

計算機程式設計

C語言 Array

郭忠義

jykuo@ntut.edu.tw

臺北科技大學資訊工程系

陣列

- ❑ 陣列是相同型別之元素所組成的集合
- ❑ 在 C 語言中，陣列使用前必須先宣告
- ❑ 一維陣列宣告
 - 陣列個數必須是整數常數

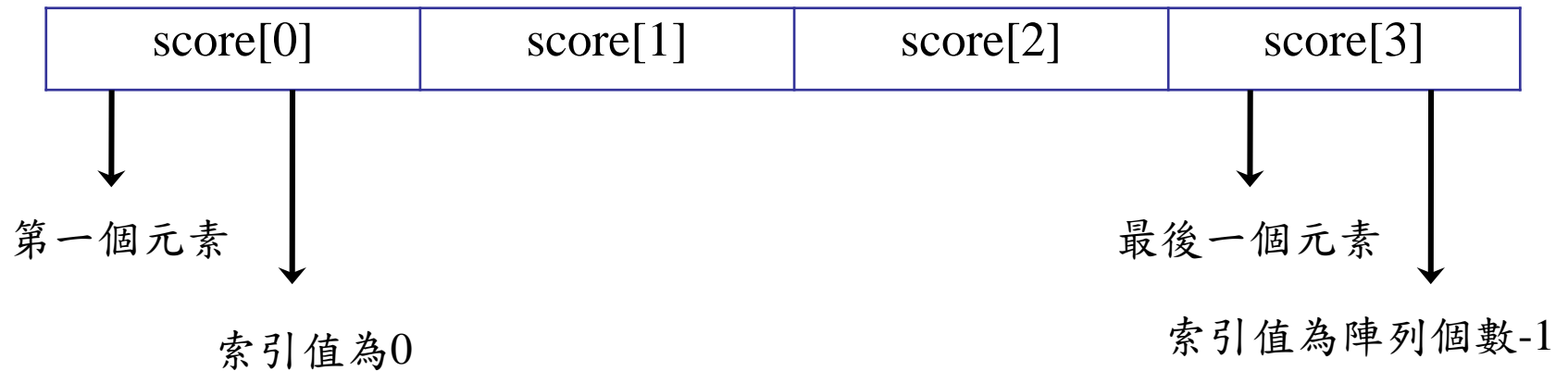
資料型別 陣列名稱[個數];

```
int score[4];      /* 宣告整數陣列score，可存放4個元素 */  
float temp[7];     /* 宣告浮點數陣列temp，可存放7個元素 */  
char name[12];     /* 宣告字元陣列name，可存放12個元素 */
```

陣列

- ❑ 陣列中的元素是以索引值來標示存放的位置
- ❑ 陣列索引值必須由0開始

```
int score[4];
```



陣列的記憶空間及大小

❑ 查詢陣列所佔的記憶空間

`sizeof (陣列名稱)` 查詢陣列所佔的位元組

```
01 int main() {  
02     double data[4];  
03     printf("陣列所佔位元組%d", sizeof(data));  
04     printf("陣列元素所佔位元組%d", sizeof(data[0]));  
05     printf("陣列個數%d", sizeof(data)/sizeof(double));  
06     return 0;  
07 }
```

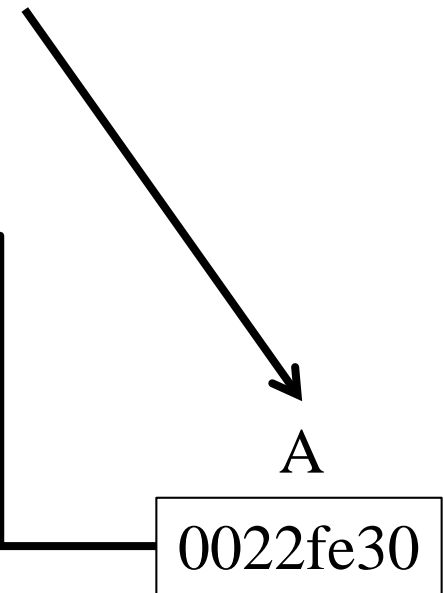
陣列所佔位元組32
陣列元素所佔位元組8
陣列個數4

陣列的位址

- 陣列的名稱是一個指標常數，它指向該陣列的位址
 - 陣列名稱A是一個指標常數，其值不能被更改

```
int A[4] = {5, 3, 6, 1};
```

A[0]	5	0022fe30
A[1]	3	0022fe34
A[2]	6	0022fe38
A[3]	1	0022fe3c



一維陣列A

陣列的位址

- ❑ C語言是以陣列第一個元素的位址當成是陣列的位址
- ❑ 陣列名稱本身就是存放陣列位址的變數

```
01 #define SIZE 4
02 int main() {
03     int A[SIZE] = {5, 3, 6, 1};
04     int i;
05     for(i=0; i<SIZE; i++)
06         printf("A[%d]=%2d,位址為%p\n", i, A[i], &A[i]);
07     printf("陣列A的位址=%p\n", A);
08     return 0;
09 }
```

```
A[0]= 5,位址為0022fe30
A[1]= 3,位址為0022fe34
A[2]= 6,位址為0022fe38
A[3]= 1,位址為0022fe3c
陣列A的位址=0022fe30
```

一維陣列基本操作

□ 初始值設定

```
int score[4]={66,23,22,1};  
int score[]={66,23,22,1};  
int score[4]={0}; // 給第一個元素值
```

```
01 #include <stdio.h>  
02 int main() {  
03     int score[4];  
04     score[0] = 66;  
05     score[1] = 23;  
06     2[score] = 22;    //也可以這樣寫//score[2] = 22;  
07     score[3] = 1;  
08     for(int i=0; i<=3; i++)  
09         printf("score[%d]=%d\n", i, score[i]);  
10     return 0;  
11 }
```

```
score[0] = 66  
score[1] = 23  
score[2] = 22  
score[3] = 1
```

一維陣列基本操作

□ 初始值設定

```
01  #include <stdio.h>
02  int main() {
03      int score[10]={0};    // 只會設定第一個元素為0
04      for(int i=0; i<10; i++)
05          score[i]=0;      // 使用迴圈設定每一個元素為0
06      for(i=0; i<10; i++)
07          printf("score[%d]=%d\n", i, score[i]);
08      return 0;
09  }
```


一維陣列基本操作

□ 由鍵盤輸入資料來設定陣列元素

```
01 int main() {  
02     int i, score[4];  
03     for(i=0; i<4; i++){  
04         printf("請輸入score[%d]的值", i);  
05         scanf(" %d ", &score[i]);  
06     }  
07     for(i=0; i<4; i++)  
08         printf("score[%d]=%d", i, score[i]);  
09     return 0;  
10 }
```

```
請輸入score[0]的值: 95  
請輸入score[1]的值: 100  
請輸入score[2]的值: 63  
請輸入score[3]的值: 77  
score[0]=95  
score[1]=100  
score[2]=63  
score[3]=77
```

傳遞一維陣列到函式

□ 一維陣列名稱為記憶體位址

- 呼叫函式時，傳遞陣列名稱，即傳遞陣列位址，函式不會為陣列配置空間，會為陣列變數配置空間

- 定義函式時，定義陣列變數，或指標變數

- `arr[i] = *(arr+i)`

```
01 #define SIZE 4
02 void show(int arr[]){ //(int arr[4]), (int *arr)等價
03     for(int i=0; i<SIZE; i++)
04         printf("%d ", arr[i]);
05         //printf(" %d", *(arr+i));
06 }
07 void change(int arr[]){
08     for(int i=0; i<SIZE; i++)
09         arr[i] = i;
10 }
```

```
12 int main() {
13     int A[SIZE] = {5, 3, 6, 1};
14     //int *A = {5, 3, 6, 1}; 錯誤
15     show(A); // show(&A[0])
16     change(A);
17     show(A);
18     return 0;
19 }
```

搜尋(Search)一維陣列

❑ 線性搜尋(Linear Search)/循序式搜尋(Sequential Search)

- 從第一筆資料開始搜尋比對，如果找到則傳回該值或該位置，如果沒找到則往下一筆資料搜尋比對，
- 例如，有一整數陣列的資料內容如下：

	0	1	2	3	4	5
data	13	25	16	23	57	66

❑ 如欲找出57，則：

- Step 1: 與陣列中第一筆資料比對， $57 \neq 13$
- Step 2: 與陣列中第一筆資料比對， $57 \neq 25$
- Step 3: 與陣列中第一筆資料比對， $57 \neq 16$
- Step 4: 與陣列中第一筆資料比對， $57 \neq 23$
- Step 5: 與陣列中第一筆資料比對， $57 = 57$


搜尋(Search)一維陣列

❑ 線性搜尋(Linear Search)/循序式搜尋(Sequential Search)

```
01 int search(int d[], int size, int key) {  
02     int i=0;  
03     for (i=0; i<size; i++)  
04         if (d[i]==key) return i;  
05     return -1;  
06 }  
07 int main() {  
08     int data[]={5,2,8,1,7,9,4,3,6};  
09     printf("%d, %d\n", search(data, 9, 1), 1);  
10     return 0;  
11 }
```

Exercise

- 輸入1~N整數，找第二小的數

```
#include <stdio.h>
int find2Min(int data[], int size) {
    int minIndex, index=0;
    
    return index;
}
int main() {
    int data[] = {5, 1, 3, 6, 2};
    printf("2 min=%d\n", find2Min(data, 5));
    return 0;
}
```

搜尋(Search)一維陣列

□ 二元搜尋(Binary Search)

- 陣列資料須要排序
- 每次都從範圍(left~right)的中間點 $mid = (left + right) / 2$ 找
- 中間點太大，往左找，中間點太小，往右找

假設找9

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	2	5	7	9	14	23	26
left=0			mid = 3 7 < 9 left = 3				right=7
					mid = 5 (7+3)/2 14 > 9 right = 5		
				mid = 4 (3+5)/2			

搜尋(Search)一維陣列

□ 二元搜尋(Binary Search)

```
01 #include <stdio.h>
02 int binarySearch(int d[], int left, int right, int key) {
03     int mid = 0;
04     while (left<right) {
05         mid = (left+right)/2;
06         if (d[mid]==key) return mid;
07         else if (d[mid]>key) right = mid;
08         else left = mid;
09     }
10     return -1;
11 }
12 int main() {
13     int data[]={ 1,2,5,7,9,14,23,26};
14     printf("%d, %d\n", binarySearch(data, 0,7,9), 9);
15     return 0;
16 }
```

大數相加

❑ 20位數以上整數加1，C語言 long long 無法表示30為整數

○ 使用整數陣列表示

➤ $100000000000000000000+1 = 100000000000000000001$

➤ $99999999999999999999+1 = 100000000000000000000$

```
#include<stdio.h>
int Add(int a[],int b[],int size){
    int i=0, carry=1;
    for (i=0; i<size; i++) {
        b[i] = (a[i] + carry)%10;
        carry = (a[i] + carry)/10;
    }
    if (carry ==1) {
        b[i] =1;
        size++;
    }
    return size;
}
```

```
void print(int a[], int size) {
    int i=0;
    for (i=size-1; i>=0; i--) printf("%d", a[i]);
    printf("\n");
}
void test01(){
    int a[]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1};
    int c[]={9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9};
    int b[22];
    Add(a, b, 20);
    print(a, 20);    print(b, 20);
    Add(c, b, 20);
    print(c, 20);    print(b, 21);
    return 0;
}
```


Exercise

- ❑ 20位數以上整數加1，C語言 long long 無法表示30位數整數
 - 使用整數陣列表示
 - $1000000000000000000001 + 99999999999999999999 = 1100000000000000000000$

```
#include<stdio.h>

int Add(int a[],int b[], int c[], int size){
    int i=0, carry=1;
    for (i=0; i<size; i++) {

    }
    if (carry ==1) {

    }
    return size;
}
```

```
void print(int a[], int size) {
    int i=0;
    for (i=size-1; i>=0; i--) printf("%d", a[i]);
    printf("\n");
}

void test01(){
    int a[]={1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1};
    int c[]={9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9};
    int b[22];
    Add(a, b, c, 20);
    print(a, 21);    print(b, 21);    print(c, 21);
}
```

Homework I

- 20位數以上整數加1，C語言 long long 無法表示30位數整數
 - 使用整數陣列表示50位數長整數
 - 計算兩個長整數的 $+$, $-$, $*$
 - 整數可以為正數與負數

二進位轉十進位

□ 運用位移運算子

```
#include <stdio.h>
int test(char s[]) {
    int sum=0, i=0;
    unsigned index=0x80; //1000 0000 的 16 進位
    for (i=0; i<=7; i++) {
        sum = sum + (s[i]-'0')*index; // s[i] * 2 的n次方，加總
        printf("%d,", sum);
        index >>= 1;
    }
    return sum;
}
int main() {
    printf("%d",test("01001010"));
    return 0;
}
```

Homework II

□ 分散度

- 輸入一串整數序列，計算此序列的 m 分散度。
- m 分散度定義為，序列中擁有長度為 m 且有 m 種不同數字的連續子序列之數量。
- 例如， $m=3$ ，序列 $\{1\ 2\ 3\ 5\ 4\ 5\ 4\}$ ，3 分散度數量 $\{1\ 2\ 3\}$, $\{2\ 3\ 5\}$, $\{3\ 5\ 4\}$ ，四個。

排序 (sorting) 一維陣列

- ❑ 將相同性質資料，由小至大或由大至小排列；身高、英文字典字詞順序、事件發生遠近、...，都可排序。
- ❑ 依資料存放位置的不同，可分：
 - 內部排序法 (internal sorting)：資料全部在主記憶體中。
 - 外部排序法 (external sorting)；主記憶體中只存放部分資料，大部分資料皆在外部記憶體 如硬碟檔案中。

選擇(selection)排序法

- 每次迴圈都挑出目前為排序資料的最小值
 - 利用n次迴圈完成排序。
 - 在第i次迴圈時，挑出第i小的資料將之與陣列中第i筆資料對調，重整陣列使成為由小至大排序的數列。

```
#include <stdio.h>
#define SWAP(x,y) {int t; t = x; x = y; y = t;}
int getMinIndex(int d[], int left, int right) {
    int i=0, minIndex = left;
    for ((i=left+1); i<right; i++) {
        if (d[i]<d[minIndex]) minIndex=i;
    }
    return minIndex;
}
void selectSort(int d[], int n) {
    int i=0, index=0;
    for (i=0; i<n; i++) {
        index = getMinIndex(d, i, n);
        printf("%d, %d\n", d[i], d[index]);
        SWAP(d[i], d[index]);
    }
}
```

插入排序法 (insertion sort)

- 將未排序資料逐一插入已排序的部分資料中。
 - 將資料分成已排序、未排序兩部份
 - 依序由未排序中的第一筆(target)，插入到已排序的適當位置
 - target與已排序資料比較，由左往右比，遇到的值比較高，則互換。

排序前：93,69,70,88,29,25,22,74,27,26

i=1: 處理值 = 69, 69 93 70 88 29 25 22 74 27 26
i=2: 處理值 = 70, 69 70 93 88 29 25 22 74 27 26
i=3: 處理值 = 88, 69 70 88 93 29 25 22 74 27 26
i=4: 處理值 = 29, 29 69 70 88 93 25 22 74 27 26
i=5: 處理值 = 25, 25 29 69 70 88 93 22 74 27 26
i=6: 處理值 = 22, 22 25 29 69 70 88 93 74 27 26
i=7: 處理值 = 74, 22 25 29 69 70 74 88 93 27 26
i=8: 處理值 = 27, 22 25 27 29 69 70 74 88 93 26
i=9: 處理值 = 26, 22 25 26 27 29 69 70 74 88 93

插入排序法 (insertion sort)

```
void print(int a[], int n) {
    for (int i=0; i<n; i++)
        printf("%d ", a[i]);
    printf("\n");
}

void insertionSort(int a[], int n) {
    int target=0, i=0, j=0;
    for (i=1; i<n; i++) {
        target = a[i];
        for (j=i; (a[j-1]>target) && (j>0); j--)
            a[j] = a[j-1];
        a[j] = target;
        print(a, n);
    }
}

int main() {
    int a[] = {89, 78, 54, 16, 64, 3, 47, 68, 90, 97};
    insertionSort(a, 10);
    return 0;
}
```


Homework III

□ 105APCS Q4

- 輸入棒球隊球員打擊結果，計算出隊得分。假設球員打擊情況：
 - 安打：以 1, 2, 3 和 H 代表一、二、三和全(四)壘打。
 - 出局：以 O 表示 (OUT)。
- 簡化版的規則如下：
 - 球場上有四個壘包，稱為本壘、一、二和三壘。
 - 本壘握球棒打的稱「擊球員」，在另外三個壘包的稱為「跑員」。
 - 當擊球員打擊「安打」時，擊球員與跑壘員可移動；「出局」時，跑壘員不動，擊球員離場換下一位。
 - 比賽開始由第 1 位打擊，當第 i 位球員打擊完後，由第 $(i+1)$ 位球員打擊。當第九位球員打擊完，則輪回第一位球員。
 - 打出 K 壘打時，場上球員(擊球員和跑壘員)會前進 K 個壘包。本壘到一壘，接著二、三壘，最後回到本壘。回到本壘可得 1 分。
 - 每達到三個出局數時，壘包清空(跑壘員都得離開)，重新開始。

Homework III

□ 105APCS Q4

○ 輸入格式

- 每組測試資料固定有十一行。
- 第一到九行，依照球員順序，每一行代表位球員的打擊資訊。每一行開始有一個正整數 a ($1 \leq a \leq 5$)，代表球員總共打 a 次。接下來有 a 個字元，依序代表每次打擊結果。資料間均以一個空白字元隔開。球員打擊資訊不會有錯誤與缺漏。
- 第十行有一個正整數 b ($1 \leq b \leq 27$)，表示想計算當總出局數累計到 b 時，該球隊得分。輸入打擊資訊中至少包含 b 個出局。
- 第十一行有一個正整數 m ($1 \leq m \leq 9$)，表示想計算第 m 個球員在總計第 b 個出局數安打數與到達本壘的次數。

○ 輸出格式

- 計算在總計第 b 個出局數發生時的總得分。
- 計算第 m 個球員在總計第 b 個出局數的安打數與到達本壘的次數。

Homework III

□ 105APCS Q4

輸入範例一

5 1 1 0 0 1

5 1 2 0 0 0

4 0 4 0 1

4 0 0 0 4

4 1 1 1 1

4 0 0 3 0

4 1 0 0 0

4 0 0 2 2

4 3 0 0 0

3

4

正確輸出

0

?

輸入範例一

5 1 1 0 0 1

5 1 2 0 0 0

4 0 4 0 1

4 0 0 0 4

4 1 1 1 1

4 0 0 3 0

4 1 0 0 0

4 0 0 2 2

4 3 0 0 0

6

7

正確輸出

5

?

氣泡排序法(bubble sort)

- 相鄰的兩元素要維持「上小下大」的順序關係；相鄰的兩元素會互相比較大小，將較大的資料往下放。

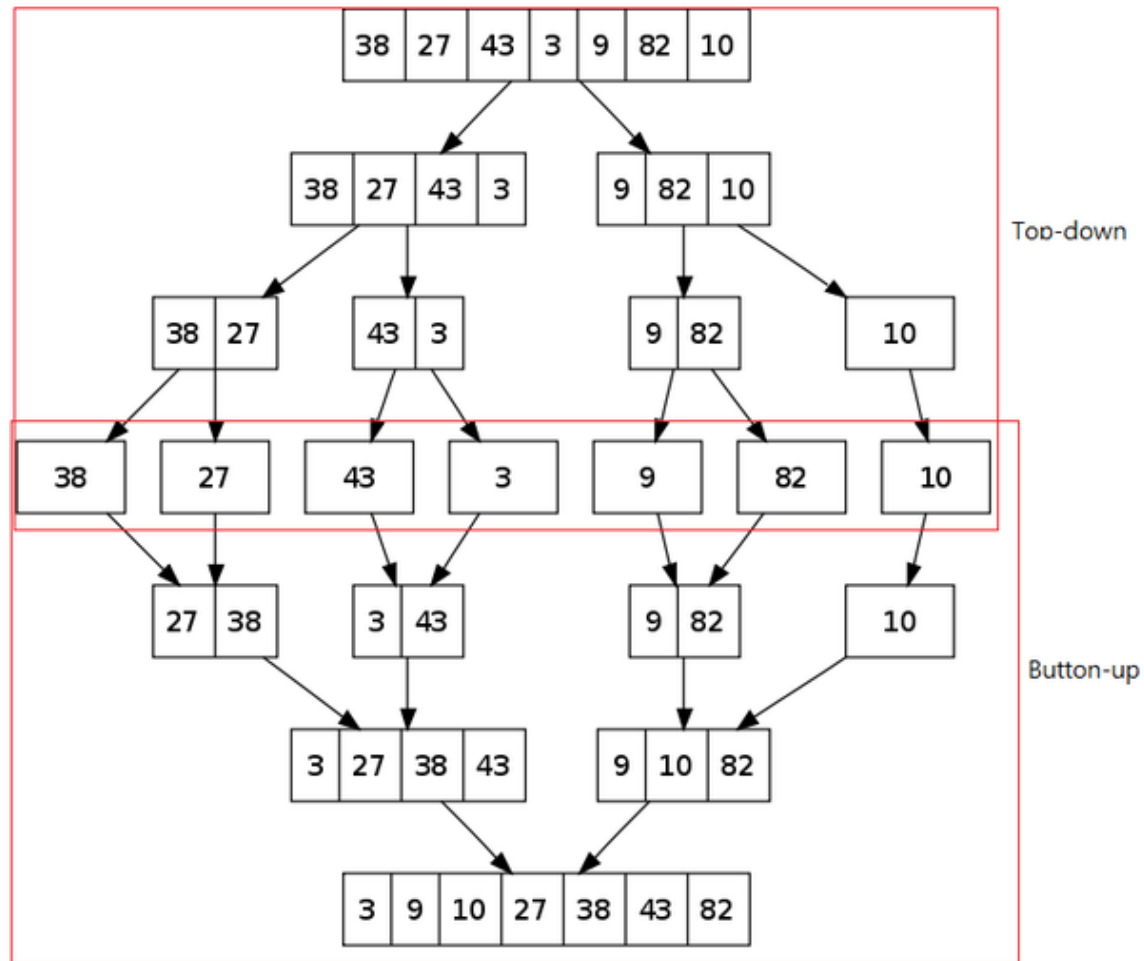
```
1  for (i=n-1; i>0; i--)  
2    for (j=1; j<=i; j++)  
3      if (data[j-1]>data[j])  
4        swap(&data[j-1], &data[j]);
```

索引	0	1	2	3	4	5	6	7	8	9
數值	69	81	30	38	9	2	47	61	32	79

播放動畫

合併排序法(merge sort)

- 將兩組已各自排序好的數列予以合併，使成為一完整的排列數列。



合併排序法(merge sort)

```
1  #include <stdio.h>
2  void print(int a[], int n) {           //印出陣列a[0]~a[n-1]
3      for (int i=0; i<n; i++) printf("%d ", a[i]);
4      printf("\n");
5  }
6  void copy(int C[], int a[], int m, int n) { // 將陣列C[m~n]複製到陣列a[m~n]
7      for (int i=m; i<=n; i++) a[i] = C[i];
8  }
9  void merge(int C[], int A[], int am, int an, int B[], int bm, int bn) {
10     int k = am;                        //將陣列 A[am~an], B[bm~bn]合併成 C[am~]
11     while ((am<=an) &&(bm<=bn)) {
12         if (A[am]<=B[bm]) C[k++] = A[am++];
13         else C[k++] = B[bm++];
14     }
15     while (am<=an) C[k++] = A[am++];
16     while (bm<=bn) C[k++] = B[bm++];
17 }
```

合併排序法(merge sort)

```
18 void mergeSort(int a[], int m, int n) { //合併排序主程式，遞迴
19     int mid=0, C[20];
20     if (n>m) { // 陣列a長度大於1，可以切成兩塊
21         mid=(m+n)/2;
22         mergeSort(a,m, mid); // 切前半段 a[m~mid]
23         mergeSort(a,mid+1, n); // 切後半段 a[mid+1~n]
24         printf("\nm=%d, n=%d\n", m, n); // 印出a處理範圍 m, n
25         merge(C, a, m, mid, a, mid+1, n); // 合併a前半段與後半段 到 陣列 C
26         copy(C, a, m, n); // 將合併後的 C 再複製回 a
27         print(a, 7);
28     }
29 }
30 int main() {
31     int a[] = {34, 12, 30, 18, 78, 56, 25};
32     mergeSort(a, 0, 6);
33     return 0;
34 }
```

Homework I

□ 問題敘述

- 考慮一個數列 $A = (a[1], a[2], a[3], \dots, a[n])$ 。如果 A 中兩個數 $a[i]$ 和 $a[j]$ 滿足 $i < j$ 且 $a[i] > a[j]$ ，則說 $(a[i], a[j])$ 是 A 中的一個反序 (inversion)。定義 $W(A)$ 為數列 A 中反序數量。例如，在數列 $A = (3, 1, 9, 8, 9, 2)$ 中，共有 $(3, 1)$ 、 $(3, 2)$ 、 $(9, 8)$ 、 $(9, 2)$ 、 $(8, 2)$ 、 $(9, 2)$ 6個反序，所以 $W(A) = 6$ 。
- 給定數列 A ，計算 $W(A)$ 簡單方法是對所有 $1 \leq i < j \leq n$ 檢查數對 $(a[i], a[j])$ ，但在序列太長時，計算時間會超過給定時限。以下運用分而治之 (divide and conquer) 策略所設計更有效率方法。
 - 將 A 等分為前後兩個數列 X 與 Y ，其中 X 的長度是 $n/2$ 。
 - 遞迴計算 $W(X)$ 和 $W(Y)$ 。
 - 計算 $W(A) = W(X) + W(Y) + S(X, Y)$ ，其中 $S(X, Y)$ 是由 X 中的數字與 Y 中的數字構成的反序數量。

Homework I

- 以 $A = (3, 1, 9, 8, 9, 2)$ 為例， $W(A)$ 計算如下。
 - 將 A 分為兩個數列 $X = (3, 1, 9)$ 與 $Y = (8, 9, 2)$ 。
 - 遞迴計算得到 $W(X) = 1$ 和 $W(Y) = 2$ 。
 - 計算 $S(X, Y) = 3$ 。因為有三個反序 $(3, 2)$ 、 $(9, 8)$ 、 $(9, 2)$ 是由 X 中的數字與 Y 中的數字所構成。所以得到 $W(A) = W(X) + W(Y) + S(X, Y) = 1 + 2 + 3 = 6$ 。
- 請撰寫一個程式，計算一個數列 A 的反序數量 $W(A)$ 。
- 輸入格式：測試資料有兩列，第一列為一個正整數 n ，代表 A 的長度。第二列有 n 個不大於 10^6 的非負整數，代表 $a[1], a[2], a[3], \dots, a[n]$ ，數字間以空白隔開。
- 輸出格式： A 的反序數量 $W(A)$ 。可能超過 32-bit 整數範圍。

範例一：輸入

6

3 1 9 8 9 2

正確輸出

6

範例二：輸入

5

5 5 4 3 1

正確輸出

9

快速排序法(quick sort)

□ 概念

- 選取某個元素做為基準值，令此基準值為target。
- 將所有比target小的資料，都放在target左邊；
- 所有不比target小的資料都放在target右邊。

□ 步驟

- 選取第0個元素 69為基準
- 從最右邊往左找比基準69還小的元素32
- 從最左邊往又找比基準69還大的元素81
- 兩個元素交換

	●									
索引	0	1	2	3	4	5	6	7	8	9
數值	69	81	30	38	9	2	47	61	32	79

快速排序法(quick sort)

```
#define SWAP(x,y) {int t; t = x; x = y; y = t;}
void QuickSort (int data[], int left, int right) {
    int i, j, target;
    if (left>=right) return;
    i = left;
    j = right;
    target = data[left];
    while (i!=j) {
        while ((data[j]>target)&&(i<j)) j--; //從右邊開始找
        while ((data[i]<=target)&&(i<j)) i++; //從左邊開始找
        // 左邊開始找比基準點大，如果有找到又沒與從右邊的相遇
        // 表示 data[i]一定可以換到比較小的
        // 否則 data[i]一定是小的最邊緣，可以跟中間值交換
        if(i<j) SWAP(data[i], data[j]); //左右沒相遇則可交換
    }
    SWAP(data[left], data[i]) //i是中間值
    QuickSort(data, left, i-1); //處理左半邊
    QuickSort(data, i+1, right); //處理右半邊
}
```

快速排序法(quick sort)

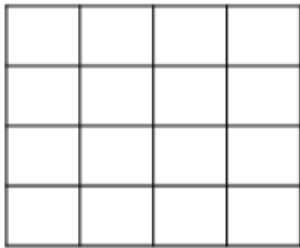
- 須從右邊right開始往左找 $j--$ ，找比中間點小的準備交換
 - 之後才可以左邊left開始往右找 $i++$ ，找比中間點大的交換

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
6	9 3	7	0	1	3 9
	i=1				j=5
6	3	7 1	0	1 7	9
		i=2		j=4	
6	3	1	0	7	9
		i=2	j=3		
			j=3, i=3		
先由右往左找 $j--$ ，再由左往右找 $i++$ ， $a[3] < 6$ ， $i=2+1=3$ 此時 $a[i=3]$ 與 $a[left=0]$ 交換，正確					
6 0	3	1	0 6	7	9

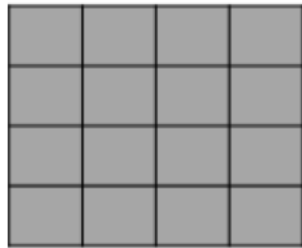
6	3	1	0	7	9
				j=4	
		i=2	i=3	i=4	
先由左往右找 $i++$ ， $i=3$ ， $j=4$ 時還可再 $i++$ ， $i=4$ ， 再由右往左找 $i=j$ ，仍然 $j=4$ 此時 $a[i=4]$ 與 $a[left=0]$ 交換將發生錯誤					
6 7	3	1	0	7 6	9

Exercise APCS

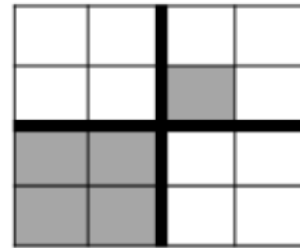
- DF - expression是儲存圖片資訊的表達方式。在一個 $n \times n$ 方陣中，若方格是白色記為0；黑色記為1；若方格可分為更小方格(左上、右上、左下、右下)，則記2，再依序(左上→右上→左下→右下)記錄這四個方格的資訊。
- 給定DF - expression的輸入，與這張圖像寬度(必為2的非負整數次方)，輸出這張圖中黑色格子的數量。



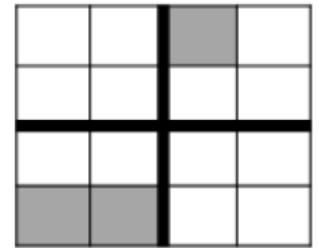
0



1



202001010



2021000200110

Exercise

□ Input

- 輸入第一行包含由0~2組成數組，代表某 DF - expression結果
- 第二行輸入一個正整數，代表圖像寬度，為2的非負整數次方
- 第一行必定為一個2後面接續4個0或1

□ Output

- 輸出整數，代表圖中黑色格子的數量

Sample Input 1

2020010120110

4

Sample Output 1

7

Sample Input 2

2020020100010

8

Sample Output 2

17

Exercise 遞迴

```
#include<stdio.h>
#include<string.h>
int square(char data[100] ,int length ,int n,int *i){
    int time=0, total=0;
    while((*i)<length){
        time++;
        if(data[*i]=='2'){
            (*i)++;
            total = total+ square(data ,length ,n/2 ,i);
        }
        else if(data[*i]=='1'){
            total = total+(n*n);
            (*i)++;
        }
        else if(data[*i]=='0'){
            (*i)++;
        }
        if(time==4) break;
    }
    return total;
}
```

```
void test01(){
    int n=0 ,length=0, i=0;
    char data[100];
    //scanf("%s",&data);  scanf("%d",&n);length = strlen(data);
    //printf("%d",square(data ,length ,n ,&i));
    printf("%d\n",square("202001010" ,9 ,4 ,&i)); i=0;//5
    printf("%d\n",square("2020020100010" ,13 ,8 ,&i));i=0;//17
    printf("%d\n",square("2020020100010" ,13 ,4 ,&i));i=0;//7
    printf("%d\n",square("2020010120110" ,13 ,8 ,&i));i=0;//28
    printf("%d\n",square("2021000200110" ,13 ,4 ,&i));i=0;//3
}
int main(){  test01();  return 0;}
```

Exercise 非遞迴

```
#include <stdio.h>
#include <string.h>
int count(int level) {
    int ans=1;
    for (int i=0; i<level;i++) ans = ans*2;
    return (ans*ans);
}
int f(int N, char data[]) {
    int level =0, index =0, sum=0;
    int len = strlen(data);
    int comp[100]={0};
    for (int i=0; i<100; i++) comp[i]=0;
    while (index <len) {
        if (data[index]=='2') level++;
        else if (data[index]=='1') {
            sum = sum + (N*N)/count(level);
        }
        if (data[index]!='2') comp[level]++;
    }
```

```
        if (comp[level]>=4) {
            comp[level]=0;
            level--;
            comp[level]++;
        }
        index++;
    }
    return sum;
}
int main() {
    printf("=>%d\n", f(4, "0"));           //0
    printf("=>%d\n", f(4, "1"));           //16
    printf("=>%d\n", f(2, "21100"));        //2
    printf("=>%d\n", f(4, "202001010"));    //5
    printf("=>%d\n", f(8, "2020020100010")); //17
    printf("=>%d\n", f(4, "2020010120110")); //7
    printf("=>%d\n", f(8, "2020010120110")); //28
    printf("=>%d\n", f(4, "2021000200110")); //3
    return 0;
}
```


二維陣列

❑ 二維陣列的宣告

資料型別 陣列名稱[列的個數][行的個數];

```
int data[10][5];      /* 可存放10列5行個整數 */  
float score[4][3];    /* 可存放4列3行個浮點數 */
```

❑ 二維陣列的宣告與資料初始化

```
int data[4][3] = {{1, 2, 3},{4, 5, 6}, {7, 7, 7},{7, 8, 9}};
```

Array Index

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)
(3,0)	(3,1)	(3,2)

Array Value

1	2	3
4	5	6
7	7	7
7	8	9

二維陣列

□ 二維陣列資料初始化

```
#include <string.h>
void *memset(void *str, int c, size_t n)
複製字元c (unsigned char) 參數str指向的字串的前n個字元(單位1Byte)。
```

```
#include <stdio.h>
#include <string.h>
void test() {
    char s1[] = "congratulation";
    char s2[][20] = {"Hi", "Hello", "Good"};
    printf("%s\n", memset(s1, '#', 10));
    printf("%s\n", s1);
    printf("%s\n", memset(s2, '#', 22));
    printf("%s\n", s2[0]);
    printf("%s\n", s2[1]);
}
```

```
#####tion
#####tion
#####llo
#####llo
##llo
```

多維陣列

- C語言把多維陣列看成是由低一維的陣列所組成的一維陣列
 - n維陣列是n-1維陣列所組成的一維陣列
 - 3維陣列是2維陣列所組成的一維陣列
 - 多維陣列名所指的元素是其低一維的陣列

```
int data[4][3] = {{1, 2, 3},{4, 5, 6}, {7, 7, 7},{7, 8, 9}};
```

Array Index

data[0]	(0,0)	(0,1)	(0,2)
data[1]	(1,0)	(1,1)	(1,2)
data[2]	(2,0)	(2,1)	(2,2)
data[3]	(3,0)	(3,1)	(3,2)

Array Value

1	2	3
4	5	6
7	7	7
7	8	9

```
data[0] = {1, 2, 3}  
data[1] = {4, 5, 6}  
data[2] = {7, 7, 7}  
data[3] = {7, 8, 9};
```

二維陣列基本操作

□ 二維陣列的給值與輸出

```
#define ROW 2
#define COL 4
int main() {
    int i, j;
    int arr[ROW][COL];
    for(i=0; i<ROW; i++){
        for(j=0; j<COL; j++){
            arr[i][j] = i*j;
        }
    }
    for(i=0; i<ROW; i++){
        for(j=0; j<COL; j++) {
            printf("%d*%d=%d\t", i, j, arr[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

二維陣列操作

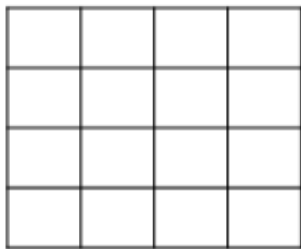
□ 傳遞二維陣列到函式

```
01 #define ROW 4
02 #define COL 3
03 void search(int arr[][COL], int p[]){
04     int i, j;
05     p[0] = p[1]=arr[0][0];
06     for(i=0; i<ROW; i++){
07         for(j=0; j<COL; j++){
08             if(p[0]<arr[i][j]) p[0] = arr[i][j];
09             if(p[1]>arr[i][j]) p[1] = arr[i][j];
10         }
11     }
12 }
13 void show(int a[][COL]) {
14     int i, j;
15     for(i=0; i<ROW; i++){
16         for(j=0; j<COL; j++)
17             printf("%2d", a[i][j]);
18         printf("\n");
19     }
20 }
```

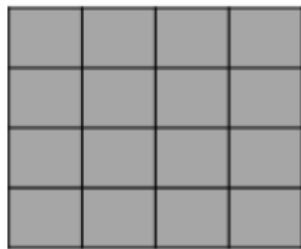
```
21 int main() {
22     int a[ROW][COL]= {{1, 2, 3}, {4, 5, 6},
23                        {7, 8, 9}, {5, 4, 3}};
24     int i, j, b[2];
25     show(a);
26     search(a, b);
27     printf("max = %d, min = %d\n", b[0], b[1]);
28 }
```

Homework II

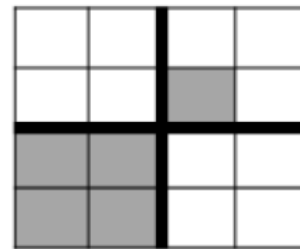
- DF - expression是儲存圖片資訊的表達方式。在一個 $n \times n$ 方陣中，若方格是白色記為0；黑色記為1；若方格可分為更小方格(左上、右上、左下、右下)，則記2，再依序(左上→右上→左下→右下)記錄這四個方格的資訊。
- 給定DF - expression的輸入，與這張圖像寬度(必為2的非負整數次方)，輸出這張圖中黑色格子的座標位置。



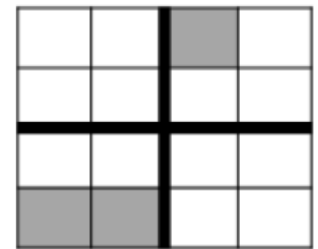
0



1



202001010

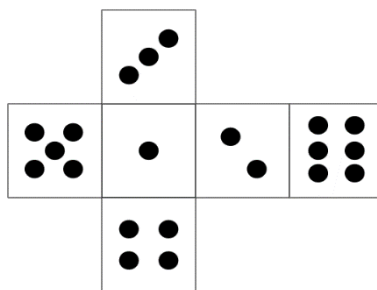
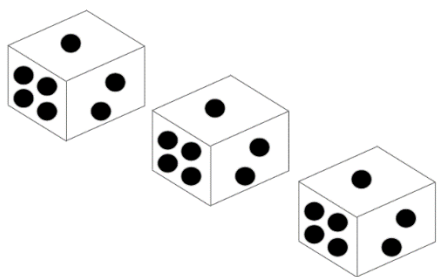


2021000200110

Exercise 骰子I

□ APCS2020骰子

- 給定 n 個骰子排成一列，一開始都是點數 1 朝上，點數 4 朝前，點數 2 朝右 (如下左圖所示)，另外骰子的展開圖如下右圖所示。



○ 輸入說明

- 第一行包含兩個正整數 n, m ($1 \leq n \leq 20, 1 \leq m \leq 100$)。
 - n 代表骰子的個數， m 代表針對 n 個骰子共有幾次操作。
- 接下來 m 行每行有兩個整數，第 i 行兩個整數 a, b 表示第 i 次操作。
 - 若 a, b 都是正整數，交換編號 a 與編號 b 的骰子的位置。
 - 若 b 為 -1 ，將編號 a 的骰子向前旋轉。
 - 若 b 為 -2 ，將編號 a 的骰子向右旋轉。

Exercise 骰子I

○輸出說明

- 在一行輸出 n 個數字以空格分隔，
- 在 m 次操作結束後，依序輸出編號 1 到編號 n 的骰子朝上的點數。

	範例輸入	範例輸出
1	1 2 (1個骰子操作 2 次) 1 -2 (編號 1 骰子向右旋轉) 1 -1 (編號 1 骰子向前旋轉)	3
2	3 3 (3個骰子操作 3 次) 2 -1 (編號 2 骰子向前旋轉) 3 -2 (編號 3 骰子向右旋轉) 3 1 (編號 3 1 骰子位置交換)	5 3 1

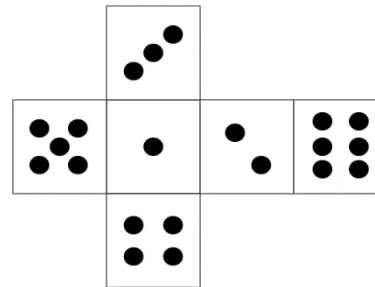
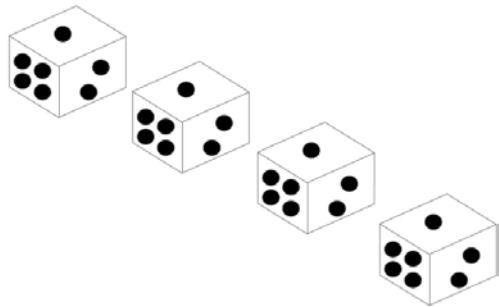
Exercise 骰子I

```
#include <stdio.h>
#define CHANGE(x, y) { int temp = x; x = y; y = 7-temp; }
#define SWAP(x, y) { int temp = x; x = y; y = temp; }
void roll(int data[], int b) {
    if (b==1) CHANGE(data[0], data[1])
    else if (b==2) CHANGE(data[2], data[1])
}
void f1() {
    int n=0, m=0, a=0, b=0;
    int data[20][3];
    for (int i=0; i<20; i++) {
        data[i][0]=4; data[i][1]=1; data[i][2]=2;
    }
    scanf("%d %d", &n, &m);
    for (int i=0; i<m; i++) {
        scanf("%d %d", &a, &b);
        if (a>=0 && b>0) {
            SWAP(data[a-1][0], data[b-1][0]);
            SWAP(data[a-1][1], data[b-1][1]);
            SWAP(data[a-1][2], data[b-1][2]);
        }
        else roll(data[a-1], b);
    }
    for (int i=0; i<n; i++)
        printf("%d %d %d\n", data[i][0], data[i][1], data[i][2]);
}
```

Homework III 骰子II

□ 骰子

- 十八骰子是一種常見的擲骰子遊戲，用四顆骰子計點。四顆一開始都是點數 1 朝上，點數 4 朝前，點數 2 朝右 (如下左圖所示)，骰子展開如下右圖所示。



- 接下來 N 次修改操作，每次操作包含四個整數 a, b, c, d 代表四個骰子操作。 a, b, c, d 意義為：
 - 1 表示向前滾一次。
 - 2 表示向後滾一次。
 - 3 表示向右滾一次。
 - 4 表示向左滾一次。

Homework III 骰子II

○操作修正完成，請輸出點數，計點方式：

- 若四顆點數均相同，稱一色，計18點，例如6, 6, 6, 6 或 3, 3, 3, 3。
- 若四顆點數均不同；或有三顆點數相同，一顆不同，計 0點，例如 1, 2, 3, 4 或 2, 2, 2, 6。
- 若兩顆點數相同，另兩顆點數也相同，但兩組兩顆點數不同，則點數計算為 - 加總兩顆較大點數，例如 2, 2, 5, 5，加總兩顆較大點數為 $5+5=10$ 點。
- 若兩顆點數相同，另兩顆點數不同，則點數計算為加總兩顆不同點數，例如 2, 2, 4, 5，加總兩顆不同點數為 $4+5=9$ 點。

輸入說明

第一行正整數 N

接下來N行每行四個正整數，第i行表示第i次操作。

輸出說明

最後朝上計點點數。

輸入範例	輸出範例
1	18
1 1 1 1	
2	18
1 2 3 4	
1 2 3 4	
2	0
1 2 3 4	
4 3 2 1	

輸入範例	輸出範例
3	10
1 2 3 4	
1 2 3 4	
3 1 4 2	
2	6
1 2 3 4	
2 3 1 1	

Homework III 骰子II

```
#include <stdio.h>
#define CHANGE(x, y) { int temp = x; x = y; y = 7-temp; }
#define SWAP(x, y) { int temp = x; x = y; y = temp; }
void roll(int data[], int b) {
    if (b==1) CHANGE(data[0], data[1])
    else if (b==2) CHANGE(data[1], data[0])
    else if (b==3) CHANGE(data[2], data[1])
    else if (b==4) CHANGE(data[1], data[2])
}
```

```
int countSame(int data[4][3], int sum[7]) {
    int maxIndex=0;
    for (int i=1; i<=6; i++) {
        for (int j=0; j<4; j++) {
            if (data[j][1]==i) sum[i]++;
        }
        if (sum[i]>=sum[maxIndex]) maxIndex=i;
    }
    return maxIndex;
}
int sameSum(int sum[], int index) {
    int num=0;
    for (int i=1; i<=6; i++)
        if (sum[i]==sum[index]) num++;
    return num;
}
```

```
int normalPoint(int sum[]) {
    int point=0;
    for (int i=1; i<=6; i++)
        if (sum[i]==1) point=point+i;
    return point;
}
```

Homework III 骰子II

```
int getPoint(int data[4][3]) {
    int sum[7]={0,0,0,0,0,0,0};
    int maxIndex=0, point=0;
    maxIndex = countSame(data, sum);
    printf("%d %d %d %d %d %d\n", sum[1], sum[2], sum[3], sum[4], sum[5], sum[6]);
    if (sum[maxIndex]==4) point=18;
    else if ((sum[maxIndex]==1)||(sum[maxIndex]==3)) point=0;
    else if ((sum[maxIndex]==2)&&(sameSum(sum, maxIndex)==2)) point=2*maxIndex;
    else if (sum[maxIndex]==2) point=normalPoint(sum);
    return point;
}

void game() {
    int m=0, a[4], data[4][3];
    for (int i=0; i<4; i++) {
        data[i][0]=4; data[i][1]=1; data[i][2]=2;
    }
    scanf("%d", &m);
    for (int i=0; i<m; i++) {
        scanf("%d %d %d %d", &a[0], &a[1], &a[2], &a[3]);
        for (int j=0; j<4; j++) roll(data[j], a[j]);
    }
    for (int i=0; i<4; i++)
        printf("%d %d %d\n", data[i][0], data[i][1], data[i][2]);
    printf("point=%d\n", getPoint(data));
}
```

Exercise

□ APCS2019

- 給定一張二維平面圖，起點為整張圖權重最小的點，在此平面圖上從起點開始走出一條路徑，求此路徑經過點的總權重。往下一個點走避須遵循規則：
 - (1) 從相鄰的點選擇一個權重最小的點往下走，相鄰的點是上、下、左、右。
 - (2) 走過的點不能重複。
- 例如以下平面圖，從1開始走，接著走6，再走13，最後走到9。
- 總權重為 $1+6+13+4+12+5+14+7+11+3+16+9 = 101$

3	16	9	1
11	7	14	6
10	15	5	13
2	8	12	4

Exercise

□ APCS2019

```
#include <stdio.h>
#define maxN 105
const int INF = 0x3f3f3f3f;          // INF = 1061109567 , 大於值域
typedef long long LL;
int mp[maxN][maxN];
int min(int x, int y) {
    int r = x>y?y:x;    return r;
}
LL dfs ( int x, int y, LL sum ){      // 先找出最低點
    int mi = min(min(mp[x+1][y], mp[x-1][y] ), min(mp[x][y+1], mp[x][y-1]));
    sum += mp[x][y];
    mp[x][y] = INF;
    if ( mi == INF )                return sum;
    if ( mp[x + 1][y] == mi )        return dfs ( x + 1, y, sum );
    if ( mp[x - 1][y] == mi )        return dfs ( x - 1, y, sum );
    if ( mp[x][y + 1] == mi )        return dfs ( x, y + 1, sum );
    if ( mp[x][y - 1] == mi )        return dfs ( x, y - 1, sum );
}
```

Exercise

□ APCS2019

```
int main(){
    int n, m, x, y, mi = INF, i=0, j=0;
    memset ( mp, INF, sizeof mp ); // INF 可以 0x3f 取代，因一次寫一個char
    scanf("%d %d", &n, &m);
    // 1 index，簡化邊界判斷，邊界都是 INF = 1061109567
    for (i = 1 ; i <= n ; i++ ) {
        for (j = 1 ; j <= m ; j++){
            scanf("%d", &mp[i][j]);
            if (mi > mp[i][j])
                mi = mp[i][j], x=i, y=j;
        }
    }
    printf("%d\n",dfs(x, y, 0));
    return 0;
}
```

INF	INF	INF	INF	INF	INF
INF	INF	INF	INF	INF	INF
INF	INF	INF	INF	INF	INF
INF	INF	INF	INF	INF	INF
INF	INF	INF	INF	INF	INF
INF	INF	INF	INF	INF	INF

Exercise

□ memset

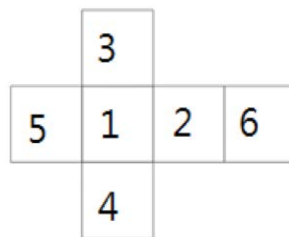
- 以位元組Byte為單位設定其值，即後8位元二進位進行賦值。
- 1的二進位是(00000000 00000000 00000000 00000001)，取後8位(00000001)，int型占4個Byte，當初始化為1時，它把一個int的每個位元組都設置為1，也就是0x01010101,二進位是00000001 00000001 00000001 00000001，十進位就是16843009。
- 輸入0,-1時正確初始化0,-1，純屬巧合。
 - 0，二進位是(00000000 00000000 00000000 00000000)，取後8位(00000000)，初始化後00000000 00000000 00000000 00000000是0
 - -1，負數在電腦中以補數存儲，二進位是(11111111 11111111 11111111 11111111)，取後8位(11111111)，則是11111111 11111111 11111111 11111111結果也是-1
- 總結：memset()只有在初始化-1, 0時才會正確。

Exercise

- 三個 5×5 矩陣相加，相減
- 隨意矩陣大小(10以下)的加、減、乘。考慮矩陣維度適當性，若使用者填錯要顯示錯誤訊息並要求重來。

Homework V 魔術方塊

- 魔術方塊有六面，一開始白色朝前，藍色朝上，紅色2朝右(如下左圖所示)，展開如下右圖所示。



- 顏色編號1: 藍色，2: 紅色，3: 黃色，4: 白色，5: 橘色，6: 綠色。
- 接下來 M 次修改操作，每次操作包含一個整數 a，意義為：
 - 1. 10 表示直欄(column)最左邊向前轉一次。
 - 2. 11 表示直欄(column)中間向前轉一次。
 - 3. 12 表示直欄(column)最右邊向前轉一次。
 - 4. 20 表示橫列(row)最上面向左轉一次。
 - 5. 21 表示橫列(row)中間向左轉一次。
 - 6. 22 表示橫列(row)最下面向左轉一次。

Homework V 魔術方塊

- ❑ 操作修正完成，請輸出朝上9格的顏色編號。

輸入說明	輸出說明
第一行正整數 M 接下來M行每行一個正整數，第i行表示第i次操作。	最後朝上計點點數。
輸入範例	輸出範例
1	4 1 1
10	4 1 1 4 1 1
3	2 1 1
10	6 1 1
20	6 1 1
10	
4	4 1 4
10	4 1 2
21	4 1 4
12	
20	

Homework I

□ 五子棋

- 檢查10*10五子棋可以構成5個連為一線的位置。1表示有放棋子，0表示沒有放棋子，如右圖。
- 可以增加5個連為一線，以下圖表示。

```
000000x010
0000001100
0000001000
00000x1111
0000101000
000x00x000
0001000000
0001000000
0x1111x000
0001000000
```

其位置為

06
35
53
56
81
86

```
0000000010
0000001100
0000001000
0000001111
0000101000
0000000000
0001000000
0001000000
0011110000
0001000000
```

Homework IV

□ 旋轉矩陣

○ 設一個 $n \times n$ 的矩陣，由左而右，由上而下標示自 1 到 $n \times n$ 的數，如下圖為 4×4 的。

○ 讀入旋轉序列後，將該矩陣的資料輸出。

○ 右圖表示向右旋轉一次 R。

➤ (順時鐘)

13	9	5	1
14	10	6	2
15	11	7	3
16	12	8	4

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20

17	18	19	20
13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

○ 右圖表示向左旋轉一次 L，

➤ (逆時鐘)

4	8	12	16
3	7	11	15
2	6	10	14
1	5	9	13

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

○ 右圖表示上下對翻一次，N

13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

旋轉矩陣

□ 找出上下替換規則

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24
30	31	32	33	34
40	41	42	43	44

size = 5

0,s-1<-0,0

1,s-1<-0,1

2,s-1<-0,2

3,s-1<-0,3

4,s-1<-0,4

0,s-2<-1,0

1,s-2<-1,1

2,s-2<-1,2

3, s-2<-1,3

4, s-2<-1,4

0,s-3<-2,0

1,s-3<-2,1

2,s-3<-2,2

3,s-3<-2,3

4,s-4<-2,4

```
for (j=0; j<size; j++)
    m[j][size-1-0] = data[0][j];
for (j=0; j<size; j++)
    m[j][size-1-1] = data[1][j];
for (j=0; j<size; j++)
    m[j][size-1-2] = data[2][j];
for (j=0; j<size; j++)
    m[j][size-1-3] = data[3][j];
```

旋轉矩陣

❑ 找出旋轉規則

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24
30	31	32	33	34
40	41	42	43	44

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24
30	31	32	33	34
40	41	42	43	44

```
for j=0~4 [j][4]<-[0][j]
```

04-00

14-01

24-02

34-03

44-04

```
for j=0~4 [j][3]<-[1][j]
```

03-10

13-11

23-12

33-13

43-14

```
for j=0~4 [j][2]<-[2][j]
```

02-20

12-21

22-22

32-23

42-24

```
for j=0~4 [j][4]<-[0][j]
```

```
for j=0~4 [j][3]<-[1][j]
```

```
for j=0~4 [j][2]<-[2][j]
```

```
for j=0~4 [j][1]<-[3][j]
```



```
for i=0~3
```

```
  for j=0~4 [j][4-i]<-[i][j]
```


Homework VI

□ 旋轉矩陣

```
#include <stdio.h>
#define N 10
void show(int d[N][N], int size) {
    int i,j;
    for (i=0; i<size; i++) {
        for (j=0; j<size; j++) {
            printf("%3d ", d[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}
void init(int data[N][N], int size) {
    int i,j, c=1;
    for (i=0; i<size; i++) {
        for (j=0; j<size; j++) {
            data[i][j]=c++;
        }
    }
    show(data, size);
}
```

```
void mir(int data[N][N], int m[N][N], int size) {
    int i,j;
    
    show(m, size);
}
void change(int data[N][N], int m[N][N], int size) {
    int i,j;
    
    show(m, size);
}

int main() {
    int data[N][N];
    int m[N][N];
    init(data, 5);
    mir(data, m, 5);
    change(data,m, 5);
    return 0;
}
```

Homework VII

□ 騎車

○ 賴先生騎腳踏車挑戰一日N塔， $N < 10$ 。每一個塔位在編號 1, 2, 3, ..., N 城市中。兩兩個城市都有一段距離的公路相連。請計算從第 1 個城市出發，騎過每一個城市的最短距離。例如 $N = 5$ ，以下是兩兩城市之間公路的距離。例如

- 城市 1 和城市 2 的距離是 4，
- 城市 1 和城市 3 的距離是 2，
- 城市 3 和城市 4 的距離是 2，
- 城市 5 和城市 4 的距離是 3，

--	1	2	3	4	5
1	0	4	2	3	6
2	4	0	3	1	4
3	2	3	0	2	5
4	3	1	2	0	3
5	6	4	5	3	0

○ 則從城市 1 出發，騎完所有城市的距離最短是，經由 13245 的距離 = $2+3+1+3=9$ 。

輸入說明

第 1 筆資料是 N，

第 2 筆資料是第 1 個城市和其他城市的公路距離。

第 3 筆資料是第 2 個城市和其他城市的公路距離。

....

第 N+1 筆資料是第 N 個城市和其他城市的公路距離。

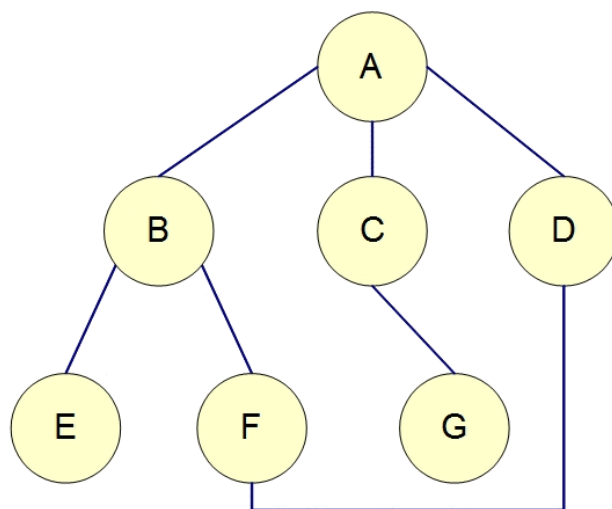
輸出說明

輸出從第 1 個城市出發，騎完所有城市最短距離。

圖論－廣度優先搜尋法

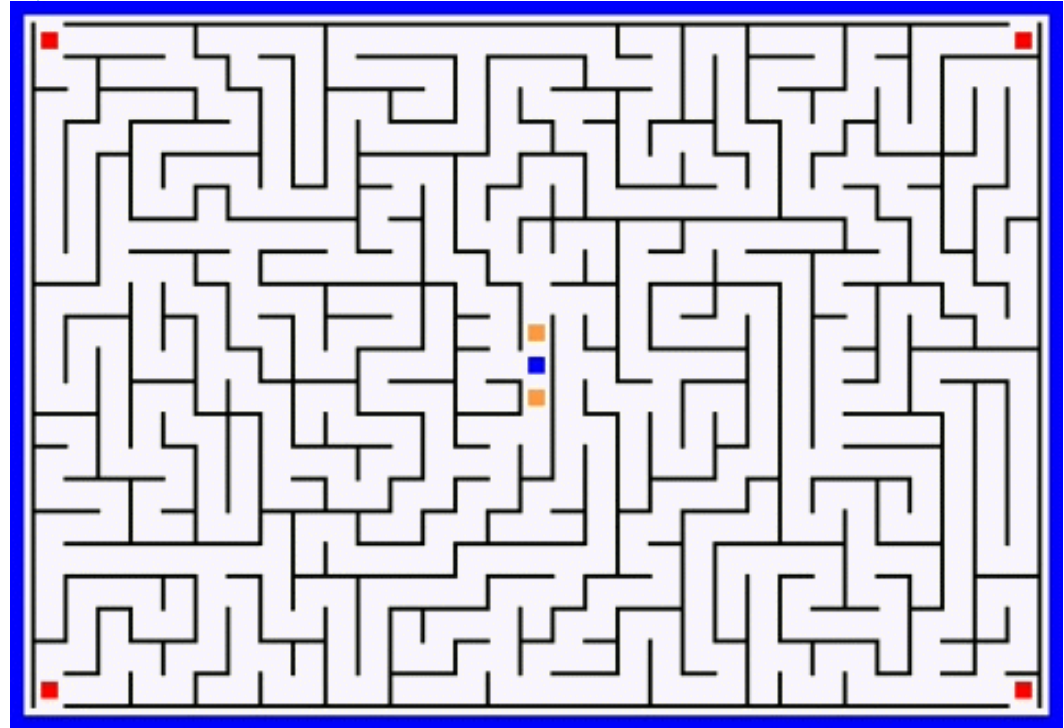
□ 廣度優先搜尋法 (Breadth-first Search)

- 從圖某節點(vertex, node)開始走訪，接著走訪此節點所有相鄰且未拜訪過的節點，
- 由走訪過節點繼續進行先廣後深搜尋。把同一深度(level)節點走訪完，再繼續向下個深度搜尋，直到找到目的節點或遍尋全部節點。
- 廣度優先搜尋法屬於盲目搜索(uninformed search)，可利用佇列(Queue)處理。



圖論－廣度優先搜尋法

```
procedure BFS(vertex s) {  
  create a queue Q  
  enqueue s onto Q  
  mark s as visited  
  while Q is not empty {  
    dequeue a vertex from Q into v  
    for each w adjacent to v {  
      if w unvisited {  
        mark w as visited  
        enqueue w onto Q  
      }  
    }  
  }  
}
```



圖論－廣度優先搜尋法

```
#include<stdio.h>
#define Max 100
#define N 6
void printMaze(int maze[N][N]){
    int i=0,j=0;
    for(i=0;i<N;i++){
        for(j=0;j<N;j++)    printf("%2d ",maze[i][j]);
        printf("\n");
    }
}
int isempty(int data[Max], int fe[]){
    if(fe[0]==fe[1]) return 1; //if(front==back) return 1;
    return 0;
}
void enqueue(int data[Max], int fe[],int x){
    if((fe[1]+1)%Max != fe[0]){        //if((back+1)%Max != front)
        fe[1]=(fe[1]+1)%Max;           //back=(back+1)%Max;
        data[fe[1]]=x;                 // data[back]=x;
    }
}
```

圖論－廣度優先搜尋法

```
int dequeue(int data[Max], int fe[]){
    if(isempty(data, fe)==0){
        fe[0]=(fe[0]+1)%Max;    //front=(front+1)%Max;
        return data[fe[0]];
    }
}

void find_path(int maze[N][N]){
    int x=4,y=4;
    while(1){
        printf("(%d,%d)\n", x, y);
        if (x==1 && y==1) break;
        if(maze[x+1][y]>maze[x][y])    x = x+1;
        else if(maze[x-1][y]>maze[x][y])    x = x-1;
        else if(maze[x][y+1]>maze[x][y])    y = y+1;
        else if(maze[x][y-1]>maze[x][y])    y = y-1;
    }
}
```

圖論－廣度優先搜尋法

```
void go_search(int maze[N][N], int data[], int fe[]){
    int r=0,x=0,y=0;
    int level =20;
    while(isempty(data, fe)==0){
        r = dequeue(data, fe);
        x = r/10;
        y = r%10;
        maze[x][y]=level--;          //printf("(%d,%d)\n", x, y);
        if(x==4 && y==4){    break;    }
        else{
            if(maze[x+1][y]==0){    enqueue(data, fe,(x+1)*10+y);    }
            if(maze[x-1][y]==0){    enqueue(data, fe,(x-1)*10+y);    }
            if(maze[x][y+1]==0){    enqueue(data,fe, (x)*10+y+1);    }
            if(maze[x][y-1]==0){    enqueue(data,fe, (x)*10+y-1);    }
        }
    }
}
```

圖論－廣度優先搜尋法

```
test02(){
    int maze[N][N]={
        {1,1,1,1,1},
        {1,0,1,0,1},
        {1,0,0,0,1},
        {1,0,0,0,1},
        {1,1,1,1,1}
    };
    int data[Max];
    int fe[2]={0,0};
    enqueue(data, fe, 11);
    go_search(maze, data, fe);
    printMaze(maze);
}
```

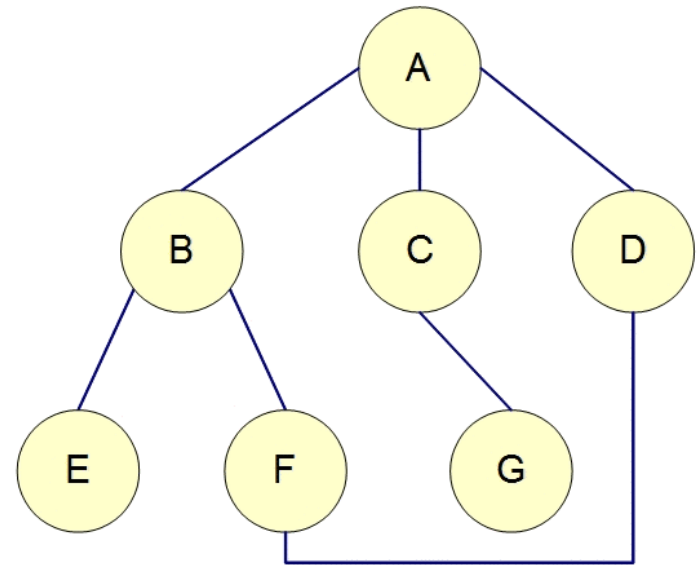
```
test01(){
    int maze[N][N]={
        {1,1,1,1,1,1},
        {1,0,1,0,0,1},
        {1,0,1,0,1,1},
        {1,0,0,0,1,1},
        {1,0,1,0,0,1},
        {1,1,1,1,1,1}
    };
    int data[Max];
    int fe[2] = {0, 0};
    enqueue(data, fe, 11);
    go_search(maze, data, fe);
    printMaze(maze);
    find_path(maze);
}

int main(){
    test01();
}
```


圖論 – 深度優先搜尋法

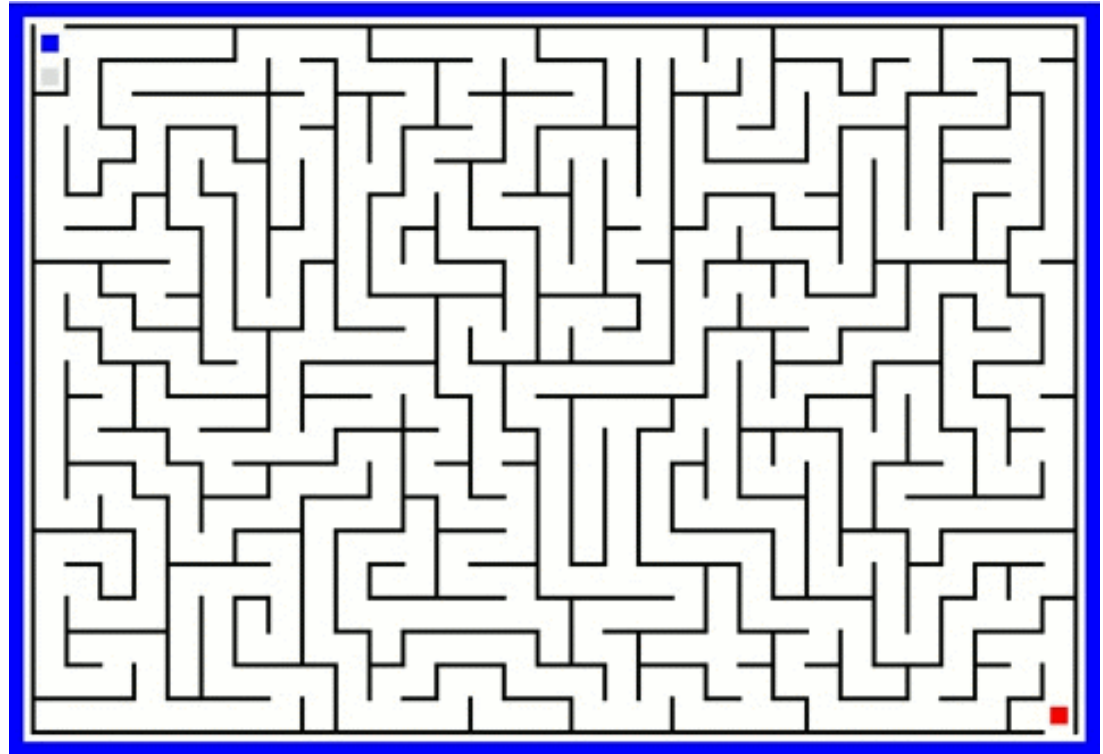
□ 深度優先搜尋法 (Depth-first Search)

- 從圖某節點開始走訪，先探尋邊(edge)上未搜尋的一節點，並儘可能深的搜索，直到該節點的所有邊上節點都已探尋；
- 回溯(backtracking)到前一節點，重覆探尋未搜尋節點，直到找到目的節點或遍尋全部節點。
- 屬盲目搜索，利用堆疊(Stack)處理



圖論 – 深度優先搜尋法

```
procedure dfs(vertex v) {  
  mark v as visited  
  for each w adjacent to v {  
    if w unvisited {  
      dfs(w)  
    }  
  }  
}
```



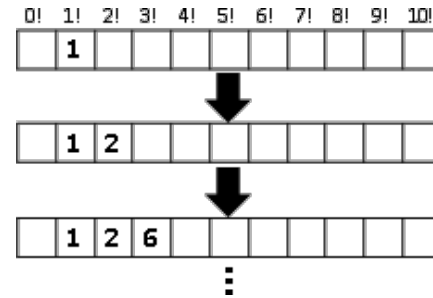
動態程式規劃

□ 動態程式規劃(Dynamic Programming)

○ 階乘 (Factorial)

```
void factorial(int f[10], int N) {  
    int i=0;  
    f[0] = 0;  f[1] = 1;  
    for (i=2; i<=N; ++i) {  
        f[i] = f[i-1] * i;  
    }  
}
```

```
void factorial(int N) {  
    int i=0, f=1;  
    for (i=2; i<=N; ++i) {  
        f = f * i;  
    }  
}
```



動態程式規劃

□ 動態規劃是分治法的延伸。

- 當遞迴分割出來的問題，一而再、再而三出現，就運用記憶法儲存這些問題的答案，避免重複求解，以空間換取時間。
- 規劃的過程是反覆讀取資料、計算、儲存資料。
- 時間複雜度 $O(N)$
- 空間複雜度 $O(N)$

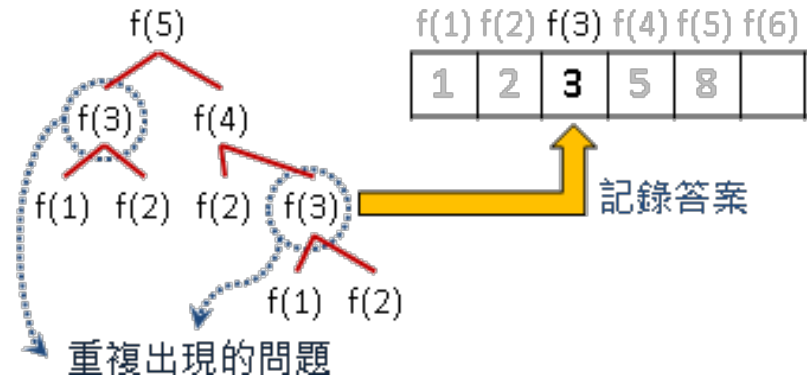
```
int f(int n) {  
    if (n == 0 || n == 1)    return 1;  
    else    return f(n-1) + f(n-2);  
}
```

Recurrence

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

Divide and Conquer

Memoization



動態程式規劃

- 動態規劃是分治法的延伸。

```
int f(int n, int s[]) {  
    if (n == 0 || n == 1) return 1;  
    // 用 0 代表該問題還未計算答案  
    if (s[n]) return s[n];  
    return s[n] = f(n-1, s) + f(n-2, s);  
}  
void stairs_climbing(){  
    int stairs[20];  
    for (int i=0; i<20; i++) {  
        stairs[i] = 0;  
    }  
    printf("%d", f(15, stairs));  
}
```

貪婪演算法

- ❑ 貪婪演算法 (Greedy Algorithm) - 換零錢遊戲
 - 有 71 個 1 元，幣值分別為 29 元、22 元、5 元、1 元，請用最少零錢個數兌換零錢。
 - 局部解：29 元 2 個，22 元 0 個，5 元 2 個，1 元 3 個， $2+2+3=7$
- ❑ 動態程式規劃
 - 最佳解：29 元 0 個，22 元 3 個，5 元 1 個，1 元 0 個， $3+1=4$
- ❑ 貪婪演算法不一定是最佳解，但效率高
 - 一種短視/近利/貪婪的想法，每一步都不管大局，只求局部解決
 - 透過一步步的選擇局部最佳解來得到問題解答。
 - 每個選擇是根據某種準則決定，前次決定不會影響後次決定。
- ❑ 動態規劃演算法可以求出最佳解，但效率略差

動態程式規劃

- 71元，最佳解：29元0個，22元3個，5元1個，1元0個，共4
 - $f(n) = \min(1+f(n-29), 1+f(n-22), 1+f(n-5), 1+f(n-1))$
 - 71元以29元兌換，剩 $71-29=42$ ，總兌換數=42元可兌換個數+1
 - $f(0)=0, f(n) = 1 + \min(f(n-c_1), f(n-c_2), \dots, f(n-c_k))$
 - $n > c_i, 1 < i < k$, c_i 是硬幣面額， k 是總共有幾種面額
 - $c_1=29, c_2=22, c_3=5, c_4=1$
 - $f(1)=1, f(2)=1+f(1)=2, f(3)=1+\min(f(2))=3, f(4)=1+\min(f(3))=4$
 - $f(5) = 1+\min(f(5-5), f(5-1)) = 1+\min(f(0), f(4)) = 1$
 - 用一個1元換；或用一個5元換；之後可以如何換最少。
 - $f(6) = 1+\min(f(6-5), f(6-1)) = 1+\min(f(1), f(5)) = 1+\min(1, 1) = 2$
 - $f(7) = 1+\min(f(7-5), f(7-1)) = 1+\min(f(2), f(6)) = 1+\min(2, 2) = 3$
 - $f(8) = \dots$
 - 要宣告 `int f[n]` 空間，換取計算時間，並計算各種可能性。

動態程式規劃

```
#include <stdio.h>
int f(int n, int coinType[], int k) {
    int p=0, i=0, coin=0, min_coin=0;
    int min_number[100]={0}, min_first_element[100];
    for (p=0; p<100; p++) min_number[p]=0;
    for (p=1; p<=n; p++){
        min_coin = n;
        for(i=0; i<k; i++){
            coin = coinType[i];
            if (((p-coin)>=0)&&((1+min_number[p-coin])<min_coin))
                min_coin = 1 + min_number[p-coin];
        }
        min_number[p] = min_coin;
        min_first_element[p] = coin;
    }
    for (p=1; p<=n; p++) {
        printf("(%d, %d)\n", p, min_number[p]);
    }
}
```

```
int main(){
    int coinType[10]={29, 22, 5, 1};
    int k=4;
    //int coinType[10]={5, 4, 2, 1};
    int n =71; //8
    f(n, coinType, k);
    return 0;
}
```


動態程式規劃-TSP

□ TSP問題 (Travelling Salesman Problem)

○ 某人拜訪 n 個城市，限制每個城市拜訪一次，最後回到原來出發的城市。目標要求最小值。

○ 將有向圖轉為鄰接矩陣

➤ V 表示所有頂點 $\{v_0 \sim v_{n-1}\}$ 集合，例如 $\{v_0, v_1, v_2\}$ 。

➤ S 表示 V 的子集，例如 (v_1, v_2) 。

➤ $d[i][j]$ 表示從 v_i 到 v_j 的直接距離，例如 $d[0][1]$ 。

➤ $R[v_0][S]$ ，從頂點 v_0 經 S 所有頂點回到 v_0 ，且僅經一次，最短距離。

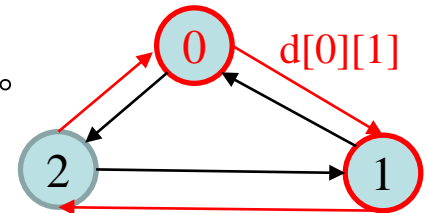
— 假設 v_1 是 v_0 的下個經過頂點，則最短距離，則

— $R[v_0][S] = d[0][1] + R[v_1][S - v_1]$ ；其中 $S = \{1, 2\}$ ， $S - v_1 = \{2\}$

— $R[v_0][\{1, 2\}] = d[0][1] + R[v_1][\{2\}]$

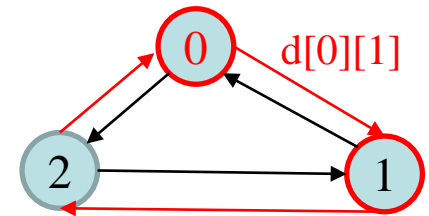
— $R[v_1][S - v_1]$ ，從頂點 v_1 經 $(S - v_1)$ 所有頂點回到 v_0 ，且僅經一次，最短距離，

— $R[v_1][\{2\}] = d[1][2] + R[v_2][\{\}]$



動態程式規劃-TSP

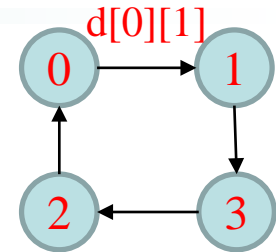
- TSP是以下二種方案，選最小值(最優結果)
 - 0出發，到1，再從1出發，經過{2}城市，回到0花費最少。
 - 0出發，到2，再從2出發，經過{1}城市，回到0花費最少。
- 最優結果，記錄在R表內，避免重複計算
 - 設計一個二維的動態規劃表R, $R[0][1, 2]$ 表示從0出發，經過1, 2 回到0花費最少。
 - 上面二方案最小值
 - $R[0][\{1, 2\}] = \min\{d[0][1] + R[1][2], d[0][2] + R[2][1]\}$
 - $R[1][2] = \min\{d[1][2] + R[2][\{\}]\} = d[1][2] + d[2][0]$
 - $R[2][1] = \min\{d[2][1] + R[1][\{\}]\} = d[2][1] + d[1][0]$
 - $R[2][\{\}]$ ，從2沒有經過任何點到0， $=d[2][0]$
 - $R[1][\{\}]$ ，從1沒有經過任何點到0， $=d[1][0]$



動態程式規劃-TSP

○ 四個點，三種路徑

- $R[0][\{1,2,3\}] = \min\{ d[0][1] + R[1][\{2,3\}], d[0][2] + R[2][\{1,3\}], d[0][3] + R[3][\{1,2\}] \}$
- $R[1][\{2,3\}] = \min\{ d[1][2] + R[2][\{3\}], d[1][3] + R[3][\{2\}] \}$
- $R[2][\{3\}] = \min\{ d[2][3] + R[3][\{\}] \} = d[2][3] + d[3][0]$
- $R[k][S] = \min\{ d[k][i] + R[i][S-i], i \in S \}$



- $R[v_0][S]$ ，從 v_0 經 S 所有頂點回到 v_0 ，且僅經一次，最短距離。
 - 等於從 v_0 到 v_j 的距離，加上從 v_j 經 S 中所有點到 v_0 的最短距離(最小值) (其中 v_j 是從 S 中取出，且是最短距離的點)
- $R[v_i][S]$ ，從 v_i 經 S 所有頂點回到 v_0 ，且僅經一次，之最短距離。
 - 等同於從 v_i 到 v_j 的距離，加上從 v_j 經 $S-v_j$ 中所有點到 v_0 的最短距離，(最小值) (其中 v_j 從 S 中取，且是最短距離的點)
 - $R[v_i][S] = \min\{ d[i][j] + R[v_j][S-v_j] \} (v_j \in S)$,
 - 初始化， $v = v_0$ ， $S = \{v_1 \sim v_{n-1}\}$

動態程式規劃-TSP

□ TSP問題 (Travelling Salesman Problem)

```
#include <stdio.h>                #include <string.h>
#define MAX_N 4                    #define INF 0x3f3f3f3f
void print(int R[][1<<MAX_N]) {
    for (int i=0; i<MAX_N; i++)
        for (int j=0; j<(1<<MAX_N); j++) printf("%d ", R[i][j]);
    printf("\n");
}
int minValue(int x, int y) { return (x>y?y:x); }
int Rec(int R[][1<<MAX_N], int d[][MAX_N], int v, int S, int n){
    int t=0, ans = INF;
    if(R[v][S] >= 0) { return R[v][S]; } // 已經計算過有紀錄，不用再計算
    for(int i = 0 ; i < n ; i ++ ) {
        if(!(S>>i&1)) {                // 集合 S，{0, 1, 2, 3}，判斷第i個存在(0為存在)
            t = Rec(R, d, i, S|(1<<i),n); // 集合 S，去掉存在的那一個，第i個設為1
            ans = minValue(ans,d[v][i]+ t);
        }
    }
    return R[v][S] = ans;
}
```

$S|(1<<i)$ ，S或(1左移i格)
 $x0xxx | (1<<3) = x0xxx | 01000$
 $= x1xxx$
將第3個設為1(1不存在)

$!(S>>i&1)$ ，S右移i個位置，跟1 and，再not
 $x1xxx >> 3 == 00x1$ ，and $0001 == 1$ ； $!1 == 0$
 $x0xxx >> 3 == 00x0$ ，and $0001 == 0$ ； $!0 == 1$
S的右邊算來第i個是0，0代表存在

動態程式規劃-TSP

□ TSP問題 (Travelling Salesman Problem)

```
void solve(){
    int v = 0, S=(0|1<<v);
    int R[MAX_N][1<<MAX_N];
    int d[MAX_N][MAX_N] = {
        {0, 3, 7, 8},
        {3, 0, 2, 6},
        {7, 2, 0, 4},
        {8, 6, 4, 0}};
    memset(R,-1,sizeof(R));
    for (int i=0; i<MAX_N; i++)
        R[i][(1<<MAX_N)-1]=d[i][0];
    printf("%d\n", Rec(R, d, v, S, MAX_N));
}
int main() {    solve();    return 0; }
```

// {0, 1, 2, 3} 編碼 0000
// 從v=0出發，經過{1, 2, 3}->{0001}
// 從v=1出發，經過{0, 2, 3}->{0010}
// R[1][空集合 1111]=d[1][0]
// R[0][空集合 1111]=d[0][0]=0

動態程式規劃-TSP

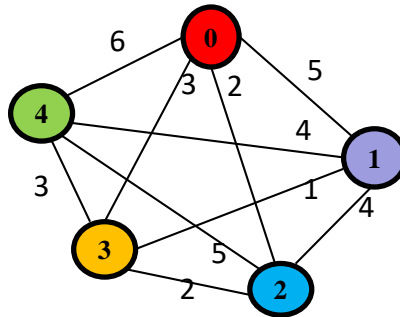
□ TSP問題 (Travelling Salesman Problem)

```
void travel(int n,const number W[][],index P[][],number& minlenth) {  
    index i,j,k;  
    number D[n][V-{v1}];  
    for(i=2;i<=n;i++)  
        D[i][空集]=W[i][1]; //從頂點vi到v1的直接距離  
    for(k=1;k<=n-2;k++) //遍歷A的每個子集  
        for(包含k個頂點的所有子集A屬於V-{vi}) //從小到大遍歷  
            for(滿足i != 1 且 vi不在v中的i) //算出任意點出發到v1結束的最短距離 {  
                D[i][A]=min{ W[i][j]+D[j][A-{vj}]} ; //vj∈A  
                P[i][A]=最小值的j值;  
            }  
    D[1][V-{vi}]=min{ W[i][j]+D[j][A-{v1 , vj}]} ;  
    //2<=j<=n 從v1出發到v1結束的最短距離  
    P[1][V-{vi}]=最小值的j值 ;  
    minlengt=D[1][V-{vi}];  
}
```

Homework-銷售員旅行城市

- 小英以UBIKE逛 N 個景點($N \leq 10$)。景點編號 0, 1, 2, 3, 4, 5, 6, 7, 8, 9。每兩個景點都有一段距離。請計算從第0 個景點出發，騎過每一個景點的最短距離。

X	0	1	2	3	4
0	0	5	2	3	6
1	5	0	4	1	4
2	2	4	0	2	5
3	3	1	2	0	3
4	6	4	5	3	0



- 例如 $N = 5$ ，以下是兩兩景點間的距離。則0-2-3-1-4-0的距離是 $15 = 2 + 2 + 1 + 4 + 6$ ，最短。

銷售員旅行城市

□ 節點編號 $\{0, 1, 2, 3, 4, \dots, N-1\}$

○ $S = \{S_{n-1}S_{n-2} \dots S_i \dots S_1S_0\}$ ，是節點存在狀態集合的編碼，

➤ S_i 存在以0表示， S_i 不存在以1表示

➤ 若存在節點 $\{1, 3, 5\}$ ，不存在 $\{0, 2, 4\}$ ，

– 編碼為 $\{010101\}$ ， $S = 2^4 + 2^2 = 16 + 4 = 20$

➤ 針對 S ，檢查節點 i 是否存在 $S_i \in S$

– 檢查 S (節點存在狀態集合的編碼)， $\{S_{n-1}S_{n-2} \dots S_i \dots S_1S_0\}$

– S 編碼中的 S_i 為0，則節點 i 存在

– $S \& (1 \ll i) == 0$ ，則節點 i 存在

銷售員旅行城市

□ 從k點出發，經過存在節點 $\{n-1, n-2, \dots, i, 1, 0\}$ ，節點存在狀態集合的編碼 $S = \{S_{n-1}S_{n-2} \dots S_i \dots S_1S_0\}$ ，

○ 最短距離定義為 $R[k][S]$

➤ 針對所有的存在節點 $S_i \in S$ ，計算經過之最小距離

$$R[k][S] = \min\{(d[k][i] + R[i][S - S_i]) \mid S_i \in S\}$$

– 從k到i ($d[k][i]$)距離，加上從i經過 $S - S_i$ 距離

➤ 針對S，把節點i從S中移除

– $S - S_i = \{S_{n-1}S_{n-2} \dots S_i \dots S_1S_0\} - \{S_i\}$ ，把設成1

– 若 $S = \{1, 3, 5\} = \{010101\}$ ， $i=3$ ， $S - S_i = \{011101\}$

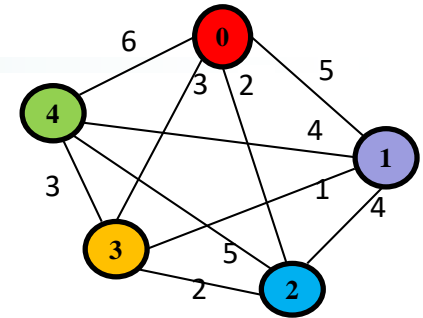
– $S - S_i = S|(1 \ll i) = 010101|(1 \ll 3) = 010101|(001000) = 011101$

銷售員旅行城市

□ 計算出所有R後，求出經過最短路徑

○ 找到下一個經過的節點(最短的那一條/那幾條)

```
findNextNodes(d[][], R[][], k, S, N, nextNodes[])  
    // ---- 找最短那一條的距離 minDistance  
    for i=0~N-1  
        if ( $S_i \in S$ ) //  $S \& (1 \leq i) == 0$   
            distance =  $d[k][i] + R[i][S - S_i]$   
            if (distance < minDistance)  
                minDistance = distance  
    // ----- 找出所有最短距離的所有點  
    for i=0~N-1  
        if ( $S_i \in S$ ) //  $S \& (1 \leq i) == 0$   
            distance =  $d[k][i] + R[i][S - S_i]$   
            if (distance == minDistance)  
                nextNodes[count++] = i  
    // 得到所有最短距離的下一個點 nextNodes[]
```



➤ 遞迴的往下找

銷售員旅行城市

□ 計算出所有R後，求出經過最短路徑

➤ 遞迴的往下找

```
findPath(R, d, k, S, N) // N為節點數，k為目前探索的節點，d距離
    S = S - Si
    if (S==(1111..11)) //結束條件，沒有存在任何節點
        印出stack中存的節點
    else //一般條件
        count = getNextNodes() //取得所有最短路徑的下一個節點
        for i = 1~count-1
            將 nextNodes[i]存入stack
            findPath(R, d, nextNodes[i], S, N)
        pop stack
```

