)

The ADT List

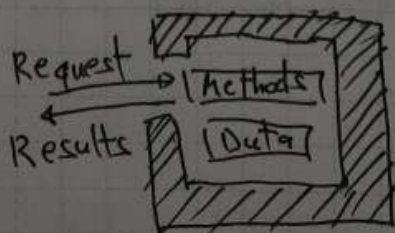
- | | | |
|--------|------|--------|
| ① 建構 | ② 解構 | ③ 是否為空 |
| ④ 計算個數 | ⑤ 插入 | ⑥ 刪除 |
| | | ⑦ 檢索 |

The ADT Sorted List

- | | | |
|--------|--------|------|
| ① 是否為空 | ② 計算個數 | ③ 新增 |
| ④ 刪除 | ⑤ 檢索 | ⑥ 定位 |

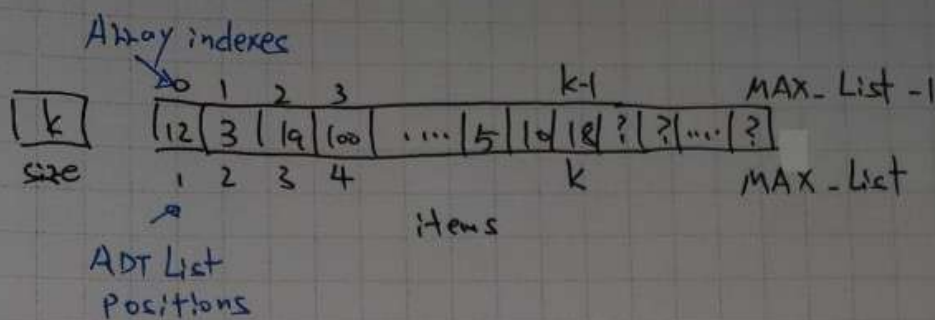
C++ class

- ① 封裝
- ② 成員
- ③ 私密公開



An Array-Based ADT List

- A list's k^{th} item is stored in $\text{items}[k-1]$



ADT Polynomial

- Operations

1. degree()

2. coefficient (in power)

3. change coefficient (in newCoefficient, in power)

① 多项式最高次项的指数

② Power 项的系数

③ 将 Power 项的系数改变为 Coefficient

※ 多项式各项的指数均为非负整数

Linked List Basics

陣列 → 需要移動資料

鏈結串列 → 不需要移動資料

Pointers

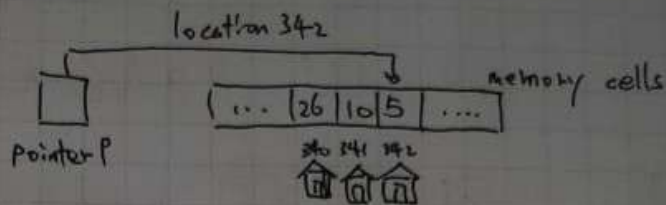
指標 = 門牌

`int *p;`

// p 存的是門牌號

E.g `p = 342`

找到 342 房子用的資料 (5)



// The new operator // 申請新房子
`p = new int;`

// 緊急配置 * 壞壞壞:
有可能搶不到
記憶體
`std::bad_alloc`

The address-of operator &

`p = &x;` // &x 是房子 x 的門牌

`delete p;` // 房子內部不清理, 歸還房子

`p = Null;` // 遺忘門牌 (清空, 避免誤用)



若使用 `new int`,
刪除使用 `delete`;

pointer 例子

* ? 代表里面的東西不存在

a) `int *p, *q;` `int x;`
 `p` `q` `x`

p 和 q 是指標變數 (空白的門牌)
x 是整數變數

b) `p = &x;` `p` → `x or *p`

獲取 x 的門牌

c) `*p = 6;` `p` → `6`
 `x or *p`

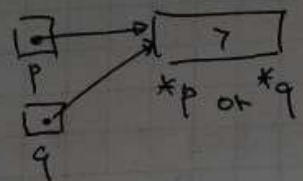
在獲取門牌後更改房子的
內容 * 必須先執行 b

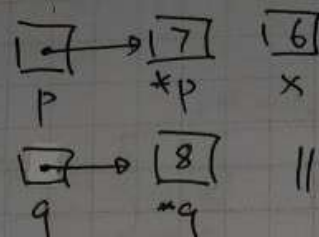
d) `p = new int;` `p` → `*p` `x`

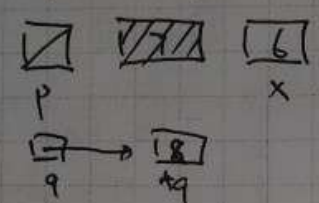
x 里面有 6 (自己的房子),
因此 `p = new int` 開一個新的房子

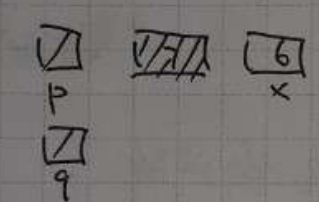
e) `*p = 7;` `p` → `*p` `x`

堆放家當

f) $q = p;$  P和q copy
指向同一個地方

g) $q = \text{new int};$
 $*q = 8$  // 房子有3棟
// q有自己房子, 和p分开

h) $p = \text{Null};$
// 錯誤寫法  // 沒有其他變數
7不會被用到

i) $\text{delete } q;$
 $q = \text{Null};$
// 歸還房子並忘記門牌  // 正確寫法

Arrays 動態(配置)陣列

$\text{double } * \text{anArray} = \text{new double}[\text{arraySize}];$ ¹⁵⁰

其他寫法

$\text{int anArray}[2] = * (\text{anArray} + 2)$ 陣列名稱 = 指標

把舊的Array搬到新的Array

$\text{double } * \text{oldArray} = \text{anArray};$ // 舊
 $\text{anArray} = \text{new double}[3 * \text{arraySize}];$ // 新

for (int i=0; i < arraySize; i++) // 一個一個搬
 $\text{anArray}[i] = \text{oldArray}[i];$

$\text{delete [] oldArray};$ // 歸還舊社區

Save / Copy a File

03-05

作業 2

include <stdio.h> // 讀檔
// 開啟指令

outfile = fopen(fileName.c_str(), "a"); // open a file to write

寫資料到檔案內

通常會傳回號碼或是錯誤狀態

→ 檢查 outfile 是不是錯誤

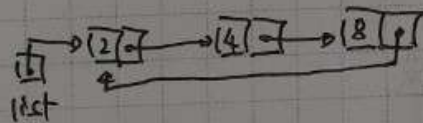
if (outfile != NULL) // 若正確讀入
// do something

資料型態

```
typedef struct student {  
    char sid[SID_LEN];  
    int score;  
}
```

```
void saveFile(FILE* fp, studentType d[], int No){  
    for (int i=0; i<No; i++){  
        fwrite (&d[i], sizeof(d[i]), 1, fp);  
        cout << d[i].sid << " " << d[i].score << endl;  
    } // for  
    fclose(fp);  
}
```

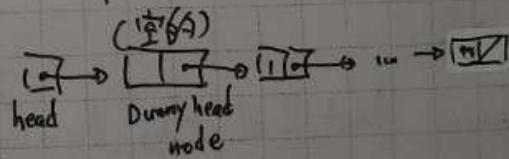
① Circular Linked Lists



最後一個節點的 next 指向
第一個節點

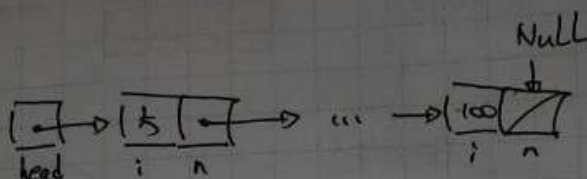
→ 隨時得到最後一個和第一個

② Dummy Head Node

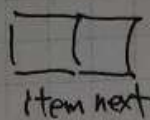


0307 0308

```
struct Node {
    int item;
    Node *Next;
};
```



```
Node *p; // 節點
p = new Node; // 新的
```



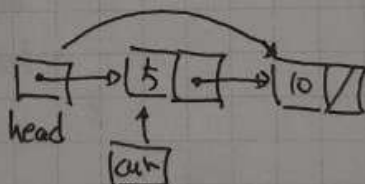
★ 把 Linked List 內容 output

```
for (Node *cur = head; cur != Null; cur = cur->Next)
    cout << cur->item << endl;
```

// 從頭(head)走到最後(Null) → output

Deleted node

```
cur->next = Null;
delete cur;
cur = Null;
```



★ 刪除分是不是頭部

ADT List

```
List (const List &alist); // 複製建構
```

```
~List (); // Destructer 解構
```