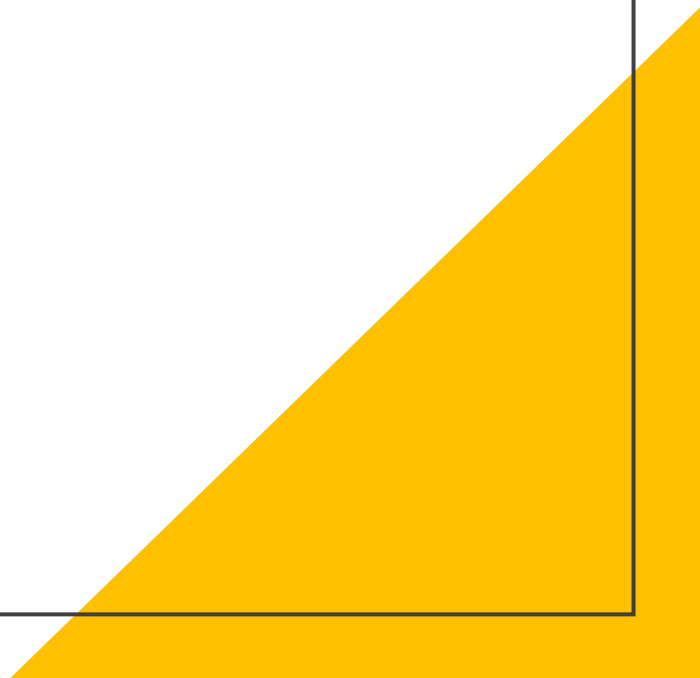
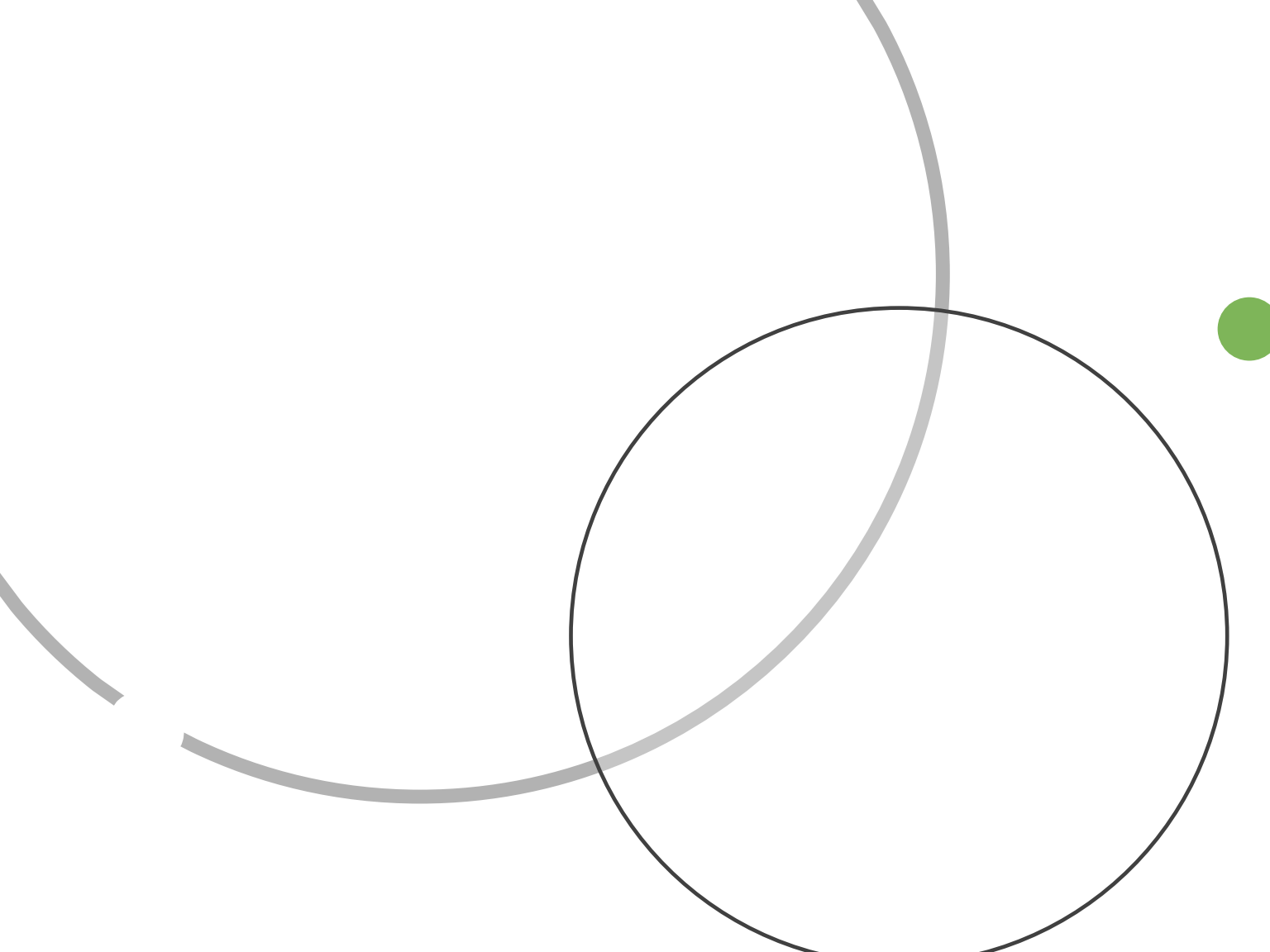


遞迴函數 與枚舉

日月卦長





複習 特殊輸入

stringstream

範例：將每行的數字加起來

Input

1 2 3 4 5

123 456

789 56461 7122

Output

15

579

64372

C++ getline (string)

- C++ 常見的整行讀取，用 `getline(cin, string)`

```
string str;  
getline(cin, str);  
cout << str << endl;
```

```
jacky860226@DESKTOP-FCBV14M:/mnt/d/users/SunMoon/Desktop$ ./a.out  
Input String  
Input String  
jacky860226@DESKTOP-FCBV14M:/mnt/d/users/SunMoon/Desktop$
```

將輸入原封不動輸出

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string str;
    while (getline(cin, str)) { // 讀到 EOF 結束
        cout << str << endl;
    }
    return 0;
}
```

stringstream

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main() {
    string str = "12 34 56";
    stringstream ss(str);
    int a = 0;
    while (ss >> a) {
        cout << a << endl;
    }
    return 0;
}
```

- #include <sstream>
- 把字串當成是輸入輸出來使用
- 個人認為是最為方便的方法

```
12
34
56
```

合併起來，就是範例解答

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main() {
    string str;
    while (getline(cin, str)) { // 讀到 EOF 結束
        stringstream ss(str);
        int sum = 0, a = 0;
        while (ss >> a) {
            sum += a;
        }
        cout << sum << endl;
    }
    return 0;
}
```

不是用空白隔開怎麼辦？

Input

1,2,3,4,5

123&456

789#56461#7122

Output

15

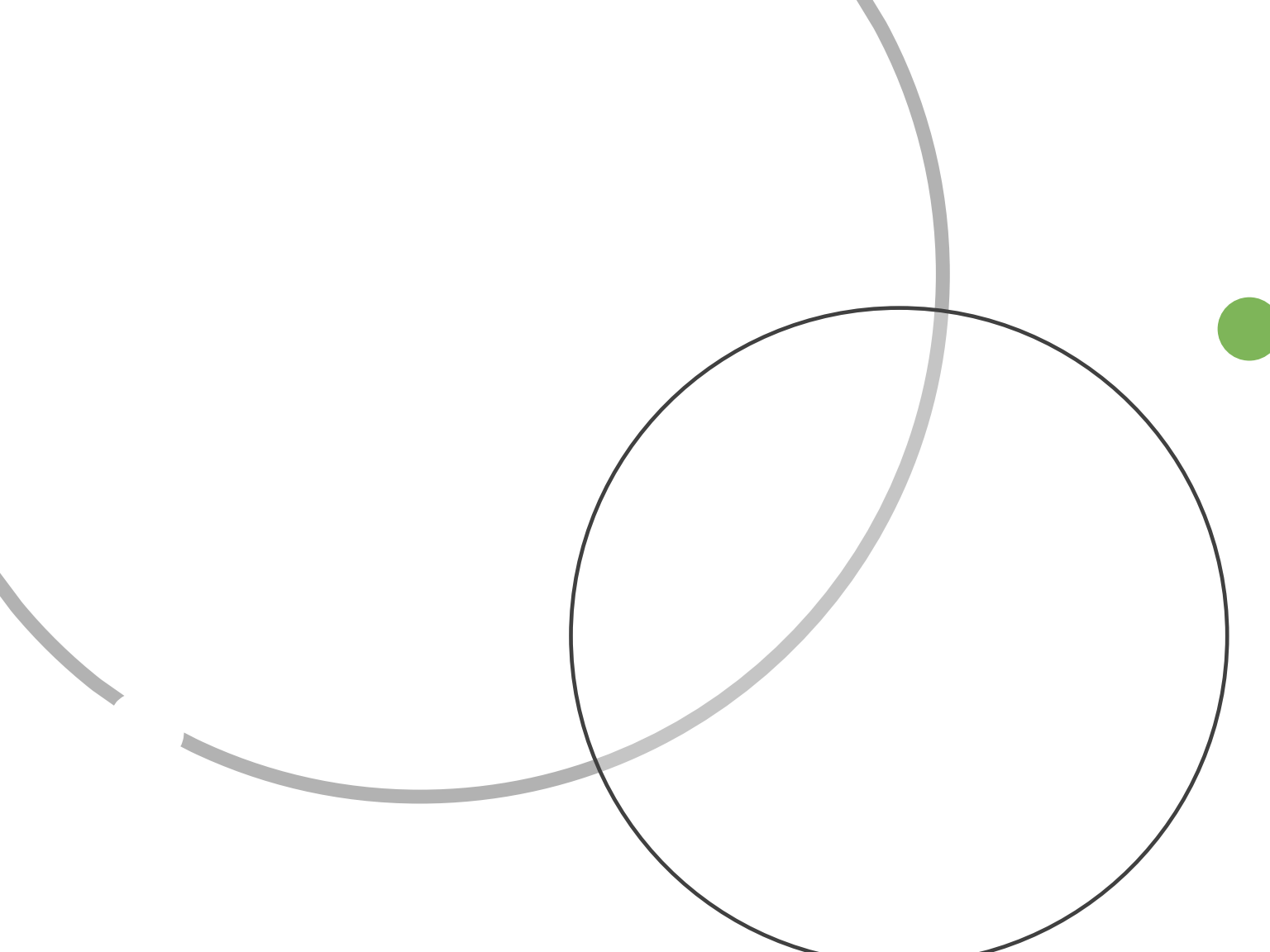
579

64372

除了數字 都換成空白

```
#include <cctype>
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main() {
    string str;
    while (getline(cin, str)) { // 讀到 EOF 結束
        for (char &c : str)
            if (!isdigit(c))
                c = ' ';
        stringstream ss(str);
        int sum = 0, a = 0;
        while (ss >> a) {
            sum += a;
        }
        cout << sum << endl;
    }
    return 0;
}
```



遞迴 函數

想法與原理

複習：函數三大要素



函數二步驟

定義函數功能

實作函數內容

定義函數功能

回傳 a, b 的最大值

我相信等我寫出函數內容後
結果是正確的
所以直接使用也沒關係

```
#include <iostream>
using namespace std;

int max(int a, int b); // 定義函數

int main() {
    cout << max(1, 2) << endl;
    return 0;
}
```

實作函數內容 – 兩種方法

比賽時通常這樣寫
因為比較短

```
#include <iostream>
using namespace std;

int max(int a, int b); // 定義函數

int main() {
    cout << max(1, 2) << endl;
    return 0;
}

int max(int a, int b) { // 實作函數
    return a > b ? a : b;
}
```

```
#include <iostream>
using namespace std;

int max(int a, int b) { // 定義 + 實作函數
    return a > b ? a : b;
}

int main() {
    cout << max(1, 2) << endl;
    return 0;
}
```

遞迴函數三步驟

定義函數功能

實作函數內容

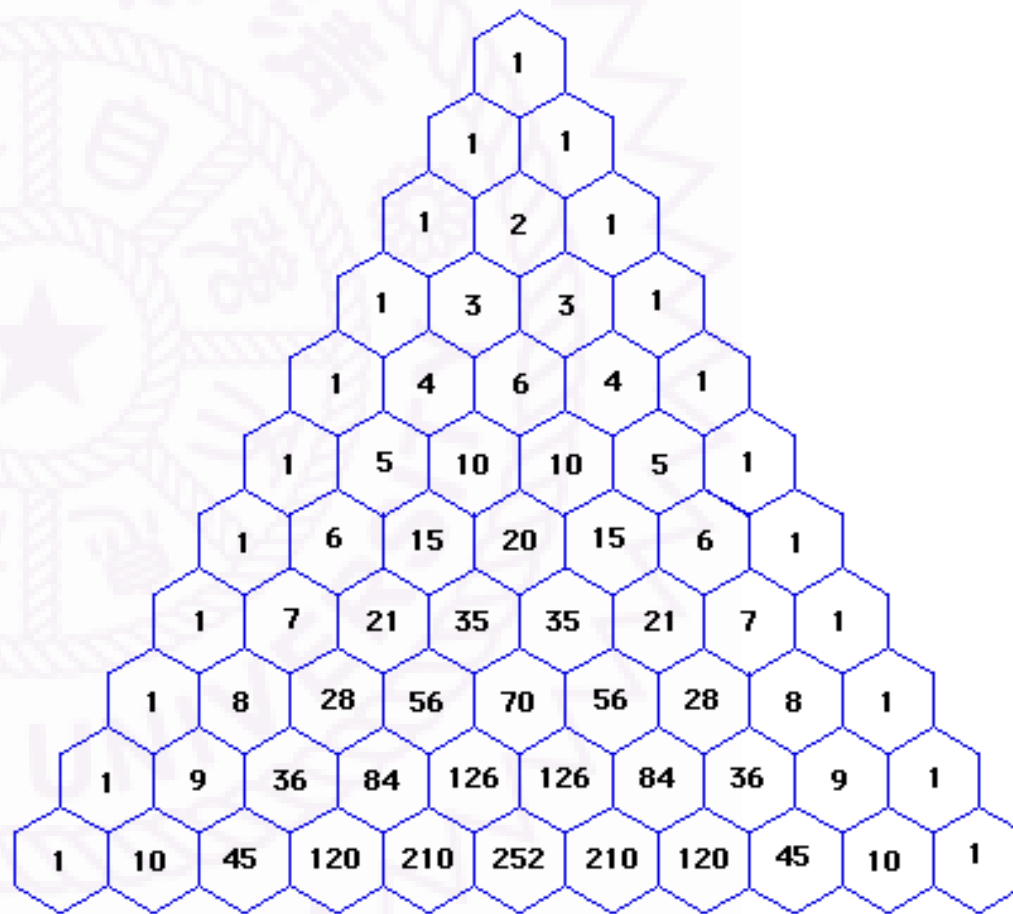
寫出邊界條件



以計算二項式係數為例

- $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

- $\binom{n}{0} = \binom{n}{n} = 1$



定義函數功能

回傳 $\binom{n}{k}$

```
#include <iostream>
using namespace std;

int C(int n, int k); // 定義函數

int main() {
    cout << C(10, 5) << endl; // 252
    return 0;
}
```

實作函數內容

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

我相信等我寫出函數內容後
結果是正確的
所以直接使用也沒關係

```
#include <iostream>
using namespace std;

int C(int n, int k) {
    return C(n - 1, k - 1) + C(n - 1, k);
}

int main() {
    cout << C(10, 5) << endl; // 252
    return 0;
}
```

寫出邊界條件

$$\binom{n}{0} = \binom{n}{n} = 1$$

如果能夠簡單計算出答案
就應該要直接計算

```
#include <iostream>
using namespace std;

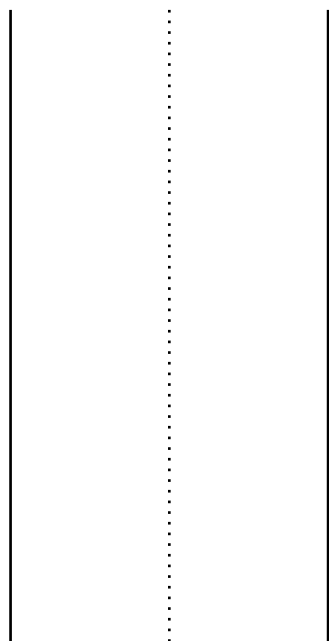
int C(int n, int k) {
    if (k == 0 || k == n)
        return 1;
    return C(n - 1, k - 1) + C(n - 1, k);
}

int main() {
    cout << C(10, 5) << endl; // 252
    return 0;
}
```

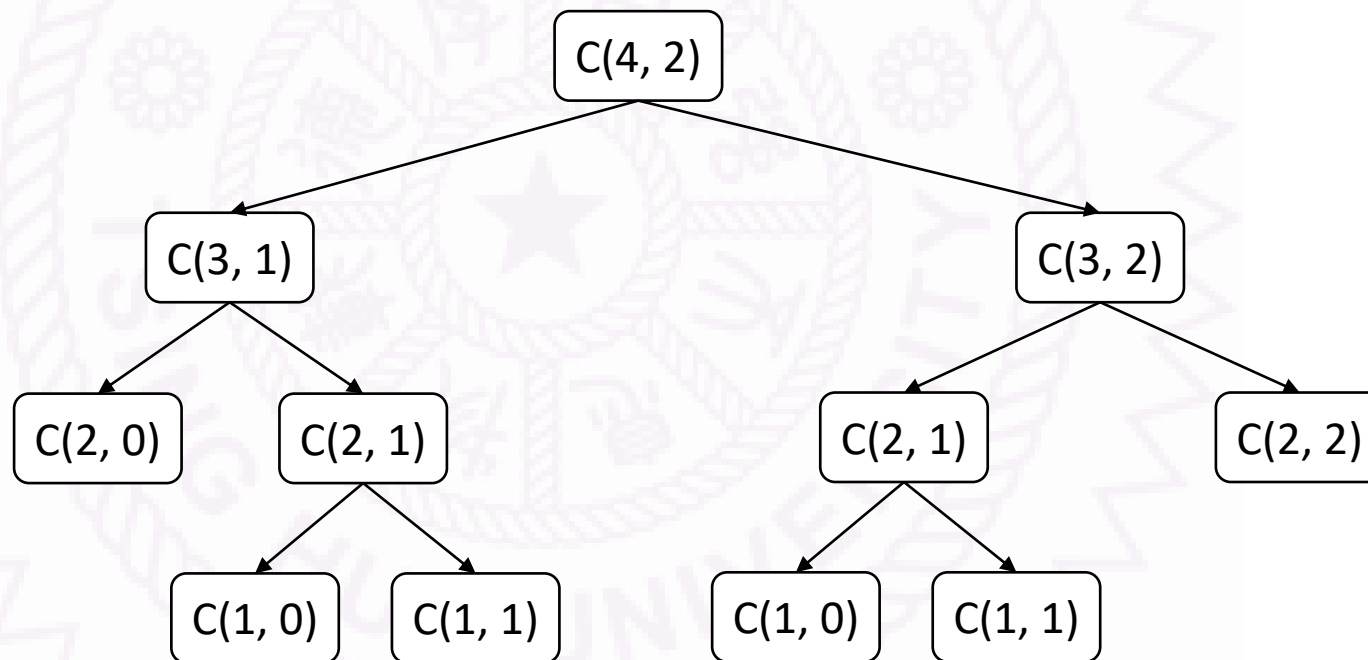
模擬執行 C(4, 2)

```
1. int C(int n, int k) {  
2.   if (k == 0 || k == n)  
3.     return 1;  
4.   return C(n - 1, k - 1) + C(n - 1, k);  
5. }
```

4-L 4-R



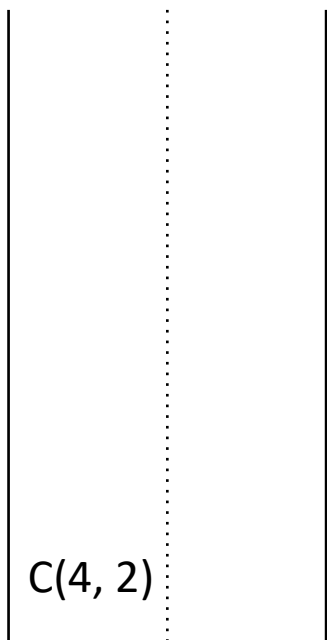
Stack



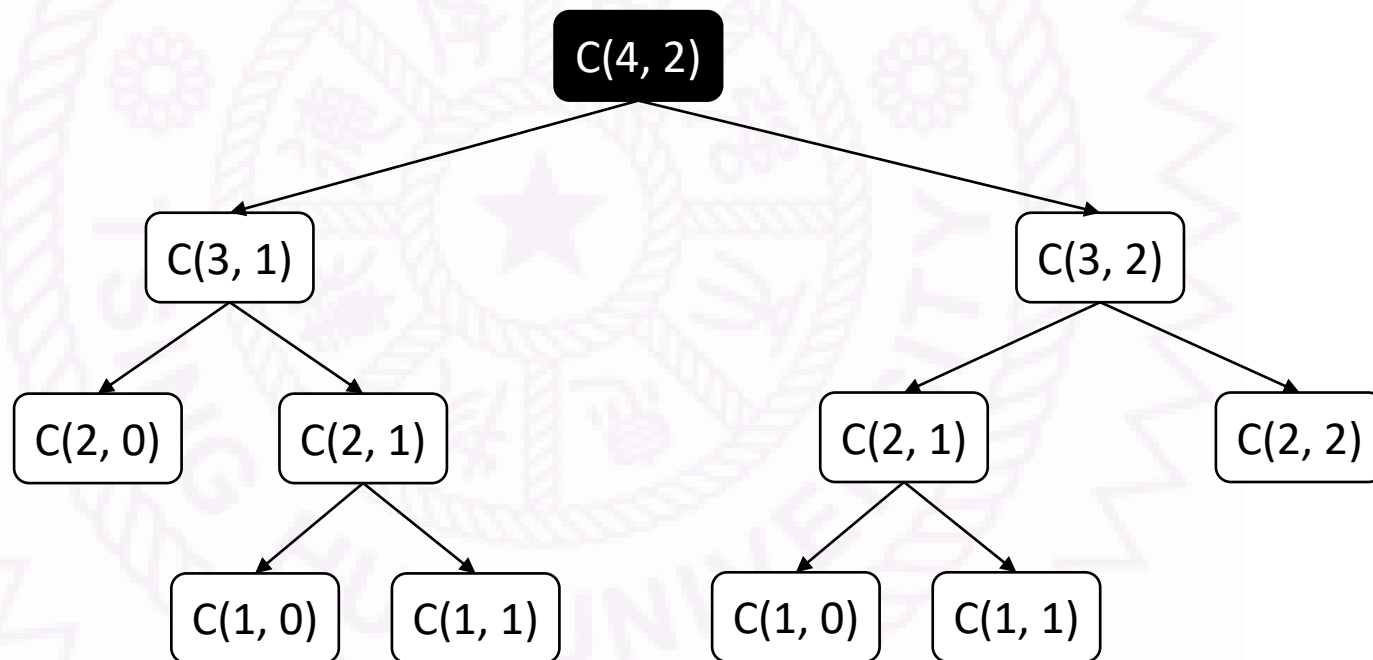
模擬執行 C(4, 2)

```
1. int C(int n, int k) {  
2.     if (k == 0 || k == n)  
3.         return 1;  
4.     return C(n - 1, k - 1) + C(n - 1, k);  
5. }
```

4-L 4-R



Stack



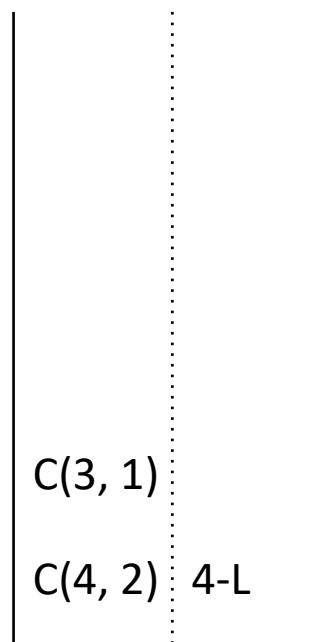
模擬執行 C(4, 2)

```
1. int C(int n, int k) {  
2.     if (k == 0 || k == n)  
3.         return 1;  
4.     return C(n - 1, k - 1) + C(n - 1, k);  
5. }
```

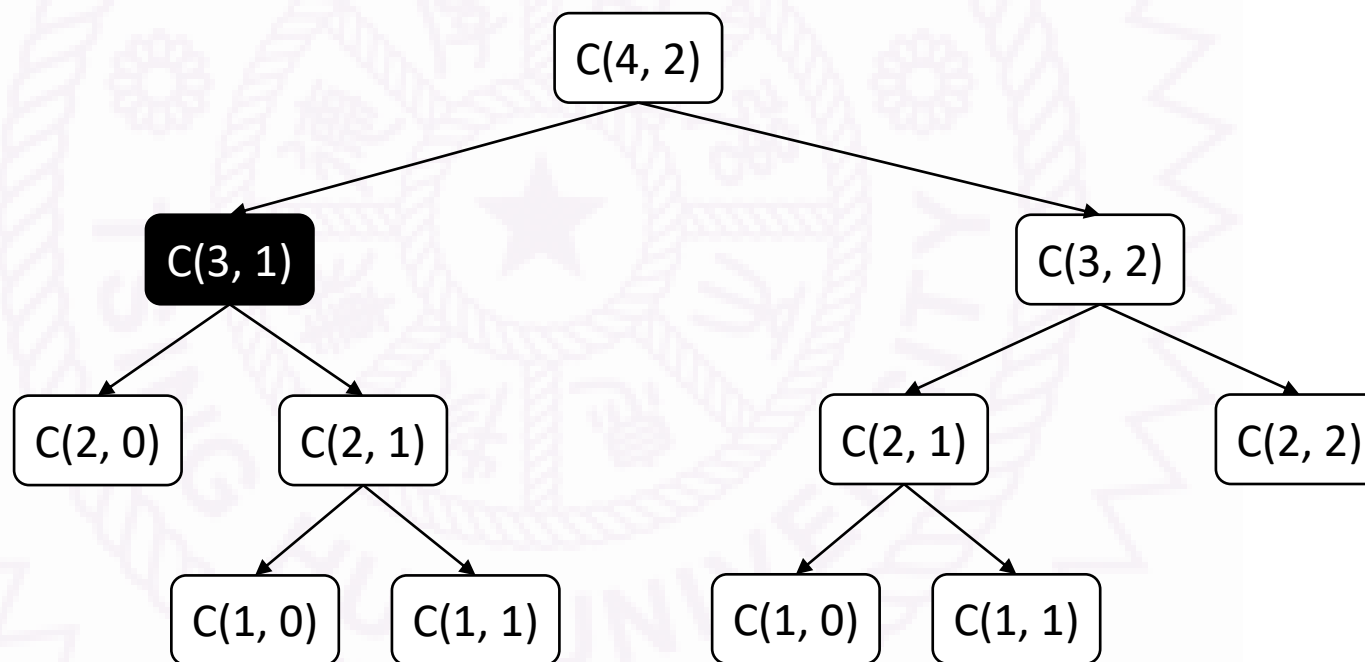
4-L 4-R

進入遞迴時

會把當前函數內所有變數以及執行到哪裡存入 **stack** 中



Stack



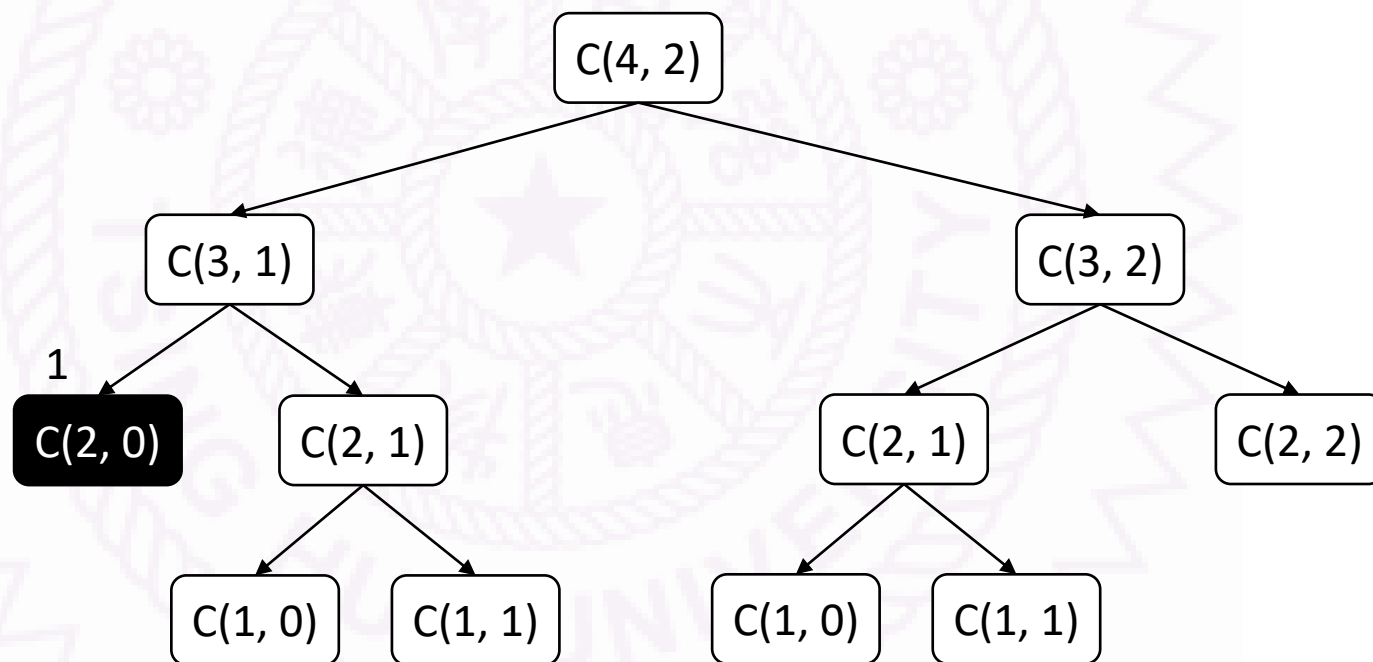
模擬執行 $C(4, 2)$

```
1. int C(int n, int k) {  
2.   if (k == 0 || k == n)  
3.     return 1;  
4.   return C(n - 1, k - 1) + C(n - 1, k);  
5. }
```

4-L 4-R

C(2, 0)	
C(3, 1)	4-L
C(4, 2)	4-L

Stack

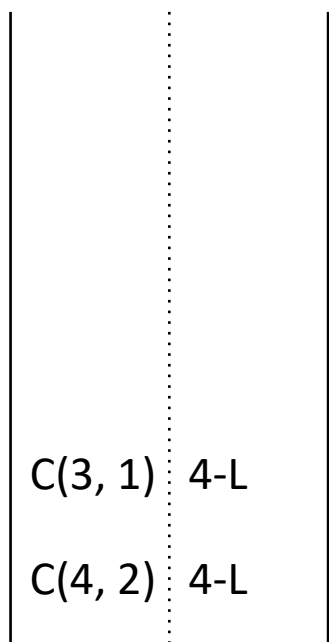


模擬執行 C(4, 2)

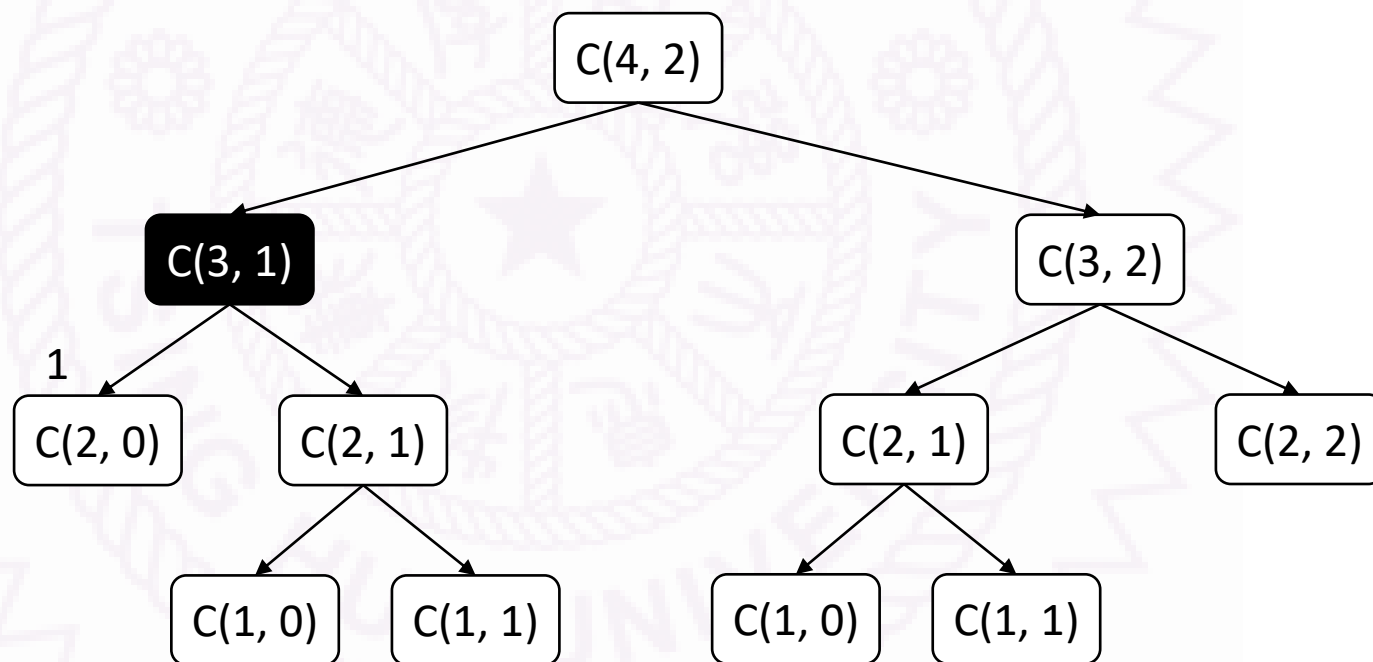
```
1. int C(int n, int k) {  
2.     if (k == 0 || k == n)  
3.         return 1;  
4.     return C(n - 1, k - 1) + C(n - 1, k);  
5. }
```

4-L 4-R

離開遞迴時
會回去上一層紀錄的地方繼續執行



Stack



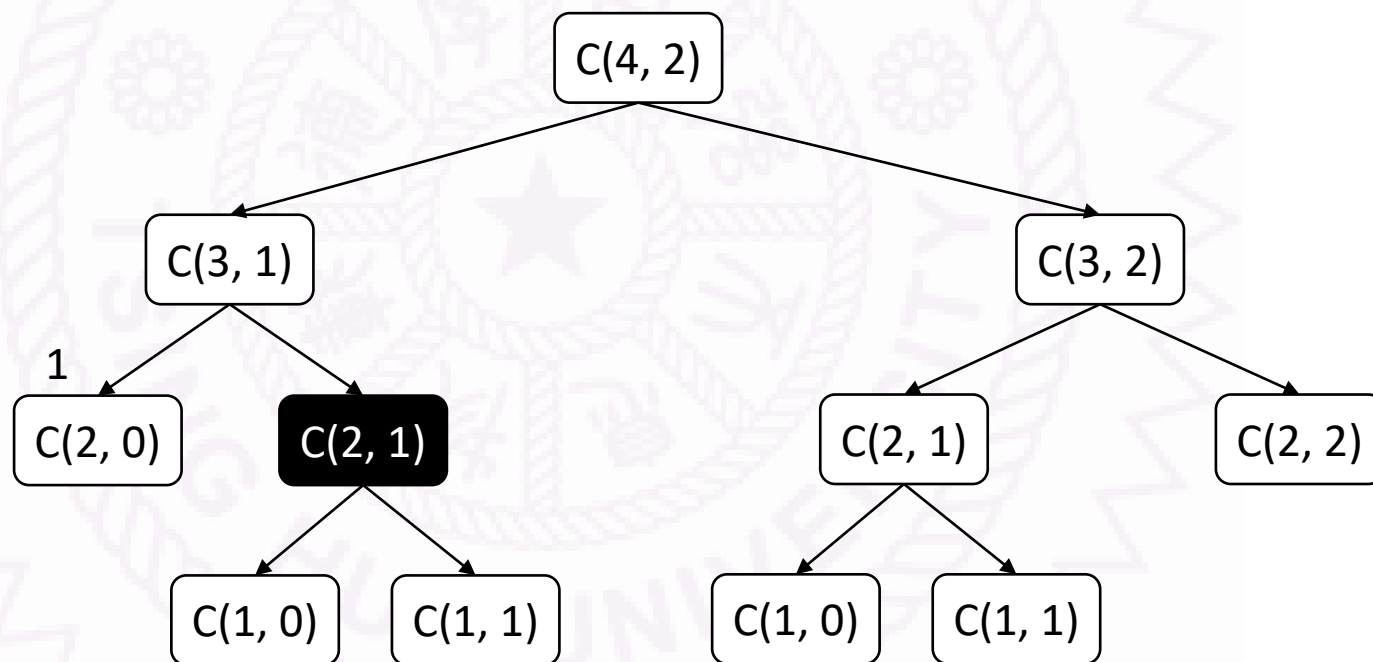
模擬執行 C(4, 2)

```
1. int C(int n, int k) {  
2.   if (k == 0 || k == n)  
3.     return 1;  
4.   return C(n - 1, k - 1) + C(n - 1, k);  
5. }
```

4-L 4-R

C(2, 1)	
C(3, 1)	4-R
C(4, 2)	4-L

Stack



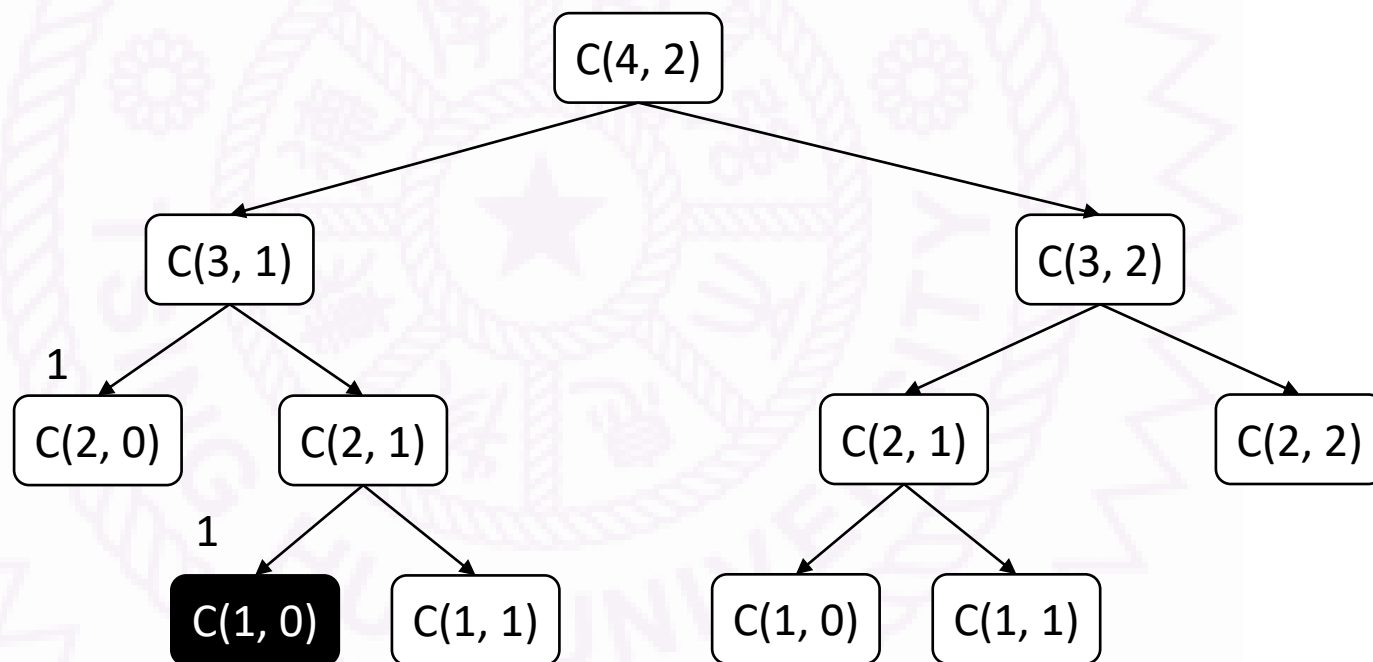
模擬執行 C(4, 2)

```
1. int C(int n, int k) {  
2.   if (k == 0 || k == n)  
3.     return 1;  
4.   return C(n - 1, k - 1) + C(n - 1, k);  
5. }
```

4-L 4-R

C(1, 0)	
C(2, 1)	4-L
C(3, 1)	4-R
C(4, 2)	4-L

Stack



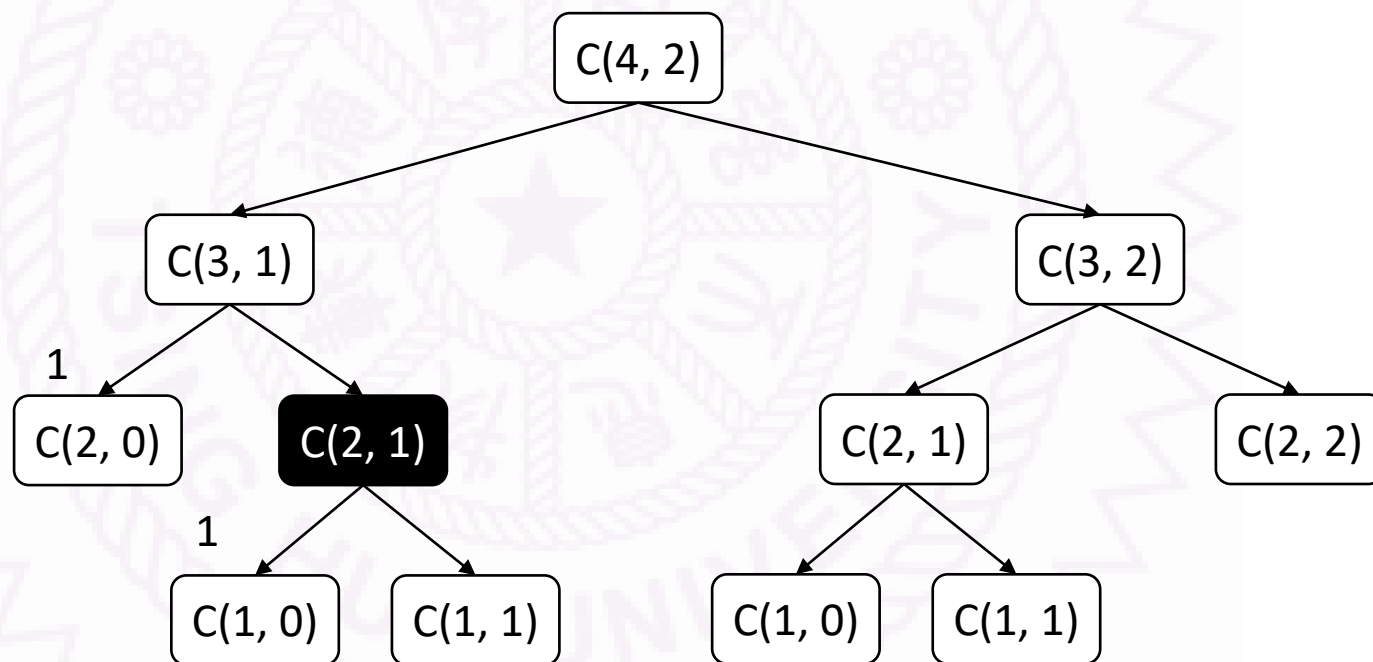
模擬執行 C(4, 2)

```
1. int C(int n, int k) {  
2.   if (k == 0 || k == n)  
3.     return 1;  
4.   return C(n - 1, k - 1) + C(n - 1, k);  
5. }
```

4-L 4-R

C(2, 1)	4-L
C(3, 1)	4-R
C(4, 2)	4-L

Stack



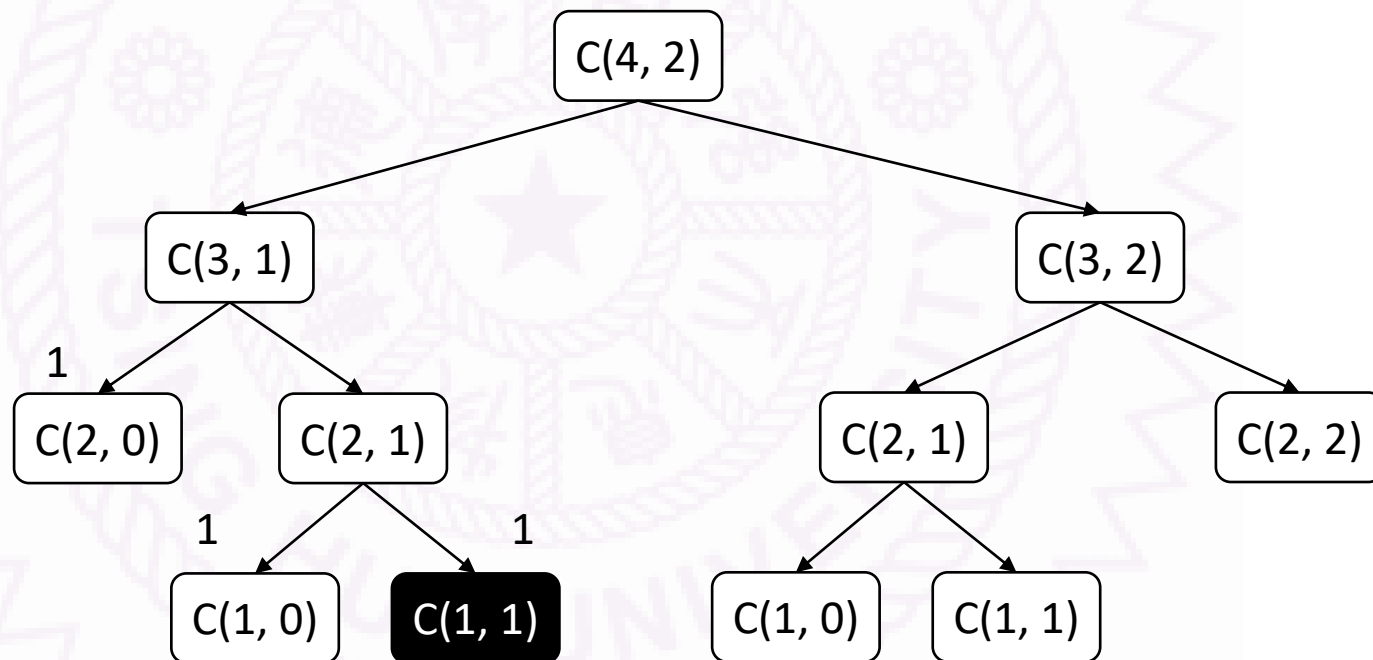
模擬執行 C(4, 2)

```
1. int C(int n, int k) {  
2.   if (k == 0 || k == n)  
3.     return 1;  
4.   return C(n - 1, k - 1) + C(n - 1, k);  
5. }
```

4-L 4-R

C(1, 1)	
C(2, 1)	4-R
C(3, 1)	4-R
C(4, 2)	4-L

Stack



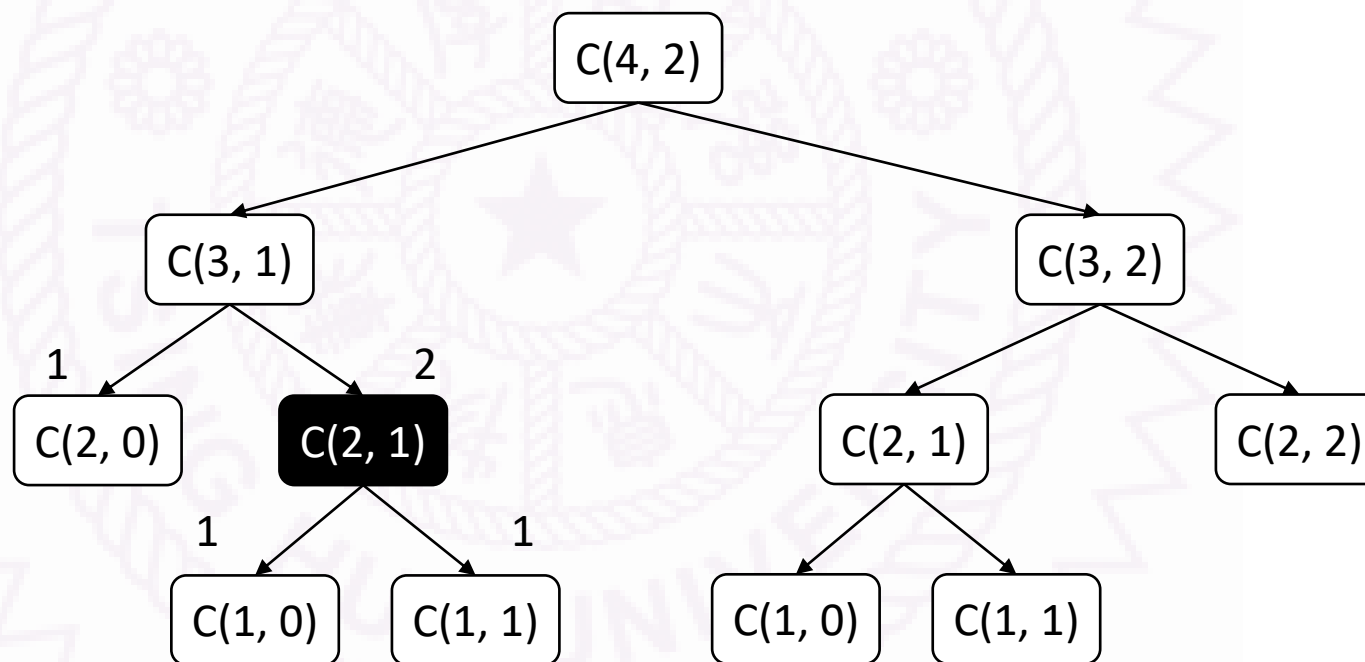
模擬執行 C(4, 2)

```
1. int C(int n, int k) {  
2.   if (k == 0 || k == n)  
3.     return 1;  
4.   return C(n - 1, k - 1) + C(n - 1, k);  
5. }
```

4-L 4-R

C(2, 1)	4-R
C(3, 1)	4-R
C(4, 2)	4-L

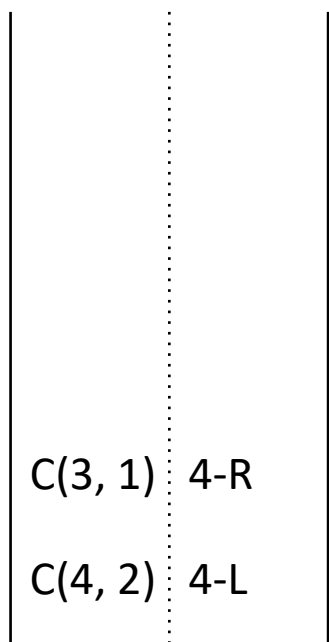
Stack



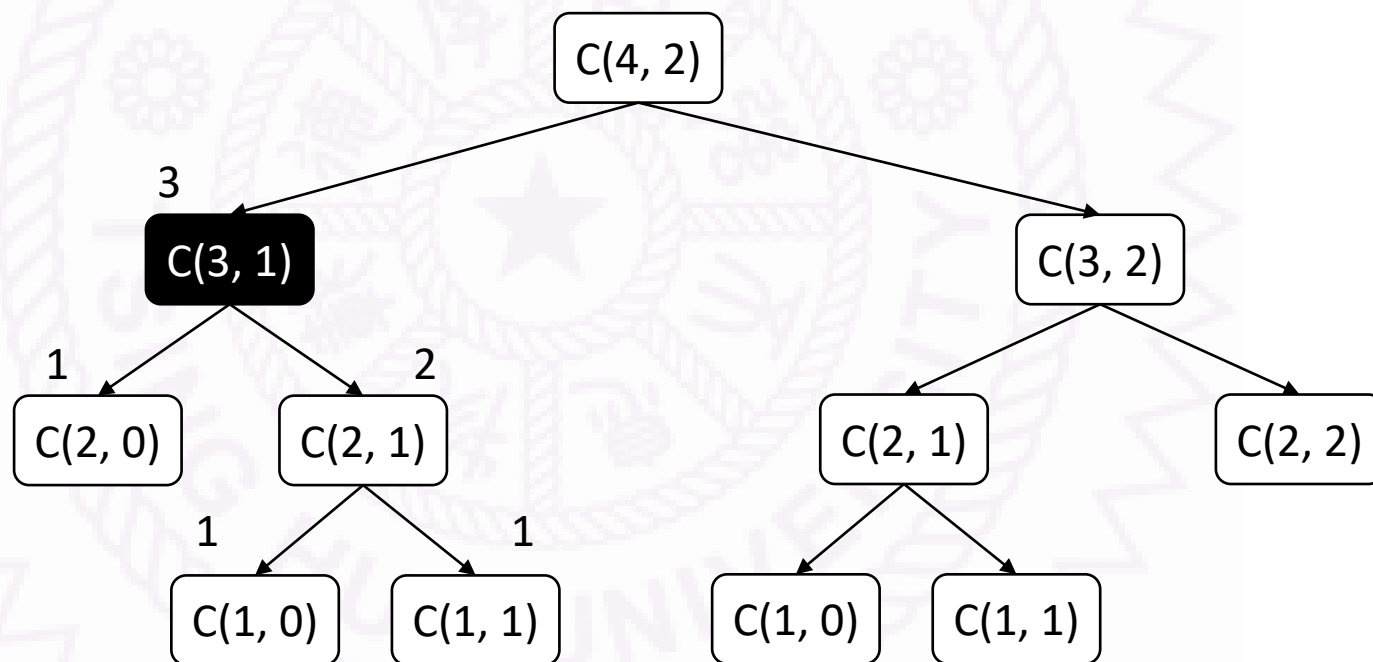
模擬執行 C(4, 2)

```
1. int C(int n, int k) {  
2.   if (k == 0 || k == n)  
3.     return 1;  
4.   return C(n - 1, k - 1) + C(n - 1, k);  
5. }
```

4-L 4-R



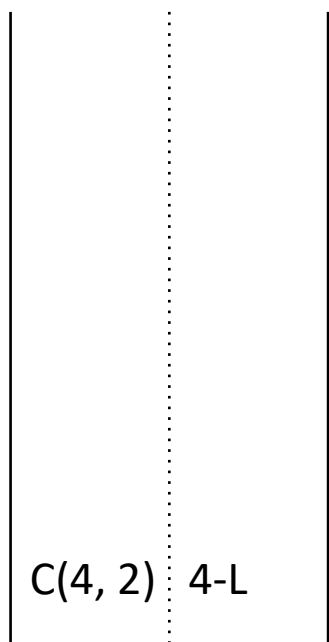
Stack



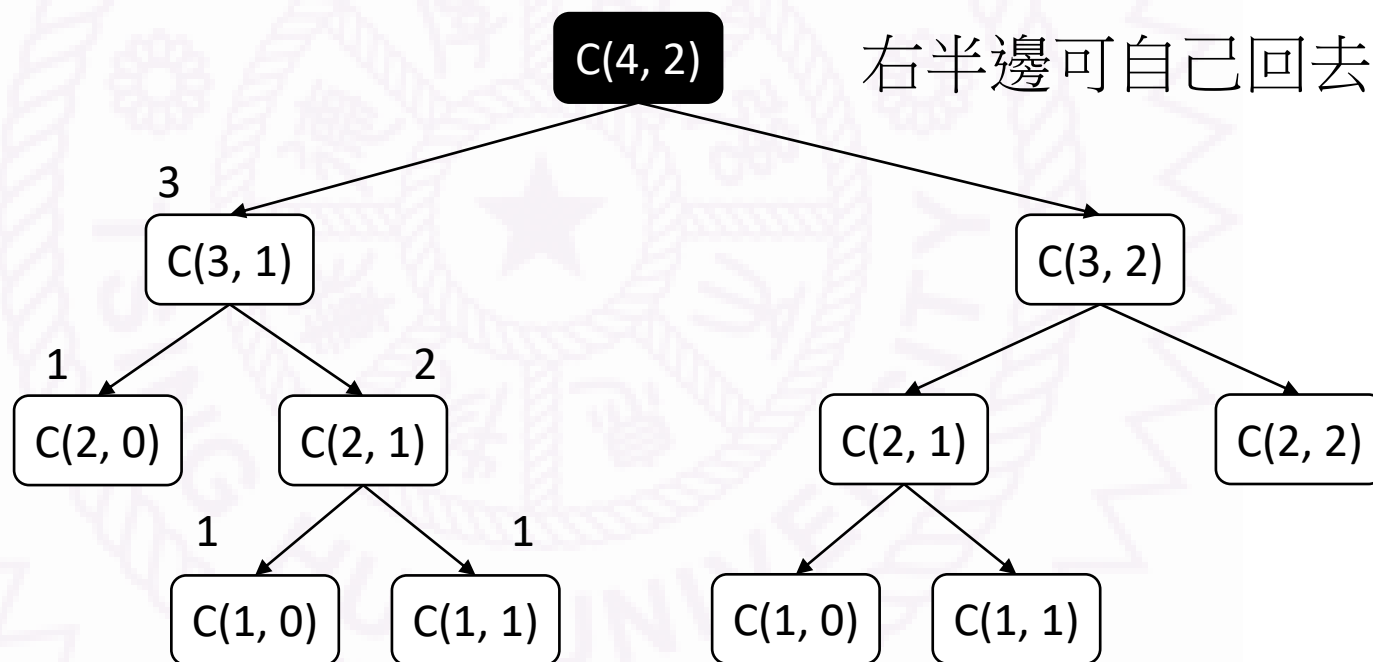
模擬執行 $C(4, 2)$

```
1. int C(int n, int k) {  
2.   if (k == 0 || k == n)  
3.     return 1;  
4.   return C(n - 1, k - 1) + C(n - 1, k);  
5. }
```

4-L 4-R



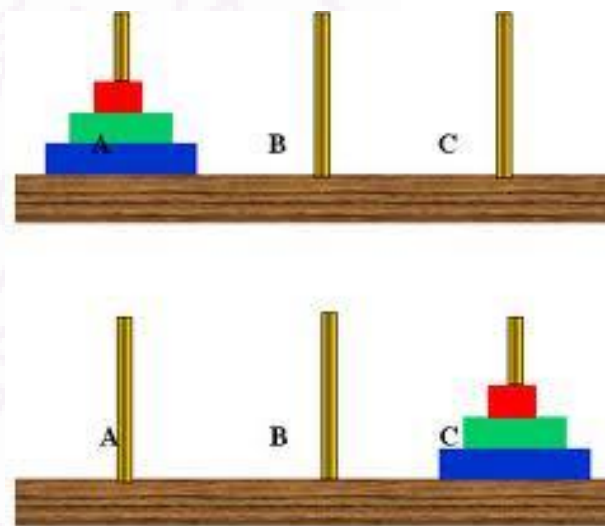
Stack



右半邊可自己回去推一下

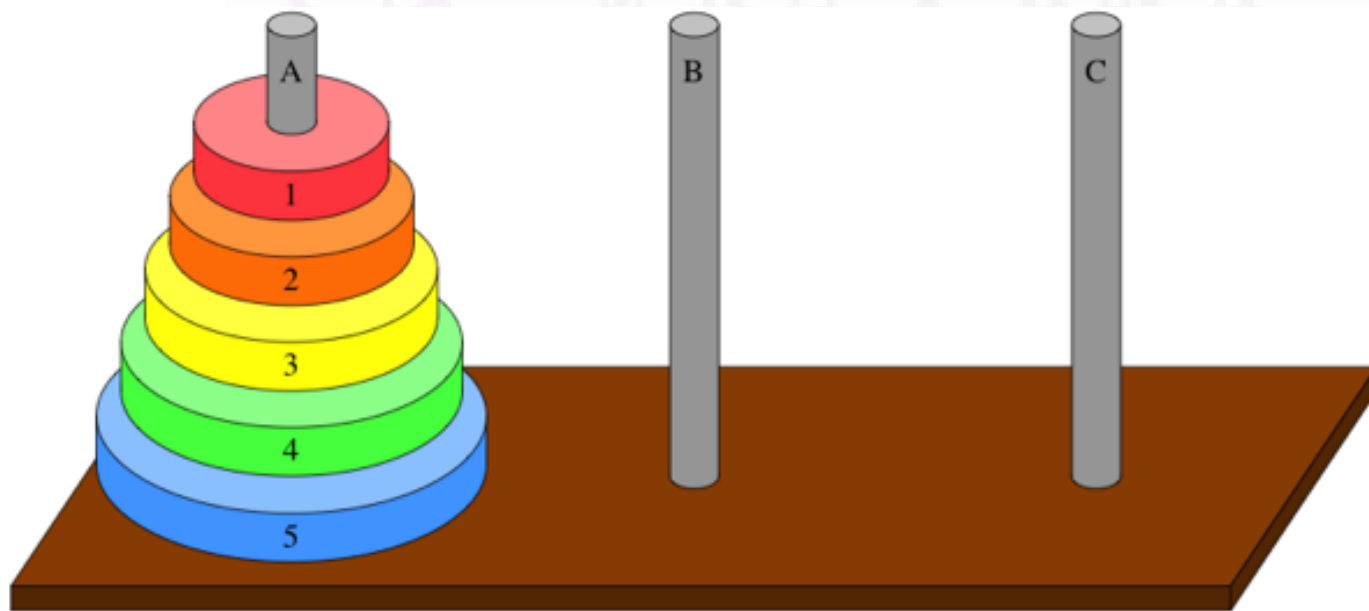
河內塔

- 有三根杆子 A, B, C
- A 杆上有 n 個 ($n > 1$) 穿孔圓盤，盤的尺寸由下到上依次變小
- 要求按下列規則將所有圓盤移至 C 杆
 - 每次只能移動一個圓盤
 - 大盤不能疊在小盤上面



河內塔

- 方便起見，盤子的編號由小到大依序為 $1 \sim n$



輸出把 n 個盤子從 A 移到 C 的方法

Input

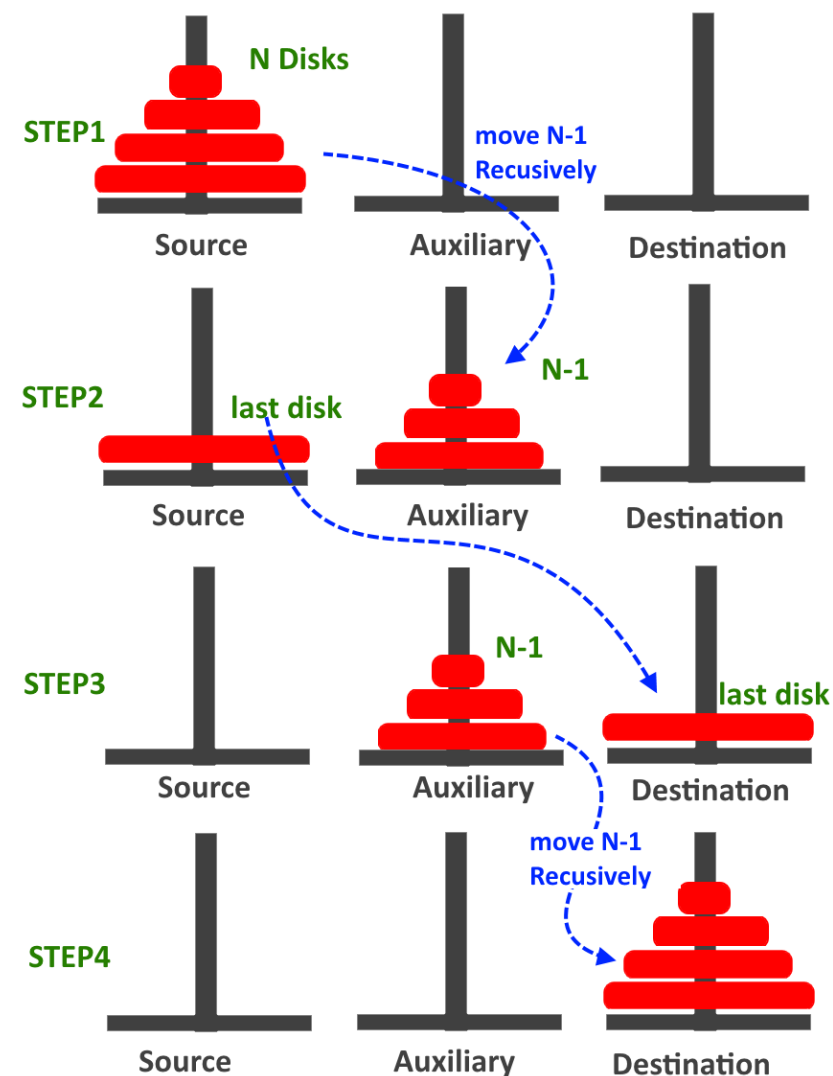
3

Output

Move ring 1 from A to C
Move ring 2 from A to B
Move ring 1 from C to B
Move ring 3 from A to C
Move ring 1 from B to A
Move ring 2 from B to C
Move ring 1 from A to C

想法拆解

- 先把 A 上面編號 $1 \sim n - 1$ 的盤子移到 B 上面
- 現在 A 上只剩下編號 n 的盤子
- 把編號 n 的盤子移到 C 上面
- 再把 B 上面編號 $1 \sim n - 1$ 的盤子移到 C 上面
- 完成



函數定義

- 定義函數 `hanoi (int n, char A, char B, char C)`
- 有n個盤子
輸出從 A 桿“透過 B ”桿移動到 C 桿的過程

函數設計

- 用剛剛定義的函數來表示我們的想法
- 先把A上面編號 $1 \sim n - 1$ 的盤子移到B上面(透過C)
→ `hanoi(n - 1, A, C, B);`
- 再把B上面編號 $1 \sim n - 1$ 的盤子移到C上面(透過A)
→ `hanoi(n - 1, B, A, C);`
- 就算你函數還沒寫完
也請相信你的函數是可以被使用的

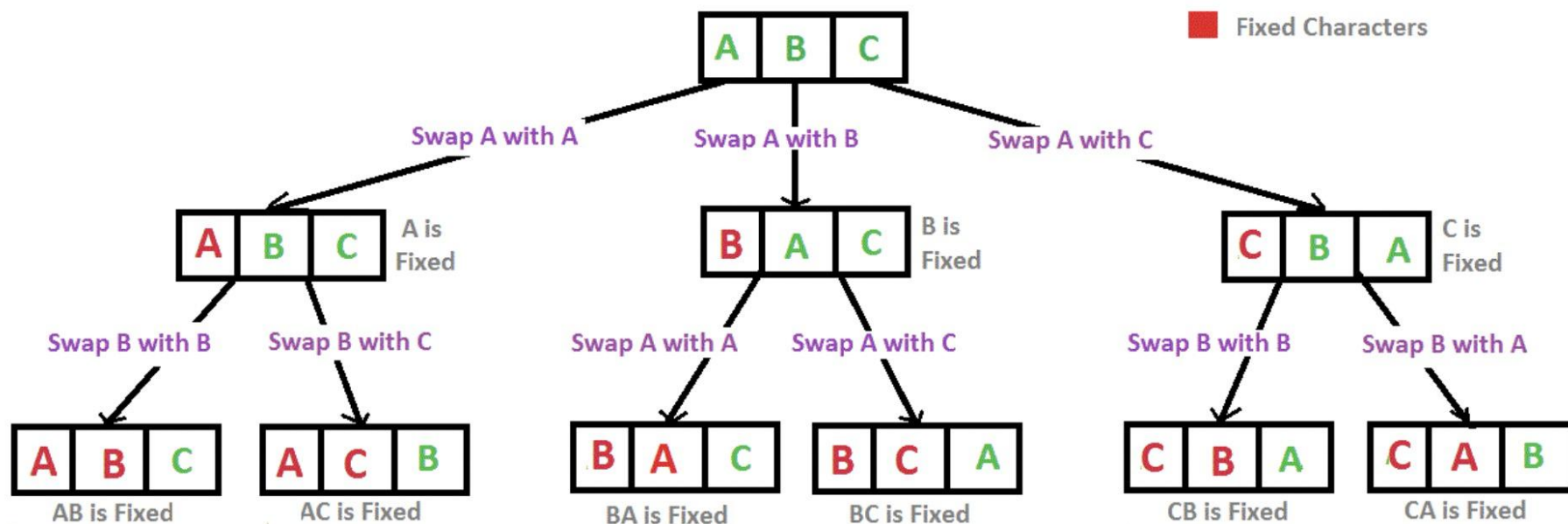
函數設計

- 先把A上面編號 $1 \sim n - 1$ 的盤子移到B
- 現在A上只剩下編號N的盤子
- 把編號N的盤子移到C上面
- 再把B上面編號 $1 \sim n - 1$ 的盤子移到C
- 完成

```
#include <iostream>
using namespace std;
void hanoi(int n, char A, char B, char C) {
    if (n == 0)
        return;
    hanoi(n - 1, A, C, B);
    cout << "Move ring " << n
          << " from " << A << " to " << C << endl;
    hanoi(n - 1, B, A, C);
}
int main() {
    int n;
    cin >> n;
    hanoi(n, 'A', 'B', 'C');
    return 0;
}
```

枚舉

找出所有的可能性



Recursion Tree for Permutations of String "ABC"

枚舉子集合

- 給定集合 $S = \{0, 1, 2, \dots, n - 1\}$
- 輸出 S 的所有子集合 (不管順序)

枚舉子集合

Input

3

Output

{}
{2}
{1}
{1,2}
{0}
{0,2}
{0,1}
{0,1,2}

想法

- 假設有個 **ans** 陣列
- 每個數字可以選擇「放」或「不放」進 **ans** 中
 - **ans** 的所有可能性有 2^n 種
- 把 **ans** 的所有可能性印出來就是答案

維護一個全域的 ans 陣列

```
#include <iostream>
using namespace std;

int ans[25], m = 0;

void print() {
    cout << "{";
    for (int i = 0; i < m; ++i) {
        if (i)
            cout << ',';
        cout << ans[i];
    }
    cout << "}\n";
}
```

dfs(i) 函數

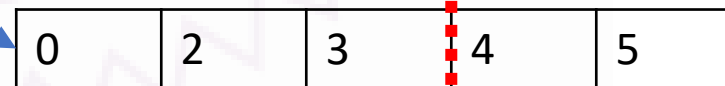
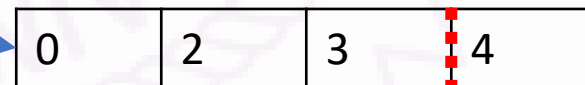
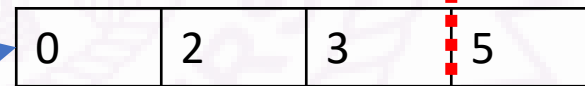
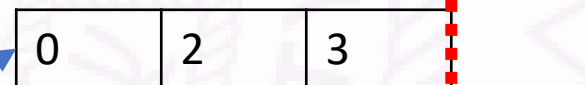
- 已經決定好 $0 \sim i - 1$ 的所有數字是否放入 **ans** 中的情況下
- 印出決定 $i \sim n - 1$ 是否放入 **ans** 中，**ans** 的所有可能

$n = 6, i = 4$



執行 dfs(4) 之前的 ans

執行 dfs(4) 後印出 ans 的所有可能



4,5 選擇放或不放的所有可能

遞迴枚舉答案

- $0 \sim n - 1$ 都決定使否放入 `ans`
印出當前 `ans` 的答案

- 決定 i 不放入 `ans` 中

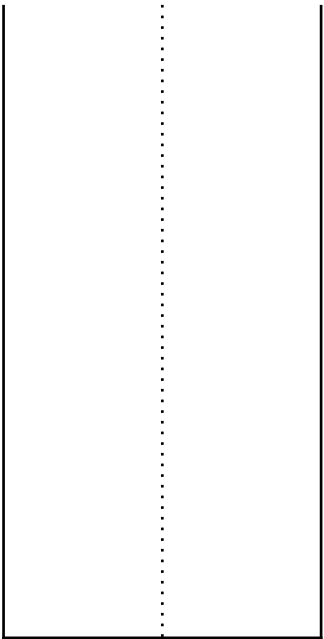
- 決定 i 要放入 `ans` 中

- 注意遞迴結束時
要恢復對 `ans` 的修改

```
int n;  
void dfs(int i) {  
    if (i == n) {  
        print();  
        return;  
    }  
    dfs(i + 1);  
    ans[m++] = i;  
    dfs(i + 1);  
    --m;  
}  
  
int main() {  
    cin >> n;  
    dfs(0);  
    return 0;  
}
```

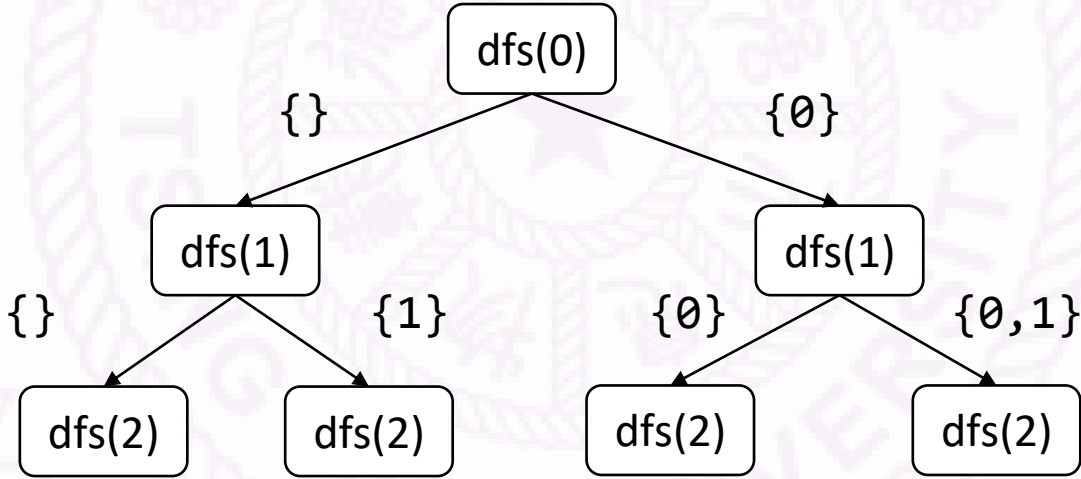
模擬執行 n=2

ans	
-----	--



Stack

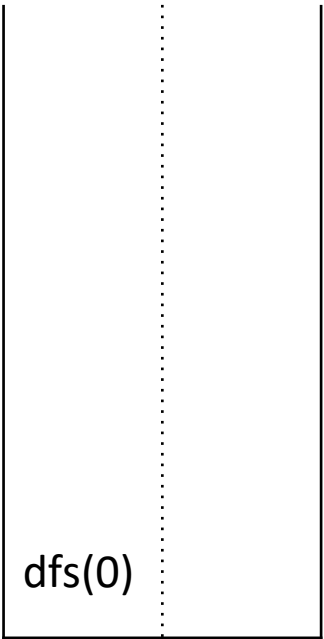
```
1. int n;  
2. void dfs(int i) {  
3.     if (i == n) {  
4.         print();  
5.         return;  
6.     }  
7.     dfs(i + 1);  
8.     ans[m++] = i;  
9.     dfs(i + 1);  
10.    --m;  
11.}
```



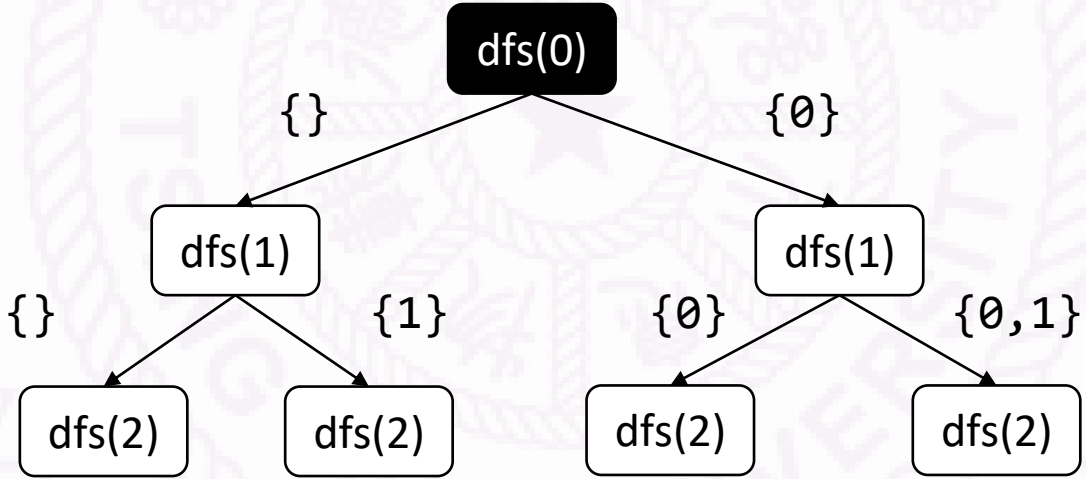
- {}
- {1}
- {0}
- {0,1}

模擬執行 n=2

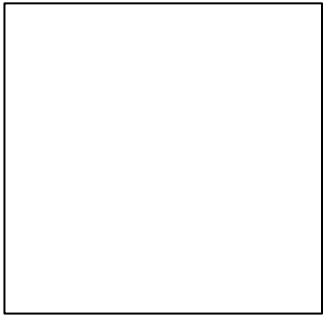
ans	
-----	--



Stack

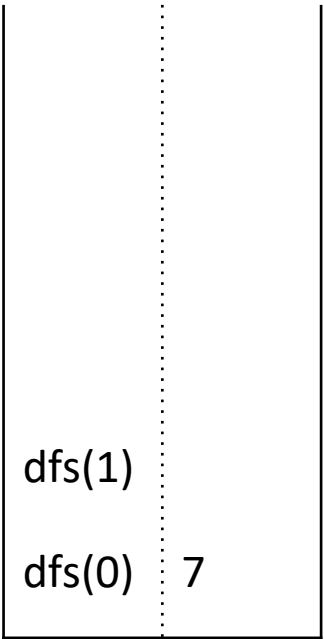


```
1. int n;  
2. void dfs(int i) {  
3.     if (i == n) {  
4.         print();  
5.         return;  
6.     }  
7.     dfs(i + 1);  
8.     ans[m++] = i;  
9.     dfs(i + 1);  
10.    --m;  
11.}
```

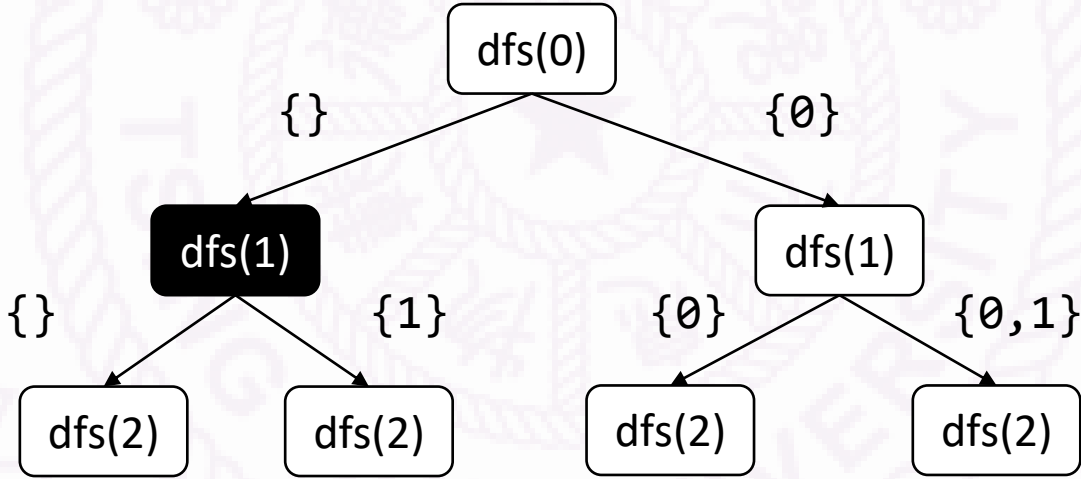


模擬執行 n=2

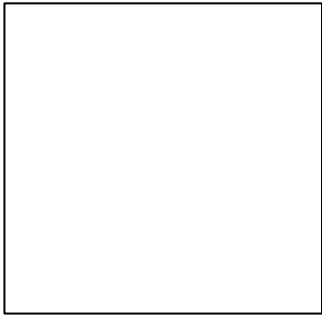
ans	
-----	--



Stack



```
1. int n;  
2. void dfs(int i) {  
3.     if (i == n) {  
4.         print();  
5.         return;  
6.     }  
7.     dfs(i + 1);  
8.     ans[m++] = i;  
9.     dfs(i + 1);  
10.    --m;  
11.}
```

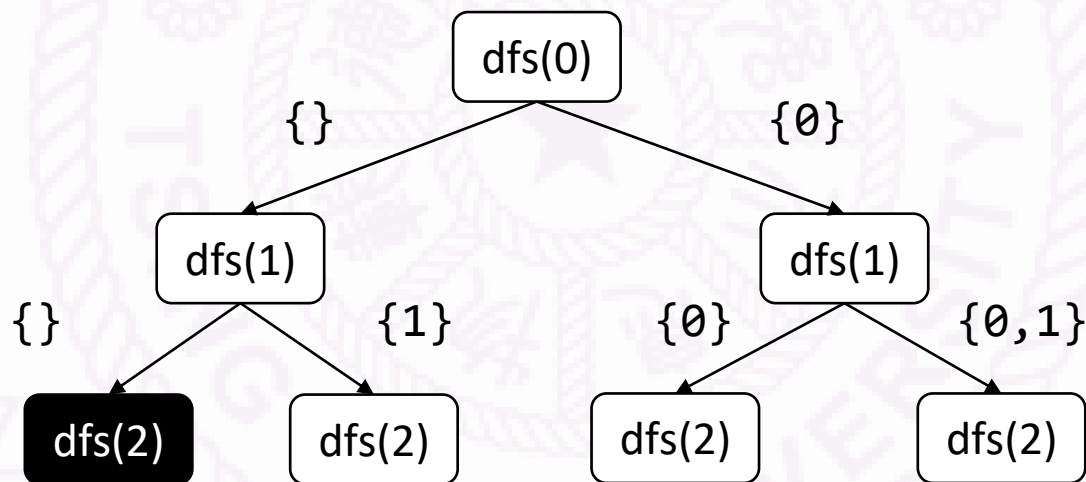


模擬執行 $n=2$

ans

dfs(2)	
dfs(1)	7
dfs(0)	7

Stack

 $\{ \}$

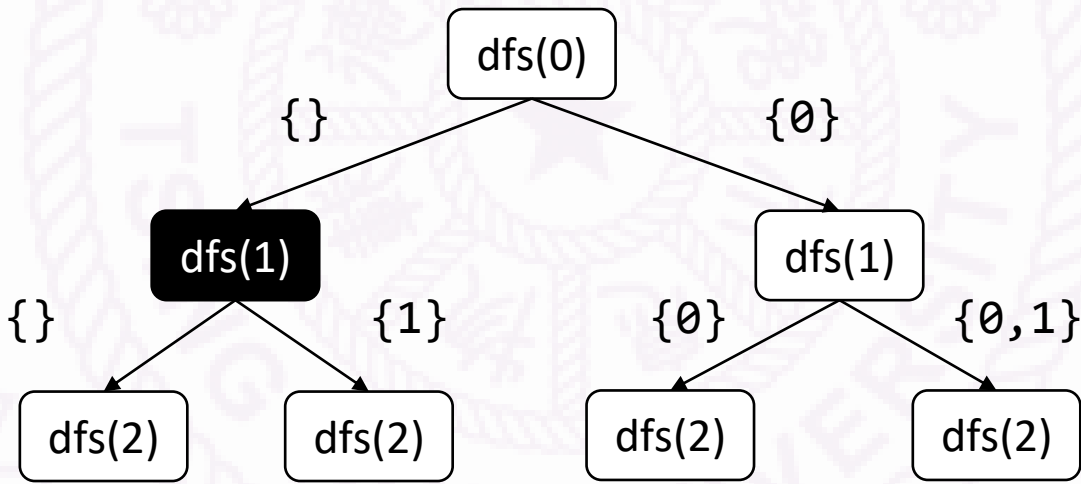
```
1. int n;
2. void dfs(int i) {
3.     if (i == n) {
4.         print();
5.         return;
6.     }
7.     dfs(i + 1);
8.     ans[m++] = i;
9.     dfs(i + 1);
10.    --m;
11.}
```

模擬執行 n=2

ans	1
-----	---

dfs(1)	7
dfs(0)	7

Stack



```
1. int n;  
2. void dfs(int i) {  
3.     if (i == n) {  
4.         print();  
5.         return;  
6.     }  
7.     dfs(i + 1);  
8.     ans[m++] = i;  
9.     dfs(i + 1);  
10.    --m;  
11.}
```

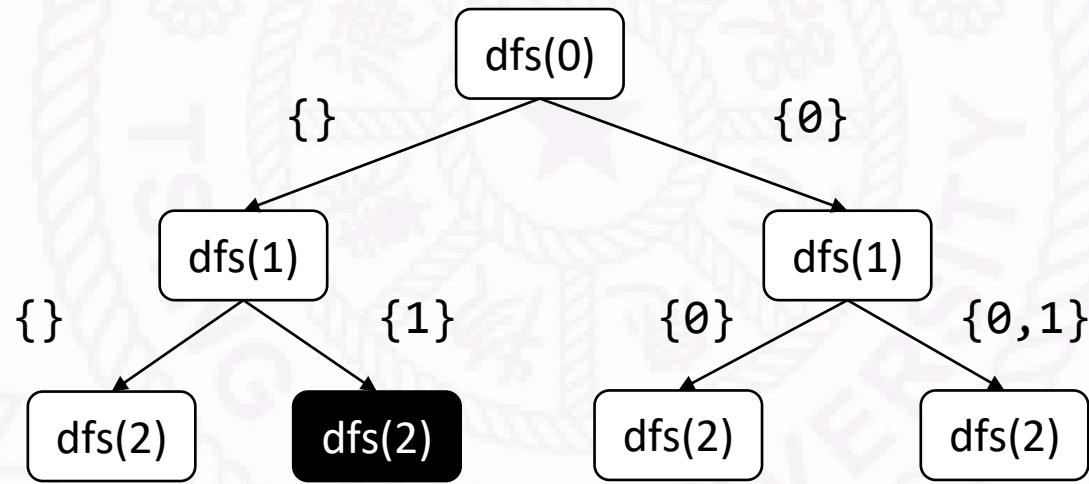
{}

模擬執行 n=2

ans	1
-----	---

dfs(2)	
dfs(1)	9
dfs(0)	7

Stack



```
1. int n;
2. void dfs(int i) {
3.     if (i == n) {
4.         print();
5.         return;
6.     }
7.     dfs(i + 1);
8.     ans[m++] = i;
9.     dfs(i + 1);
10.    --m;
11.}
```

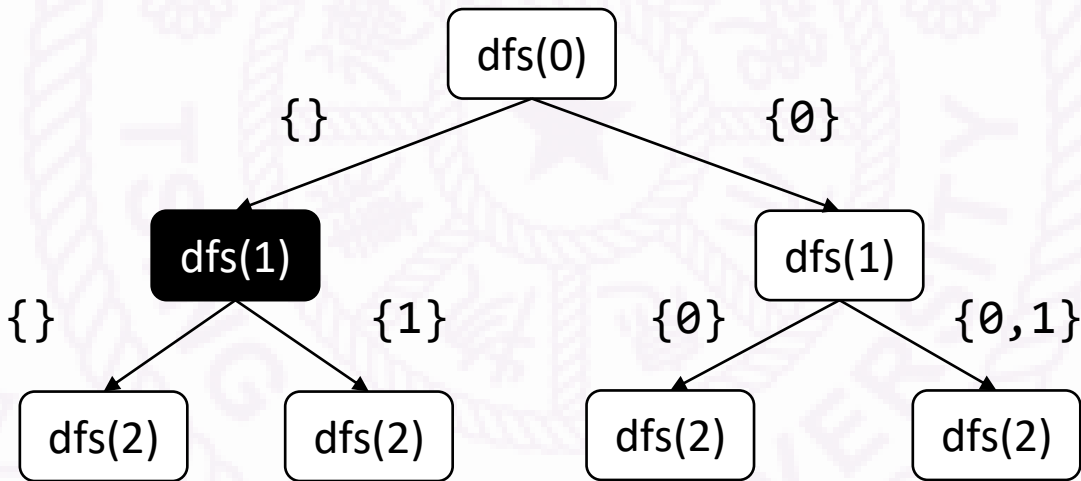
{}
{1}

模擬執行 n=2

ans	
-----	--

dfs(1)	9
dfs(0)	7

Stack



```
1. int n;  
2. void dfs(int i) {  
3.     if (i == n) {  
4.         print();  
5.         return;  
6.     }  
7.     dfs(i + 1);  
8.     ans[m++] = i;  
9.     dfs(i + 1);  
10.    --m;  
11.}
```

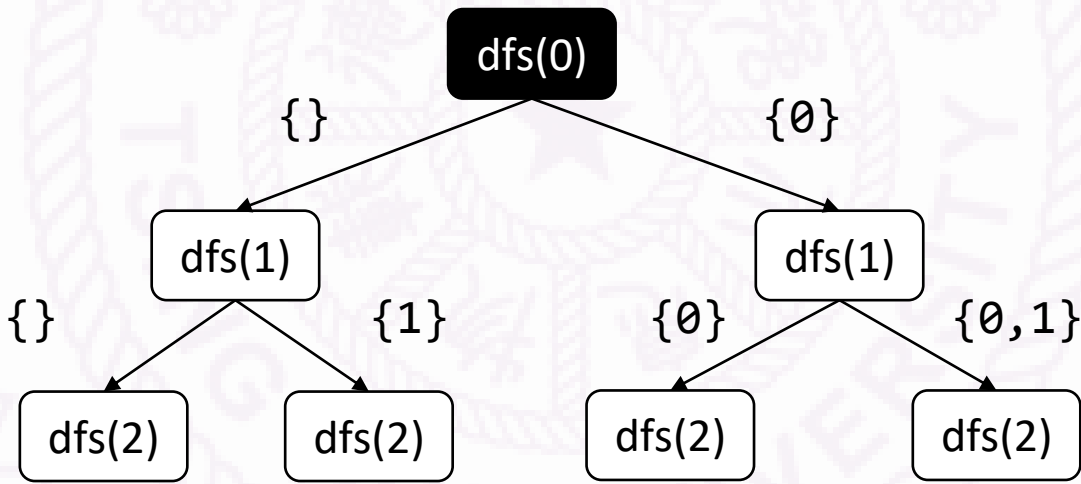
{}
{1}

模擬執行 n=2

ans	0
-----	---

dfs(0)	7
--------	---

Stack



```
1. int n;  
2. void dfs(int i) {  
3.     if (i == n) {  
4.         print();  
5.         return;  
6.     }  
7.     dfs(i + 1);  
8.     ans[m++] = i;  
9.     dfs(i + 1);  
10.    --m;  
11.}
```

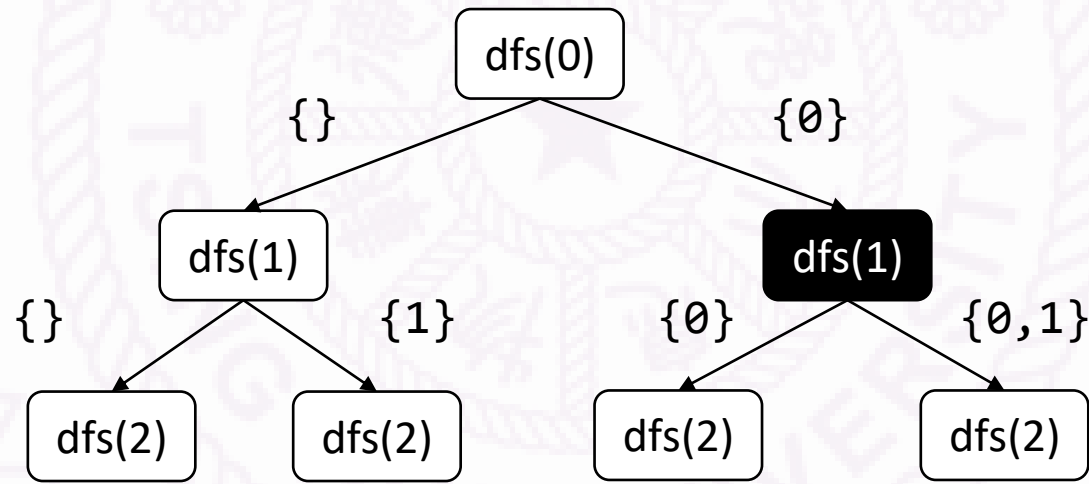
{}
{1}

模擬執行 n=2

ans	0
-----	---

dfs(1)	
dfs(0)	9

Stack



```
1. int n;  
2. void dfs(int i) {  
3.     if (i == n) {  
4.         print();  
5.         return;  
6.     }  
7.     dfs(i + 1);  
8.     ans[m++] = i;  
9.     dfs(i + 1);  
10.    --m;  
11.}
```

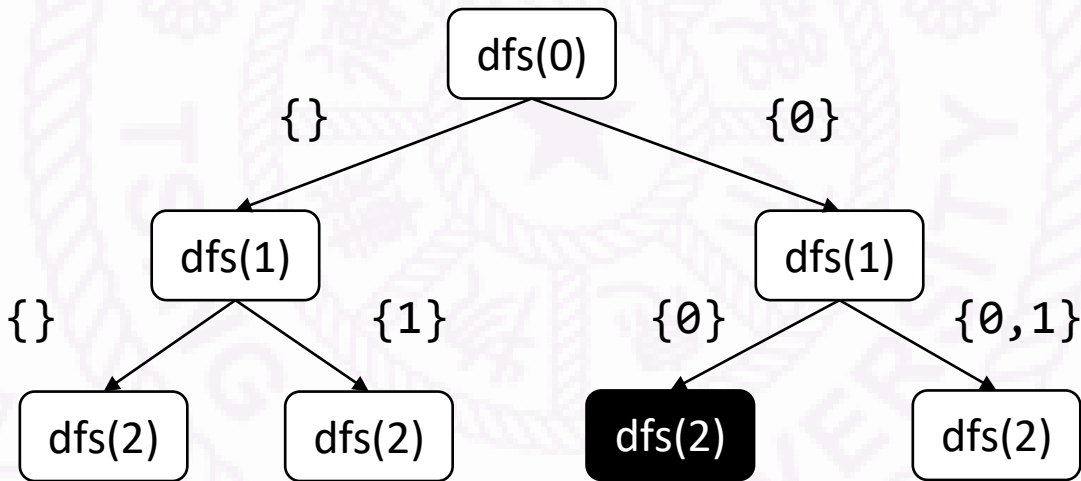
{}
{1}

模擬執行 n=2

ans	0
-----	---

dfs(2)	
dfs(1)	7
dfs(0)	9

Stack



```
1. int n;  
2. void dfs(int i) {  
3.     if (i == n) {  
4.         print();  
5.         return;  
6.     }  
7.     dfs(i + 1);  
8.     ans[m++] = i;  
9.     dfs(i + 1);  
10.    --m;  
11.}
```

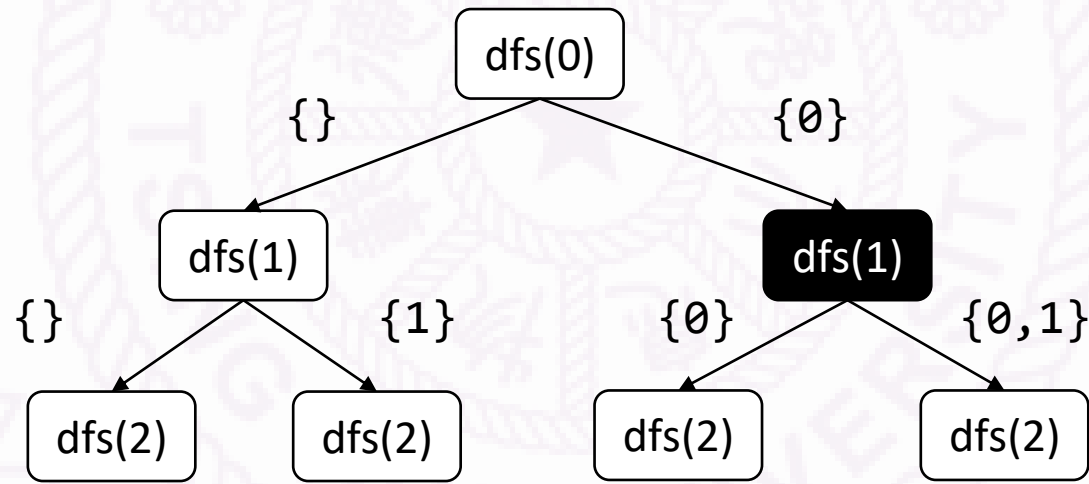
```
{}  
{1}  
{0}
```

模擬執行 n=2

ans	0	1
-----	---	---

dfs(1)	7
dfs(0)	9

Stack



```
1. int n;  
2. void dfs(int i) {  
3.     if (i == n) {  
4.         print();  
5.         return;  
6.     }  
7.     dfs(i + 1);  
8.     ans[m++] = i;  
9.     dfs(i + 1);  
10.    --m;  
11.}
```

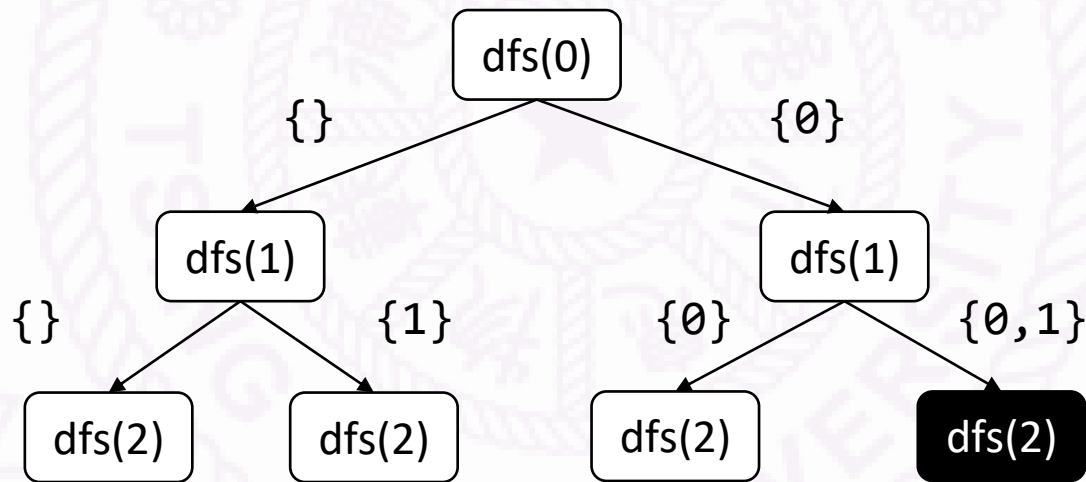
{}
{1}
{0}

模擬執行 n=2

ans	0	1
-----	---	---

dfs(2)	
dfs(1)	9
dfs(0)	9

Stack



```
1. int n;  
2. void dfs(int i) {  
3.     if (i == n) {  
4.         print();  
5.         return;  
6.     }  
7.     dfs(i + 1);  
8.     ans[m++] = i;  
9.     dfs(i + 1);  
10.    --m;  
11.}
```

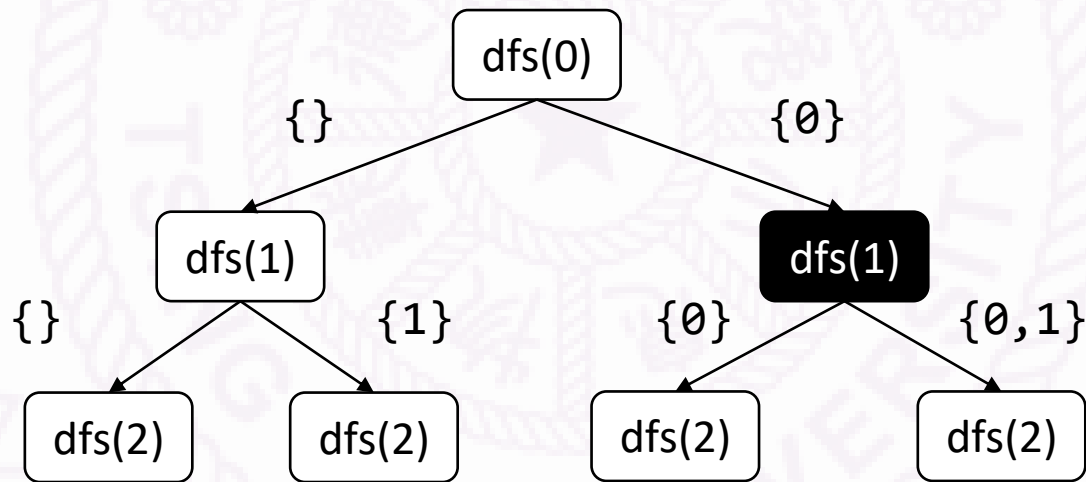
```
{}  
{1}  
{0}  
{0,1}
```

模擬執行 n=2

ans	0
-----	---

dfs(1)	9
dfs(0)	9

Stack

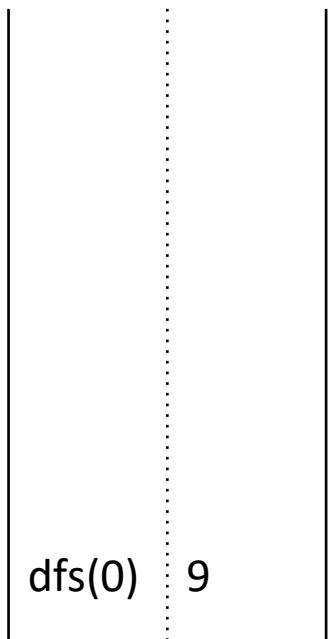


```
1. int n;  
2. void dfs(int i) {  
3.     if (i == n) {  
4.         print();  
5.         return;  
6.     }  
7.     dfs(i + 1);  
8.     ans[m++] = i;  
9.     dfs(i + 1);  
10.    --m;  
11.}
```

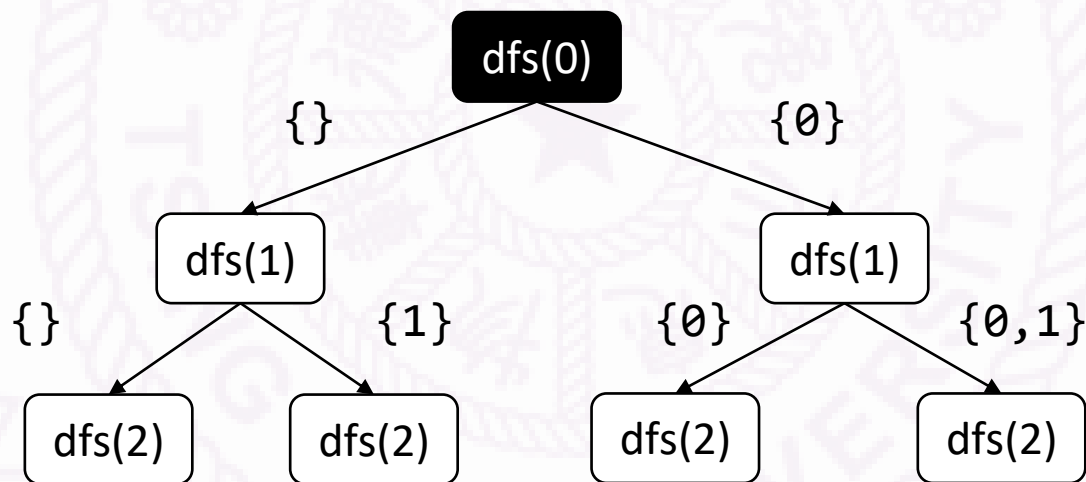
```
{}  
{1}  
{0}  
{0,1}
```

模擬執行 $n=2$

ans



Stack


$$\begin{array}{l} \{\} \\ \{1\} \\ \{0\} \\ \{0, 1\} \end{array}$$

```
1. int n;
2. void dfs(int i) {
3.     if (i == n) {
4.         print();
5.         return;
6.     }
7.     dfs(i + 1);
8.     ans[m++] = i;
9.     dfs(i + 1);
10.    --m;
11.}
```

枚舉全排列

- 給定陣列 $\{a_0, a_1, a_2, \dots, a_{n-1}\} = \{0, 1, 2, \dots, n-1\}$
- 輸出 a 的所有排列方式 (不管順序)

枚舉全排列

Input

3

Output

0 1 2

0 2 1

1 0 2

1 2 0

2 1 0

2 0 1

基於交換的全排列

```
void swap(int &a, int &b) {  
    int c = a;  
    a = b;  
    b = c;  
}
```

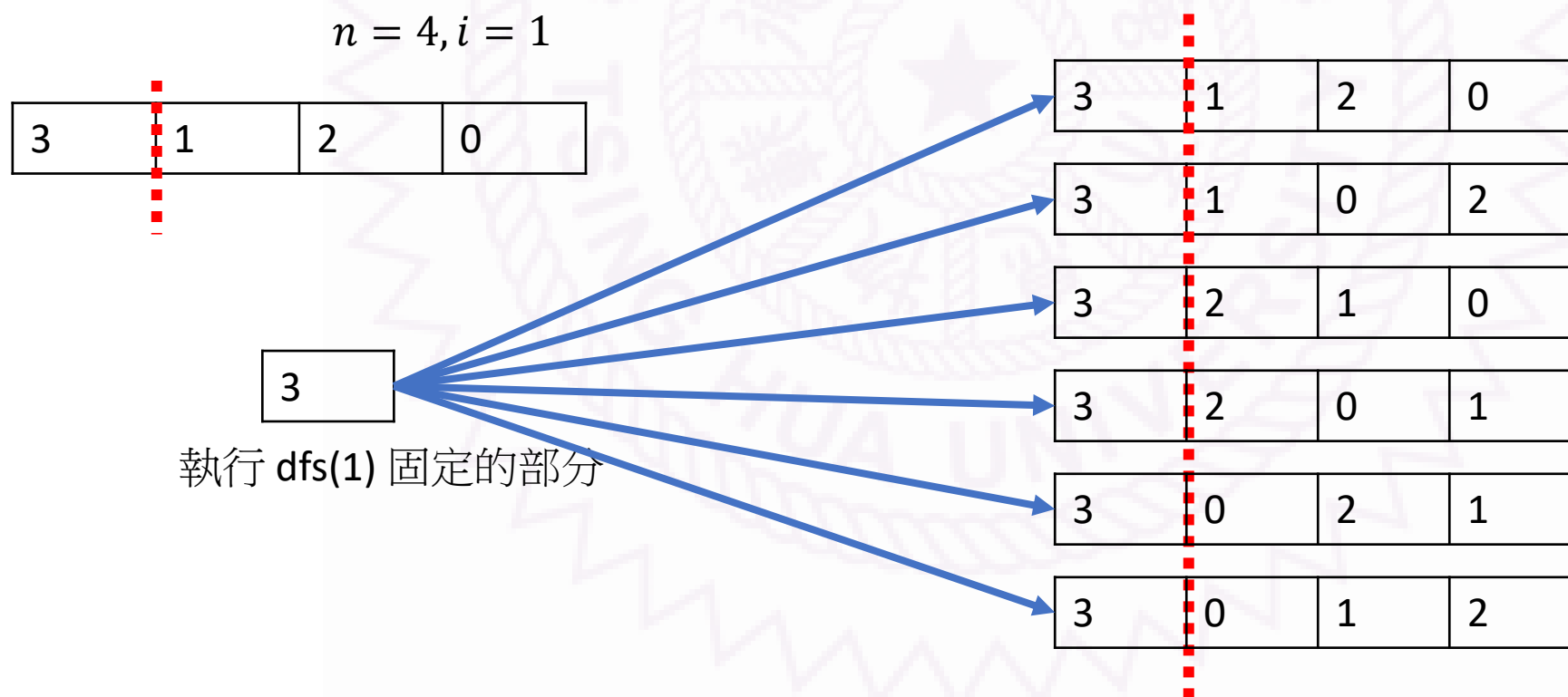
基於交換的全排列

- 由於是透過交換進行排列
- 所以不需要使用額外的陣列，直接輸出陣列 **a** 即可

```
int a[20], n; // input
void print() {
    for (int i = 0; i < n; ++i)
        cout << a[i] << " \n"[i == n - 1];
}
```

dfs(i) 函數

- 固定 $a_0 \sim a_{i-1}$ 的所有數字
- 印出接著排列 $a_i \sim a_{n-1}$ 後 a 陣列的所有情形



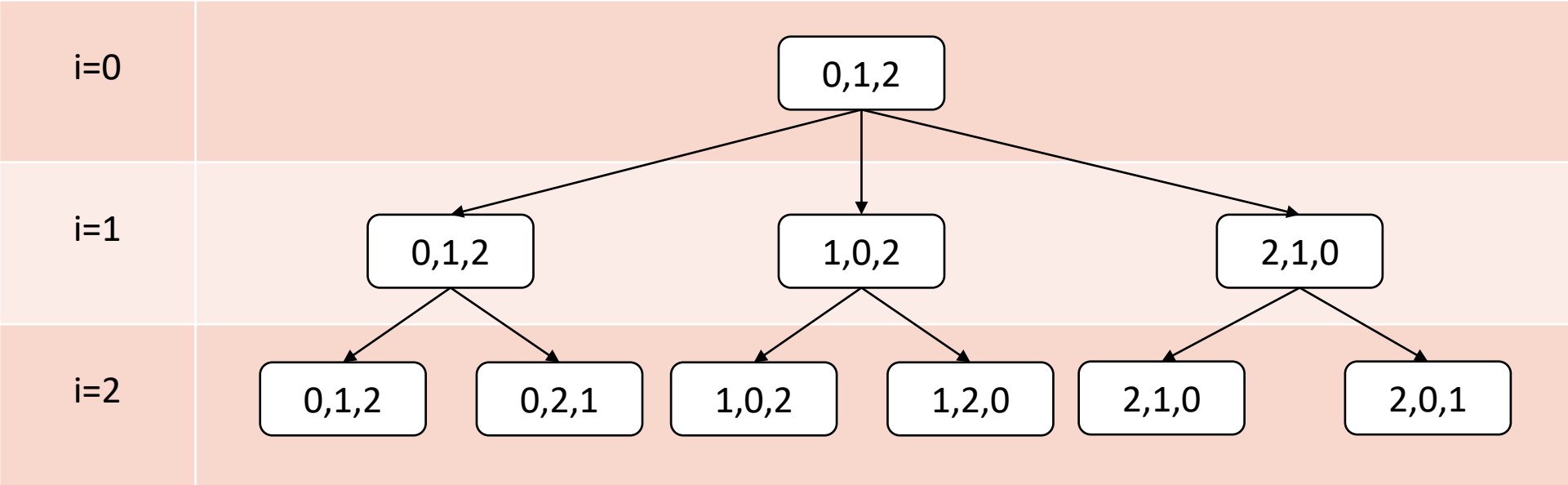
執行 `dfs(1)` 後
印出 `a` 的所有可能

基於交換的全排列

- $a_0 \sim a_{n-1}$ 都排好了
印出當前 a 陣列
- 依序枚舉 $a_i \sim a_{n-1}$ 交換至 a_i
- 注意遞迴結束時
要恢復對 a 陣列的修改

```
void dfs(int i) {  
    if (i == n) {  
        print();  
        return;  
    }  
    for (int j = i; j < n; ++j) {  
        swap(a[i], a[j]);  
        dfs(i + 1);  
        swap(a[i], a[j]);  
    }  
}  
  
int main() {  
    cin >> n;  
    for (int i = 0; i < n; ++i)  
        a[i] = i;  
    dfs(0);  
    return 0;  
}
```

模擬執行



枚舉全排列 - 按字典順序

Input

3

Output

0 1 2

0 2 1

1 0 2

1 2 0

2 0 1

2 1 0

按字典順序的全排列

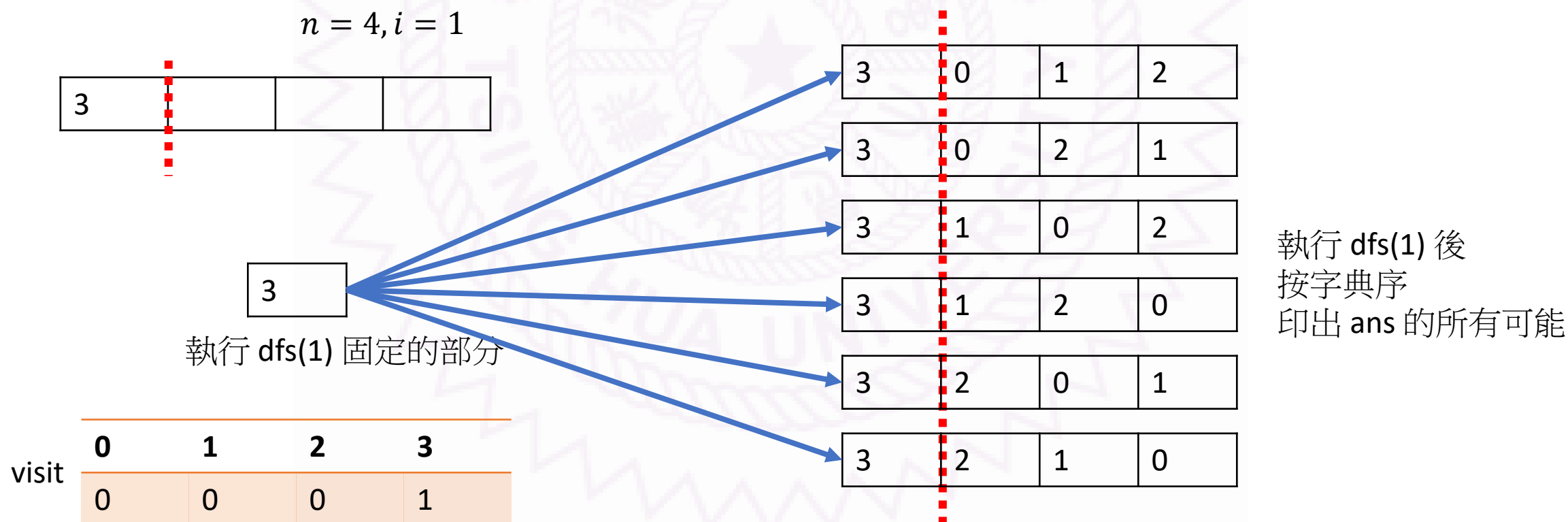
- 如同枚舉子集集合那樣，用 **ans** 紀錄當前結果
- 為了字典序，用 **visit** 陣列紀錄數字是否出現過

```
int ans[20], m;
void print() {
    for (int i = 0; i < m; ++i)
        cout << ans[i] << " \n"[i == m - 1];
}

bool visit[20]; // 紀錄某個數字有沒有出現在 ans 中
```

dfs(i) 函數

- 固定 $ans_0 \sim ans_{i-1}$ 的所有數字
- 按字典順序，將剩下的數字填入 **ans** 中按字典序印出來



按字典順序的全排列

- $ans_0 \sim ans_{n-1}$ 都排好了
印出當前 **ans** 陣列
- 由小到大把沒出現的數字
依序填入 **ans** 中接著遞迴
- 注意遞迴結束時
要恢復對 **ans** 陣列的修改

```
int n;
void dfs(int i) {
    if (i == n) {
        print();
        return;
    }
    for (int j = 0; j < n; ++j) {
        if (visit[j]) continue;
        visit[j] = true;
        ans[m++] = j;
        dfs(i + 1);
        visit[j] = false;
        --m;
    }
}
int main() {
    cin >> n;
    dfs(0);
    return 0;
}
```

枚舉全排列 – 有重複數字且按字典順序

Input

4
2 2 1 7

Output

1 2 2 7
1 2 7 2
1 7 2 2
2 1 2 7
2 1 7 2
2 2 1 7
2 2 7 1
2 7 1 2
2 7 2 1
7 1 2 2
7 2 1 2
7 2 2 1

想法： 修改剛剛的程式

- 將 j 當作陣列 a 的 index 存入 ans 中

- 由於有重複數字
需要額外把資料存入陣列 a 中

```
int a[20], n;
void dfs(int i) {
    if (i == n) {
        print();
        return;
    }
    for (int j = 0; j < n; ++j) {
        if (visit[j]) continue;
        ans[m++] = a[j];
        visit[j] = true;
        dfs(i + 1);
        visit[j] = false;
        --m;
    }
}
int main() {
    cin >> n;
    for (int i = 0; i < n; ++i)
        cin >> a[i];
    sort(a, a + n);
    dfs(0);
    return 0;
}
```


可是這樣會出現問題

Input

```
3
2 1 2
```

Output

```
1 2 2
1 2 2
2 1 2
2 2 1
2 1 2
2 2 1
```

觀察問題

- 如果把 **index** 也印出來看的話
那些重複數字的 **index** 會是不同的
- 如果我們規定有重複數字的情況
重複數字的 **index** 在排列時必須由小到大
就可以解決這個問題了

a

0	1	2
1	2	2

ans

0	1	2
1	2	2

0	2	1
1	2	2

1	0	2
2	1	2

1	2	0
2	2	1

2	0	1
2	1	2

2	1	0
2	2	1

包含重複數字 按字典順序的全排列

- 讓重複的數字在同一次遞迴中只被枚舉一次
- 這樣就可以讓重複數字被枚舉時 **index** 都是由小到大的
- 只需要用一個 **prev** 記錄前一次枚舉的數字就能輕鬆判斷

```
int a[20], n;
void dfs(int i) {
    if (i == n) {
        print();
        return;
    }
    for (int prev = -1, j = 0; j < n; ++j) {
        if (visit[j] || prev == a[j]) continue;
        ans[m++] = a[j];
        visit[j] = true;
        dfs(i + 1);
        visit[j] = false;
        --m;
        prev = a[j];
    }
}
int main() {
    cin >> n;
    for (int i = 0; i < n; ++i) cin >> a[i];
    sort(a, a + n);
    dfs(0);
    return 0;
}
```

回家作業：
 n 個數字取 k 個排列 – 有重複數字且按字典序

Input

```
5 3  
7 1 2 2 2
```

Output

```
1 2 2  
1 2 7  
1 7 2  
2 1 2  
2 1 7  
2 2 1  
2 2 2  
2 2 7  
2 7 1  
2 7 2  
7 1 2  
7 2 1  
7 2 2
```