



中國人民大學
RENMIN UNIVERSITY OF CHINA

第8讲 递归(2)

余力

buaayuli@ruc.edu.cn



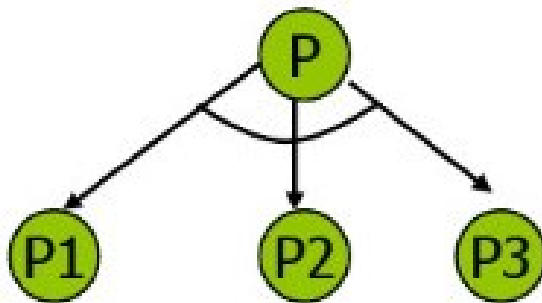
中國人民大學
RENMIN UNIVERSITY OF CHINA



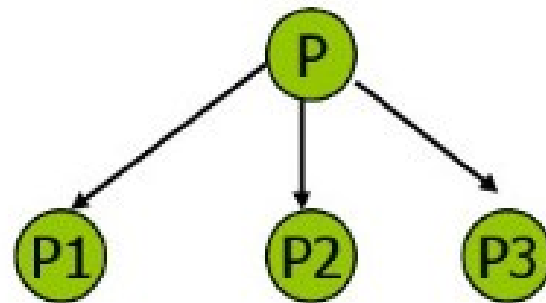
4. 递归分析工具-与或图

与或图

- 与图: 把一个原问题**分解**为若干个子问题, P_1, P_2, P_3, \dots 可用“与图”表示; P_1, P_2, P_3, \dots 对应的子问题节点称为“与节点”。
- 或图: 把一个原问题**变换**为若干个子问题, P_1, P_2, P_3, \dots 可用“或图”表示; P_1, P_2, P_3, \dots 对应的子问题节点称为“或节点”。



与



或

阶乘

■ 阶乘 $n!$ 的计算

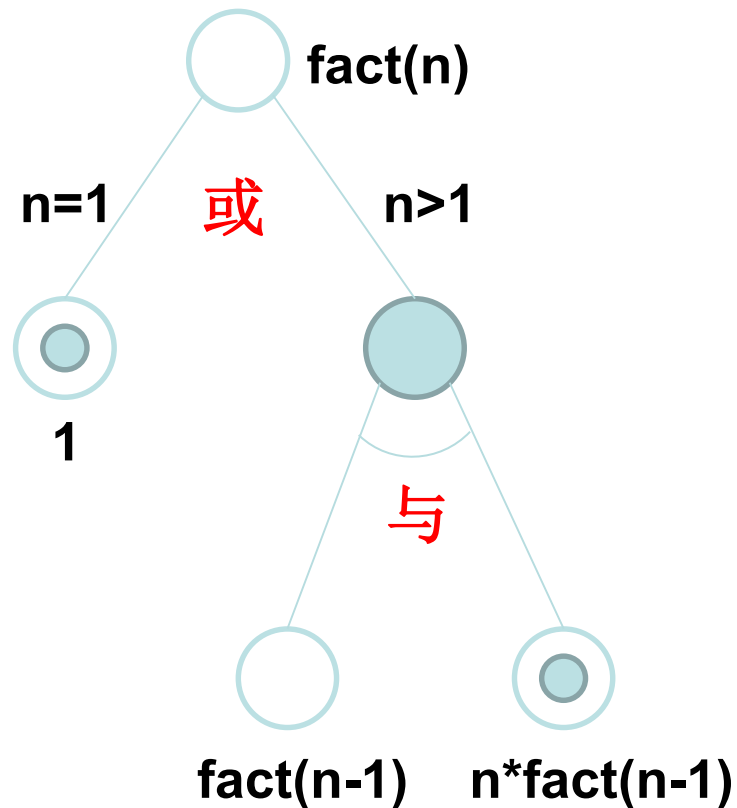
➤ Base case:

- $n=1$ 时, 返回结果1

➤ Inductive case:

- $n>1$ 时, 返回 $n!=n*(n-1)!$

如何用图表
形式表示?



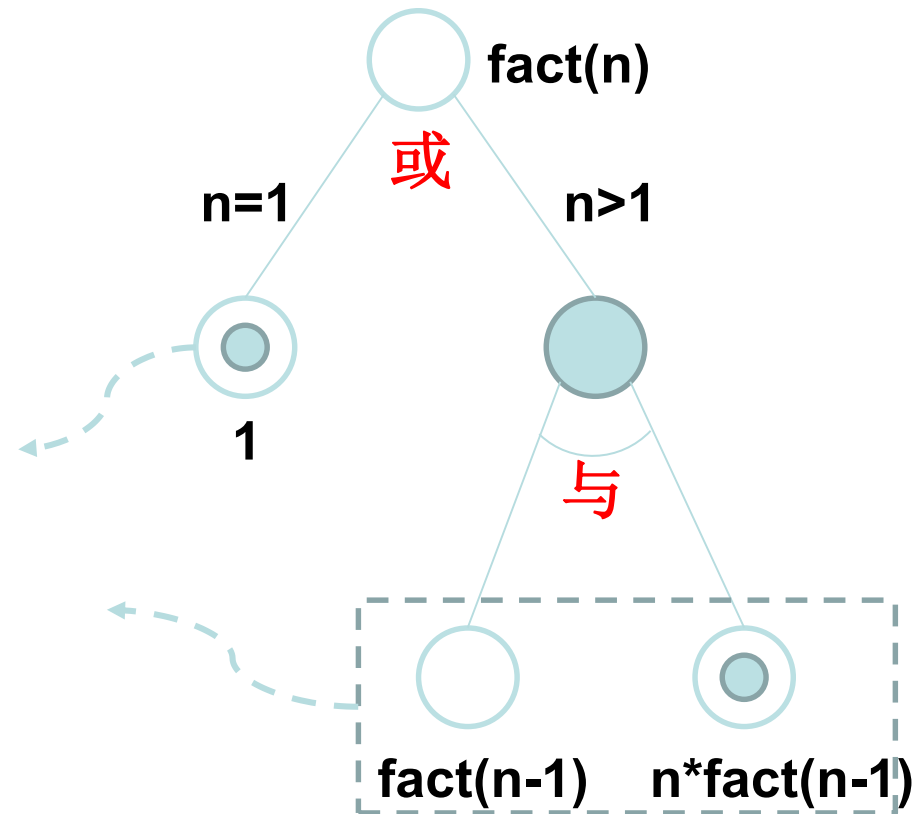
与或图与递归程序编写

```
#include <iostream>

using namespace std;

int fact (int n) {
    if (n==1) return 1;
    else {
        int fn1 = fact(n-1);
        return n*fn1;
    }
}

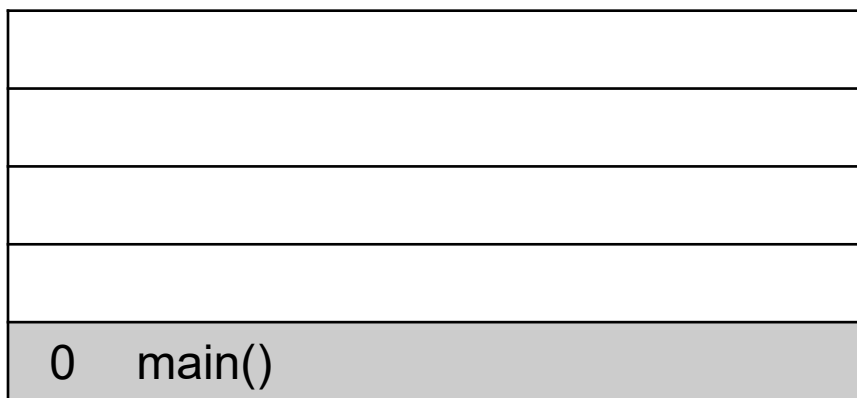
int main () {
    cout << fact(3);
}
```



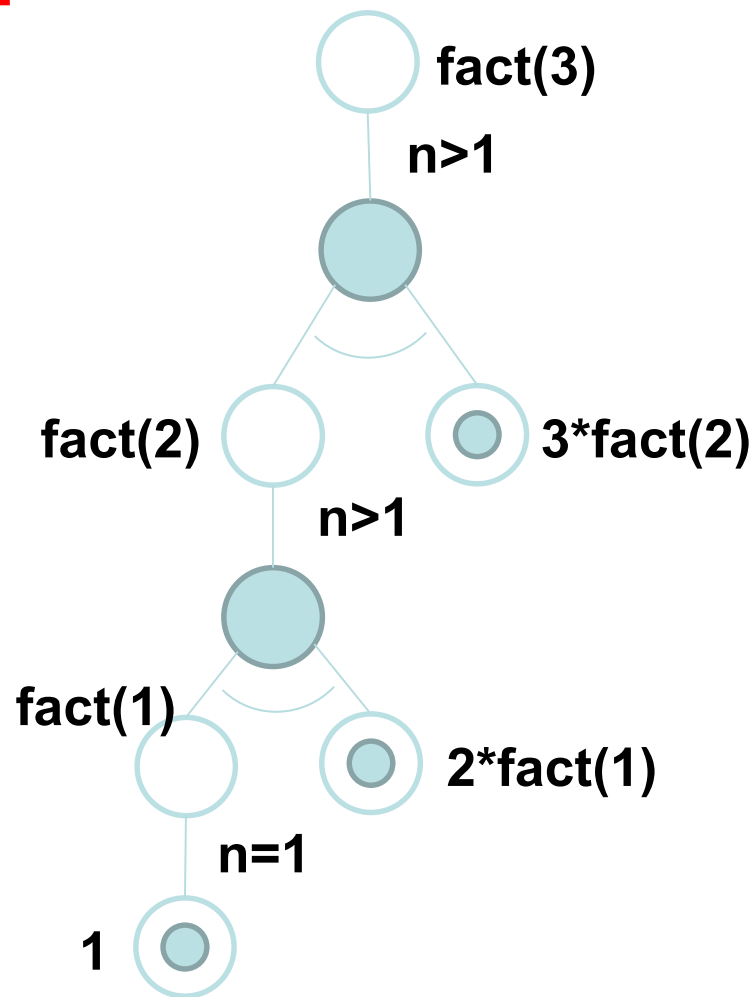
Fact(阶乘).cpp

执行过程

```
int fact (int n) {  
    if (n==1) return 1;  
    else {  
        int fn1 = fact(n-1);  
        return n*fn1;  
    }  
}  
int main () {  
    cout << fact(3);  
}
```



函数调用栈(Call Stack)



与或图

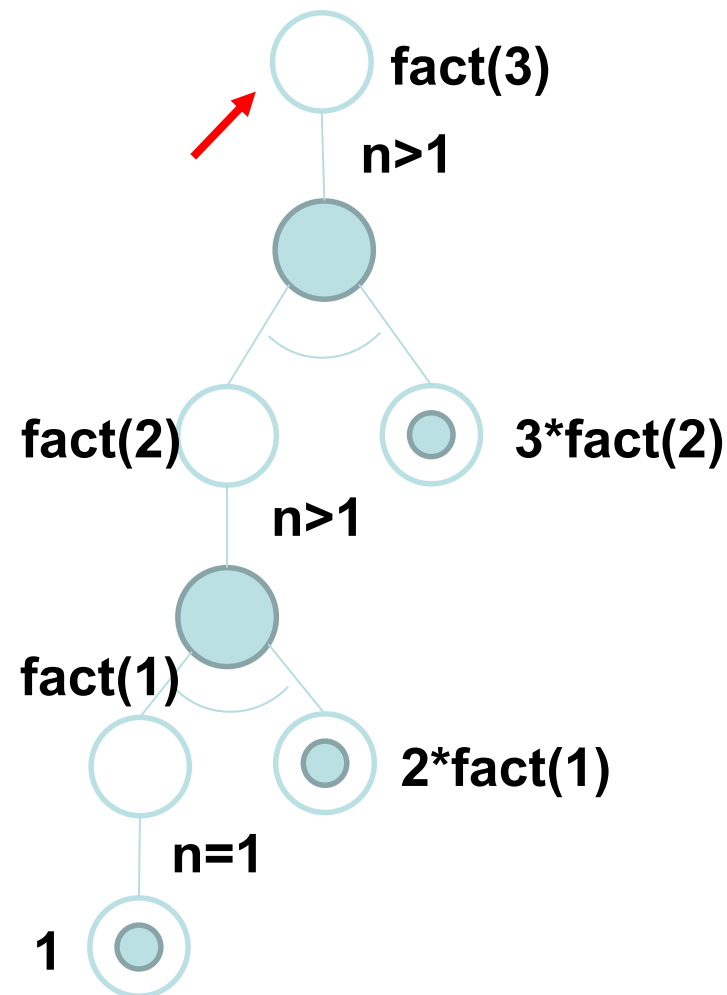
```

→ int fact (int n) {
    if (n==1) return 1;
    else {
        int fn1 = fact(n-1);
        return n*fn1;
    }
}
int main () {
    cout << fact(3);
}

```

1	fact(3)
0	main()

函数调用栈(Call Stack)



与或图

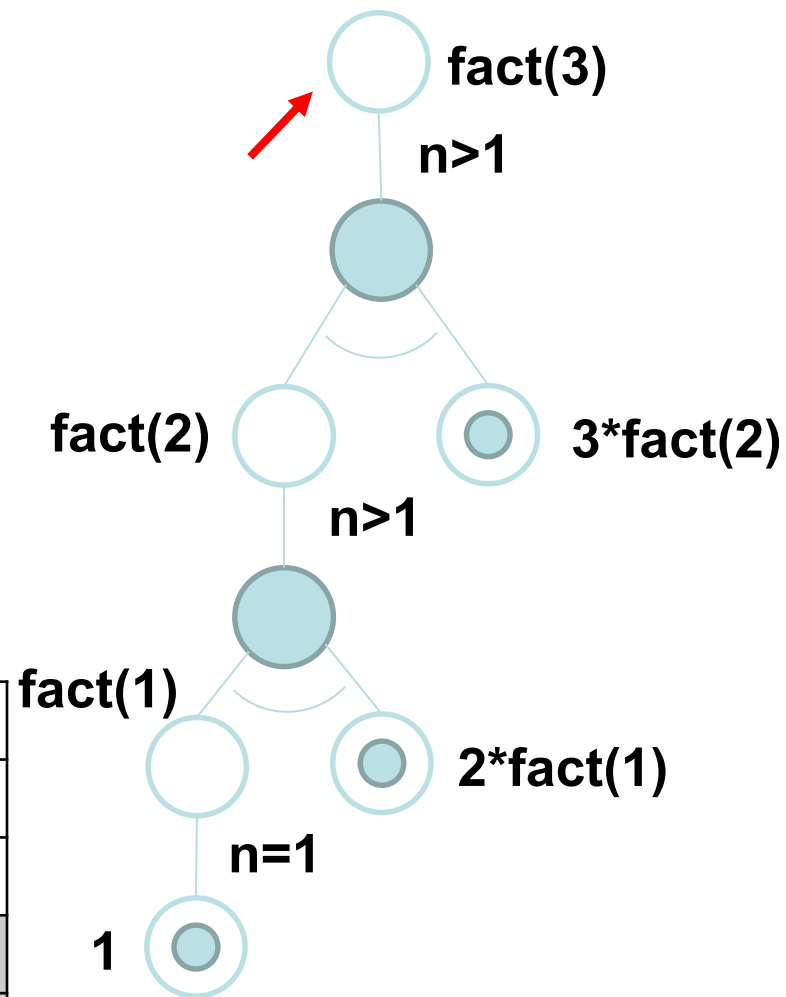
```

int fact (int n) {
    if (n==1) return 1;
    else {
        int fn1 = fact(n-1);
        return n*fn1;
    }
}
int main () {
    cout << fact(3);
}

```

1	fact(3)
0	main()

函数调用栈(Call Stack)



与或图

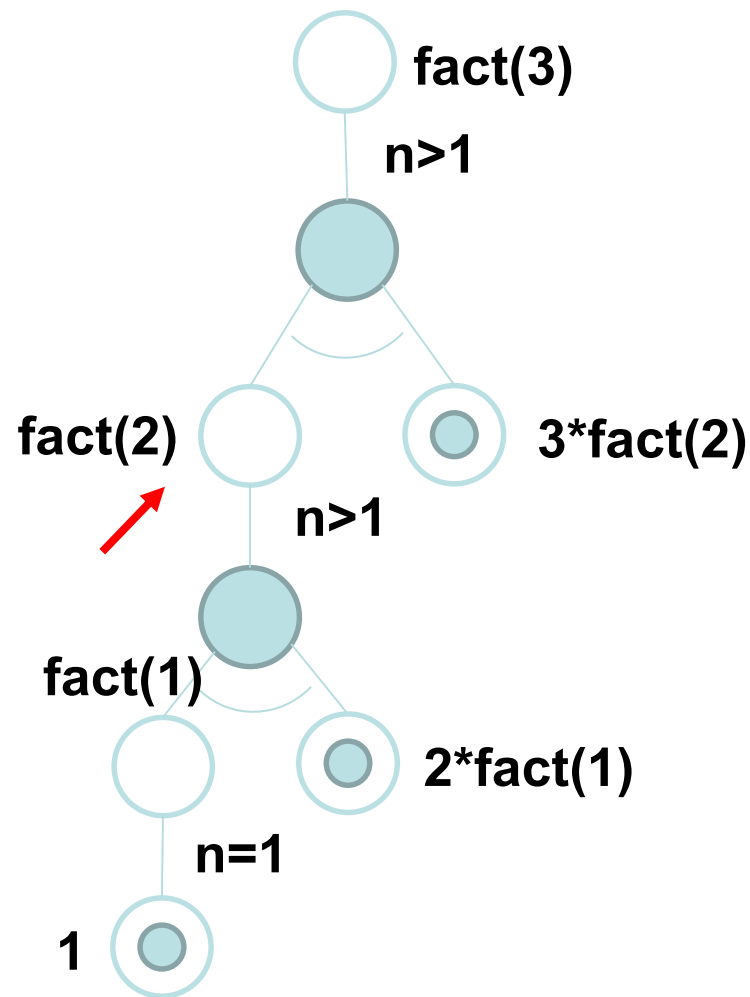

```

→ int fact (int n) {
    if (n==1) return 1;
    else {
        int fn1 = fact(n-1);
        return n*fn1;
    }
}
int main () {
    cout << fact(3);
}

```

2	fact(2)
1	fact(3)
0	main()

函数调用栈(Call Stack)



与或图

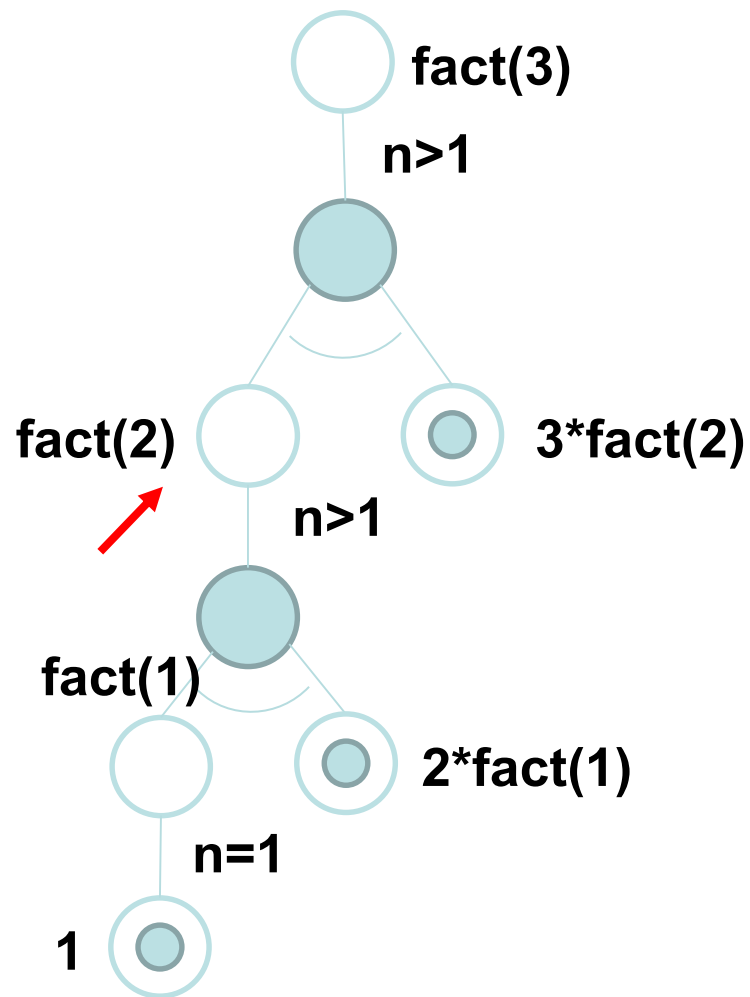
```

int fact (int n) {
    if (n==1) return 1;
    else {
        int fn1 = fact(n-1);
        return n*fn1;
    }
}
int main () {
    cout << fact(3);
}

```

2	fact(2)
1	fact(3)
0	main()

函数调用栈(Call Stack)



与或图

→

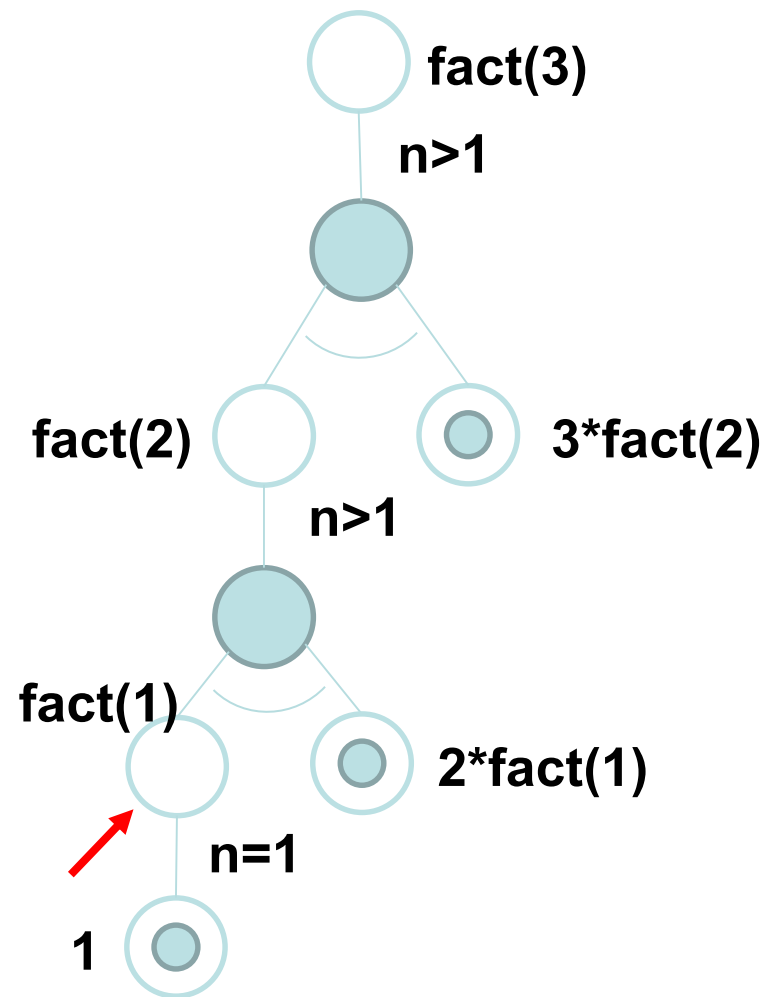
```

int fact (int n) {
    if (n==1) return 1;
    else {
        int fn1 = fact(n-1);
        return n*fn1;
    }
}
int main () {
    cout << fact(3);
}

```

3	fact(1)
2	fact(2)
1	fact(3)
0	main()

函数调用栈(Call Stack)



与或图

```

int fact (int n) {
    if (n==1) return 1;
    else {
        int fn1 = fact(n-1);
        return n*fn1;
    }
}

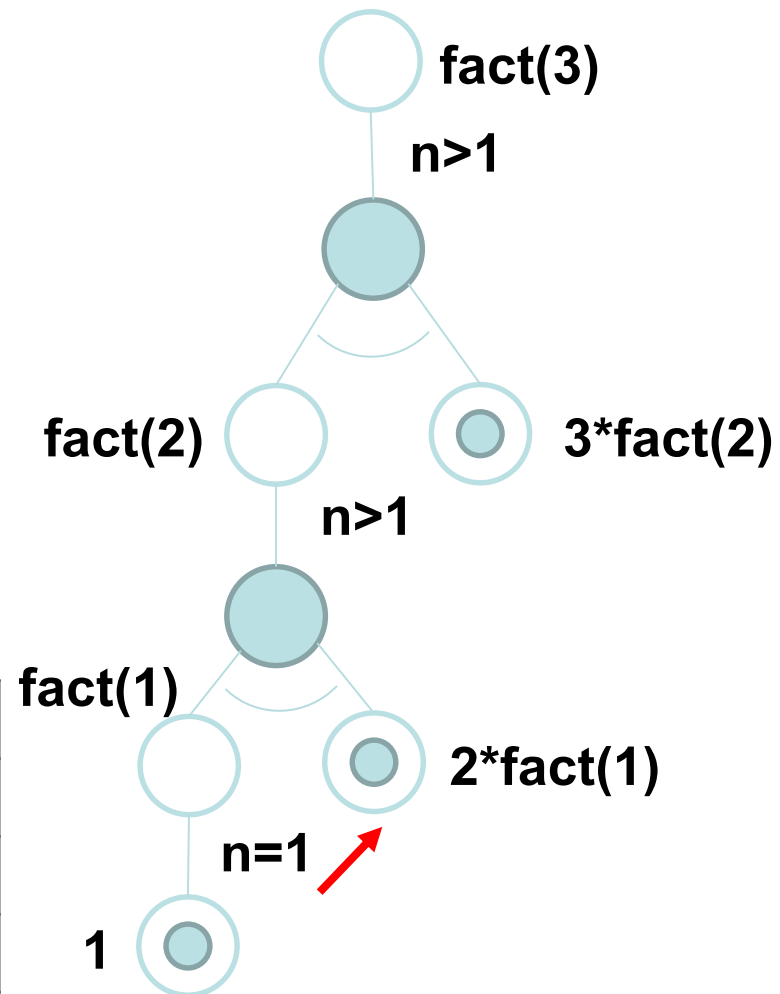
int main () {
    cout << fact(3);
}

```



2	fact(2)
1	fact(3)
0	main()

函数调用栈(Call Stack)



与或图

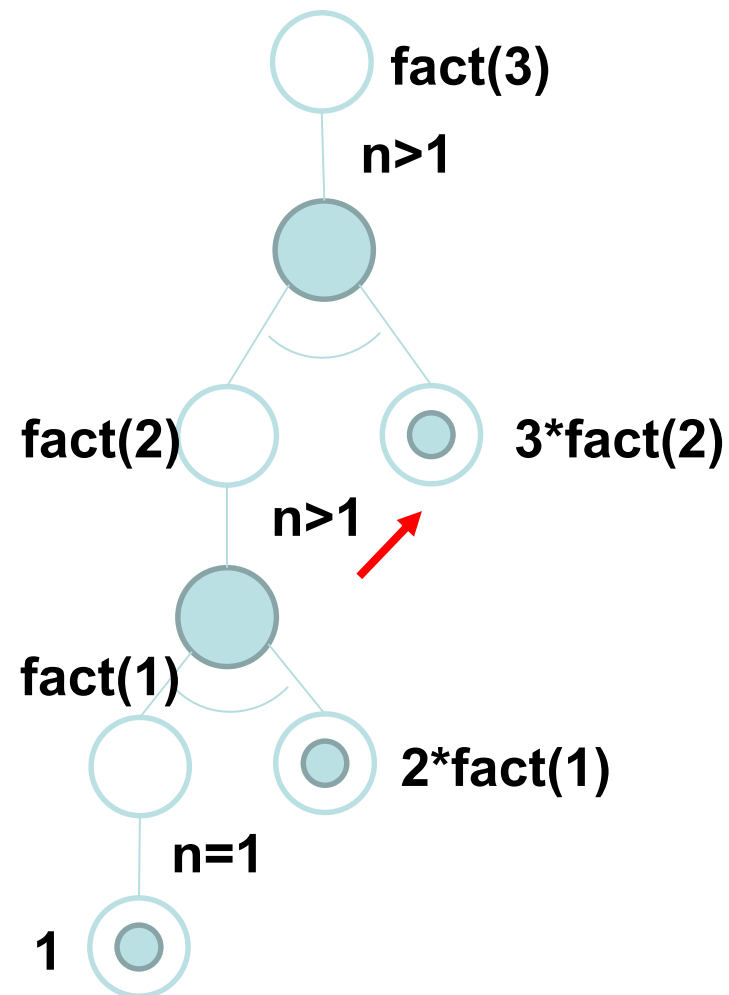
```

int fact (int n) {
    if (n==1) return 1;
    else {
        int fn1 = fact(n-1);
        return n*fn1;
    }
}
int main () {
    cout << fact(3);
}

```

1	fact(3)
0	main()

函数调用栈(Call Stack)

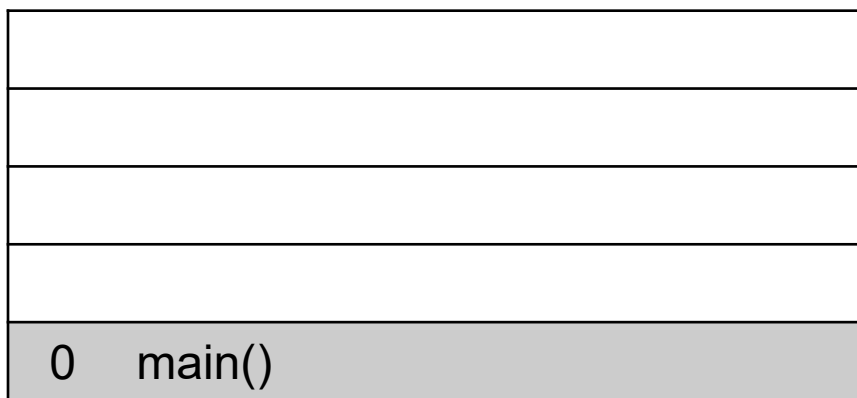


与或图

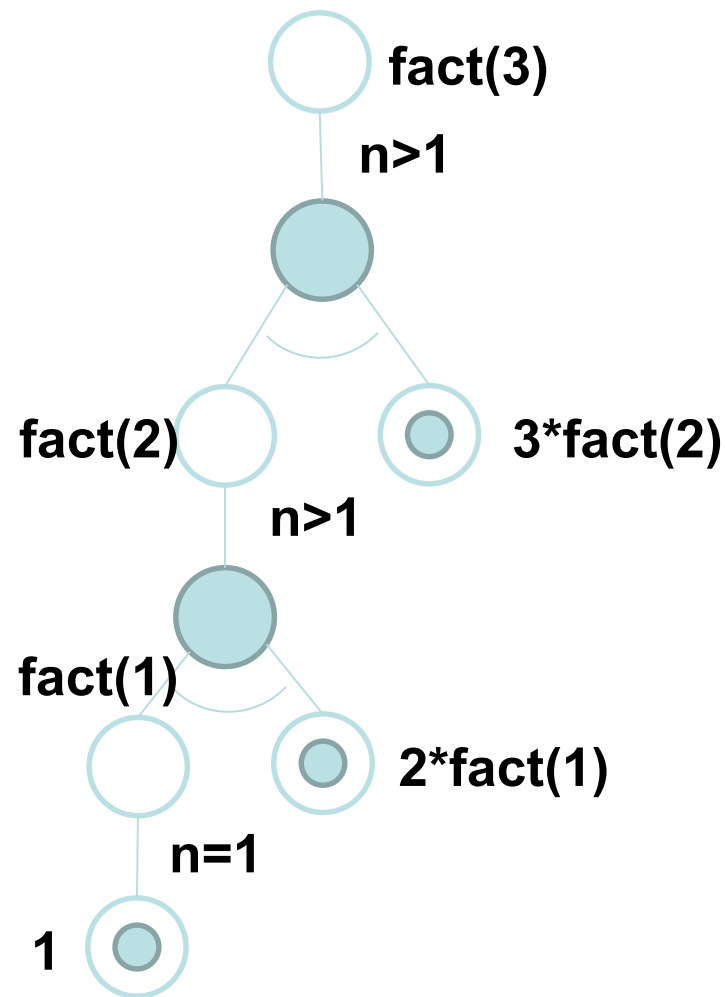
```

int fact (int n) {
    if (n==1) return 1;
    else {
        int fn1 = fact(n-1);
        return n*fn1;
    }
}
int main () {
    cout << fact(3);
}

```



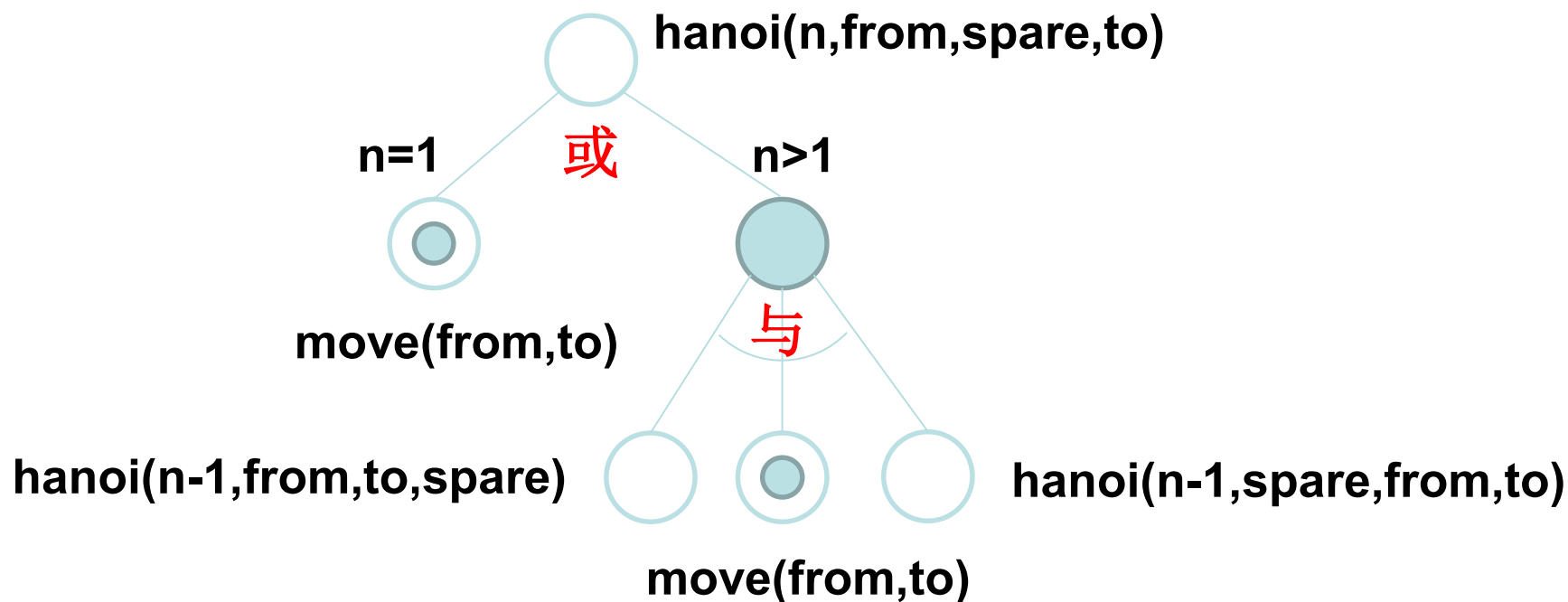
函数调用栈(Call Stack)



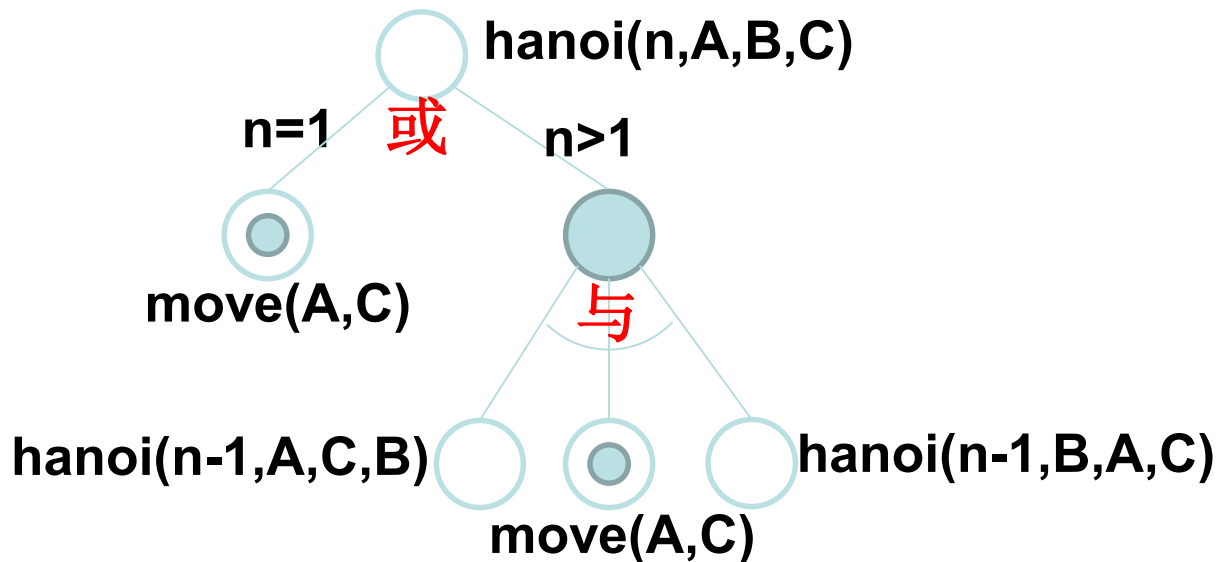
与或图

汉诺塔问题

- 若汉诺塔问题表示为 $\text{hanoi}(n, \text{from}, \text{spare}, \text{to})$ ，移动一个盘子操作为 $\text{move}(\text{from}, \text{to})$
 - 画出 $\text{hanoi}(n, \text{from}, \text{spare}, \text{to})$ 的与或图
 - 分析 $\text{hanoi}(4, \text{from}, \text{spare}, \text{to})$ 的情况（板书）



汉诺塔



```
void move (int n, char A, char B, char C) {  
    if (n == 1)  
        cout << "move from " << A << " to " << C << endl;  
    else {  
        move (n-1, A, C, B);  
        cout << "move from " << A << " to " << C << endl;  
        move (n-1, B, A, C);  
    }  
}  
  
int main () {  
    int n = 4;  
    char A = 'A', B = 'B', C = 'C';  
    move (n, A, B, C);  
}
```

#123 汉诺塔.cpp


```

#include <stdio.h>
int main()
{ void hanoi(int n, char one, char two, char three);
  int m;
  printf("the number of disks:");
  scanf("%d",&m);
  printf("move %d disks:\n",m);
  hanoi(m,'A','B','C');
}

```

```

void hanoi(int n, char one, char two, char three)

```

```

{ void move(char x,char y);
  if(n==1) move(one,three);
  else
  { hanoi(n-1,one,three,two);
    move(one,three);
    hanoi(n-1,two,one,three);
  }
}

```

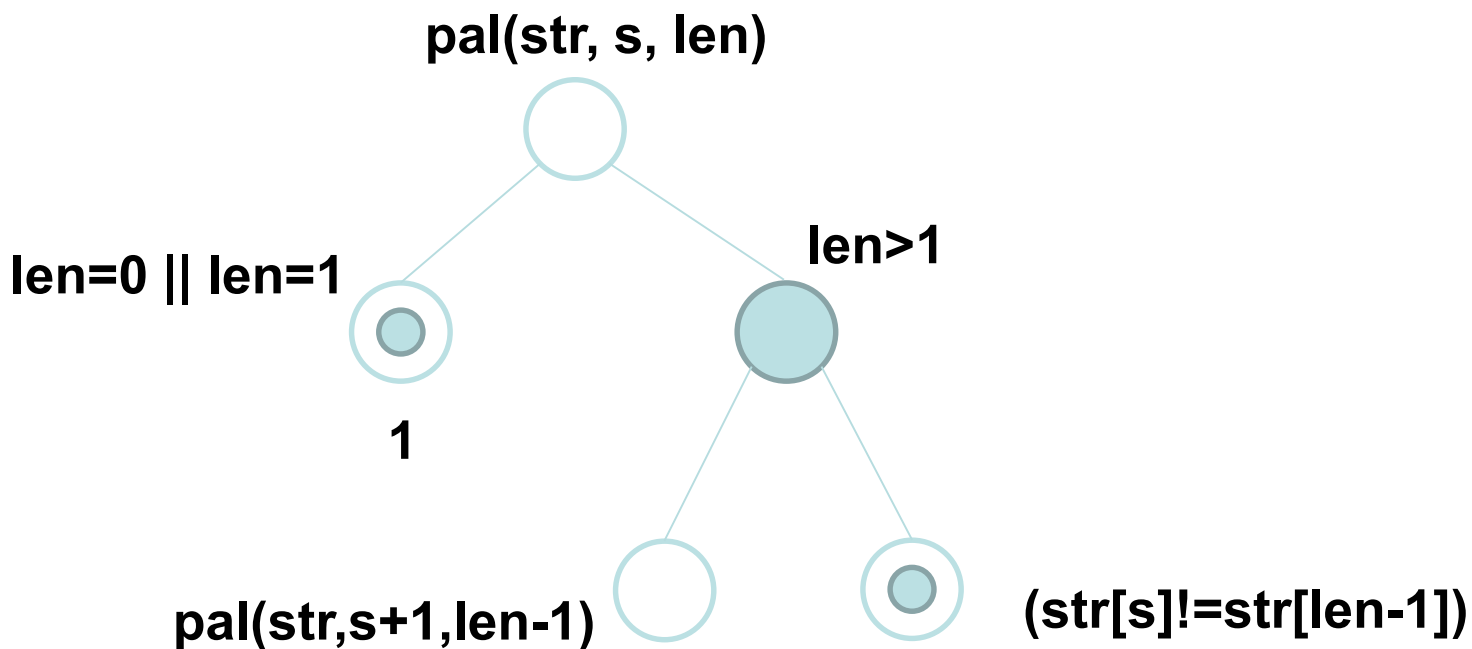
```

void move(char x,char y)
{
  printf("%c-->%c\n",x,y);
}

```


回文判断

- 若问题表示为 $\text{pal}(\text{str}, s, \text{len})$
 - 画出 $\text{pal}(\text{str}, s, \text{len})$ 的与或图
 - 分析 $\text{pal}(\text{str}, 0, 11)$ 的情况 (板书)



判断回文

```
#include<iostream>
using namespace std;
#include<string.h>
int pal(char str[], int low, int high) {
    if(high<=low)
        return 1; // base case
    else
        if (str[low]!=str[high]) return 0;
        else {low=low+1;high=high-1;}
    return pal(str,low+1,high-1);
    return (str[low]==str[high]) && pal(str,low+1,high-1)
}
int main() {
    char str[100] = "madamimadam";
    int len = (int)strlen(str);
    cout << pal(str,0,len-1) << endl;
```

#110 回文判断.cpp

```

int main()
{
    int n,j,i;
    char a[1000];
    gets(a);
    n=strlen(a);
    for ( i = 0,j=n-1; i < (n+1)/2; i++,j--)
        if (a[i]!=a[j])
            { printf("No"); break;}
    if (i==(n+1)/2) printf("Yes");
    return 0;
}

```

```

int pal(char str[], int low, int high) {
    if (high <= low)
        return 1; // base case
    else if (str[low] != str[high])
        return 0;
    else
        return pal(str, low + 1, high - 1);
}

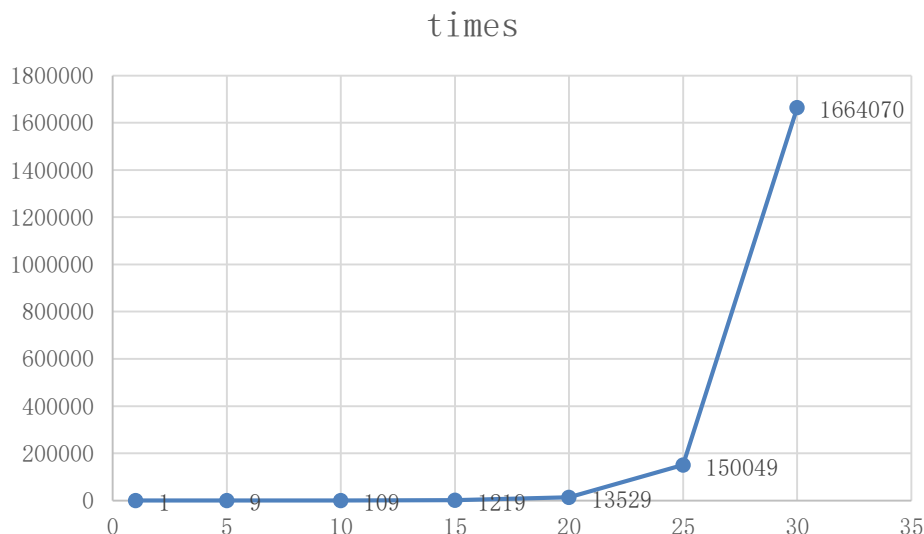
```

斐波那契数列

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

int times = 0;
int fib(int n) {
    times ++;
    if (n==1||n==2) return 1;
    return fib(n-1)+fib(n-2)+fib(n-3);
}

int main () {
    cout << fib(6) << endl;
    cout << "times: " << times << endl;
}
```

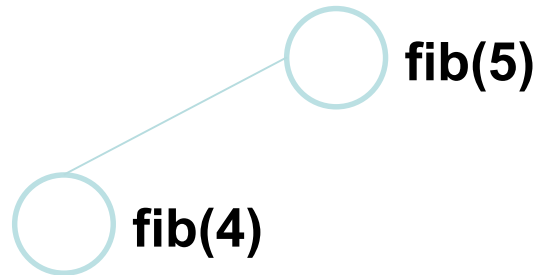


指数级增长!

Fibonacci.cpp

使用memo避免重复计算

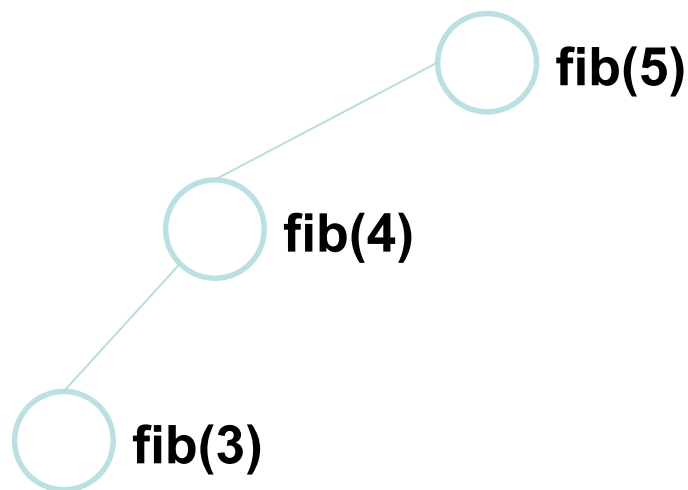
- 使用数组存储memo[1]-memo[n]分别存储已经计算出的fib(1)到fib(n)的值
- 数组的初值设为-1,表示“未计算”



memo 数组	下标	1	2	3	4	5
	取值	-1	-1	-1	-1	-1

使用memo避免重复计算

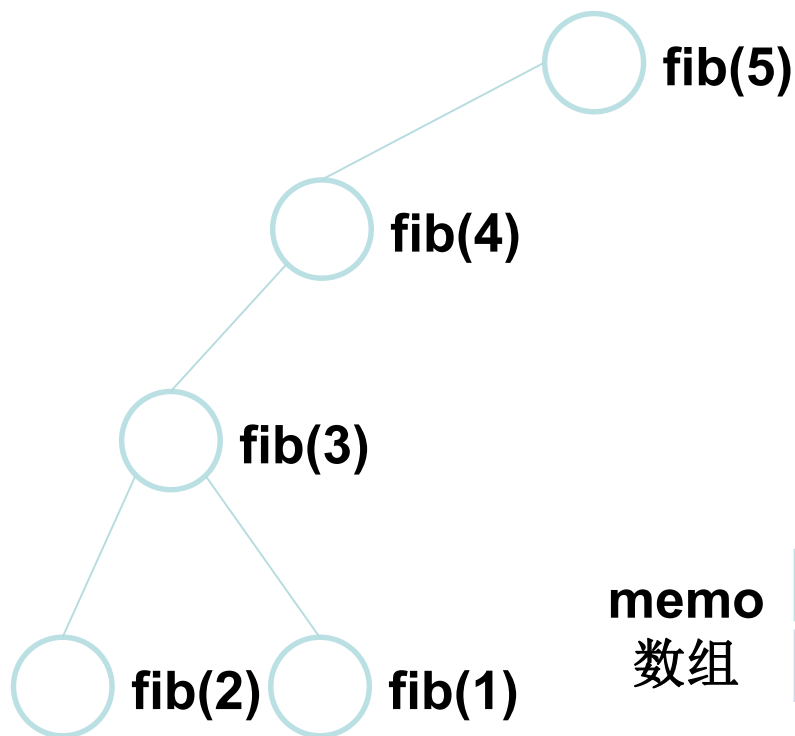
- 使用数组存储memo[1]-memo[n]分别存储已经计算出的fib(1)到fib(n)的值
- 数组的初值设为-1,表示“未计算”



memo 数组	下标	1	2	3	4	5
	取值	-1	-1	-1	-1	-1

使用memo避免重复计算

- 使用数组存储memo[1]-memo[n]分别存储已经计算出的fib(1)到fib(n)的值
- 数组的初值设为-1,表示“未计算”

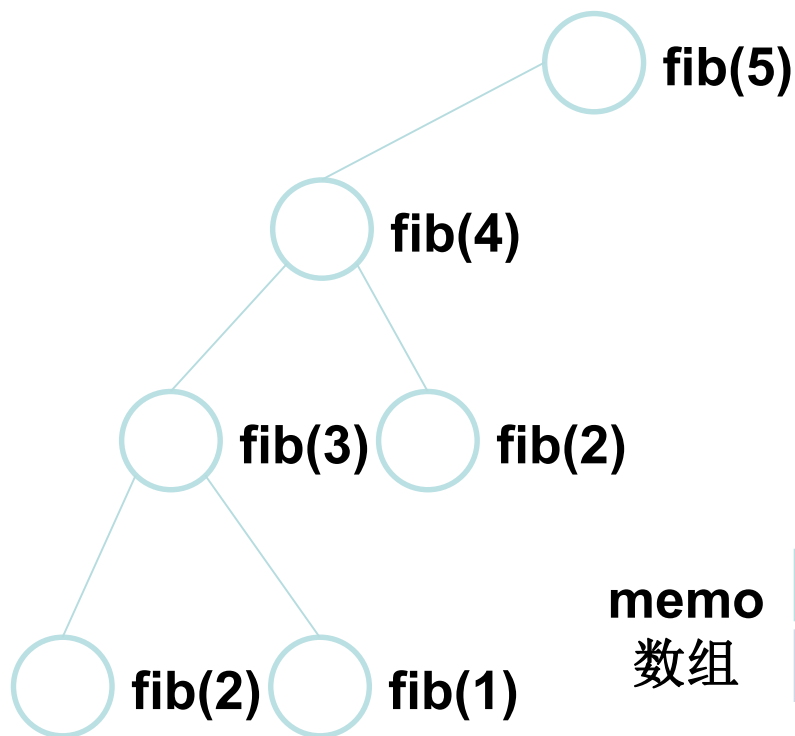


memo
数组

下标	1	2	3	4	5
取值	1	1	2	-1	-1

使用memo避免重复计算

- 使用数组存储memo[1]-memo[n]分别存储已经计算出的fib(1)到fib(n)的值
- 数组的初值设为-1,表示“未计算”

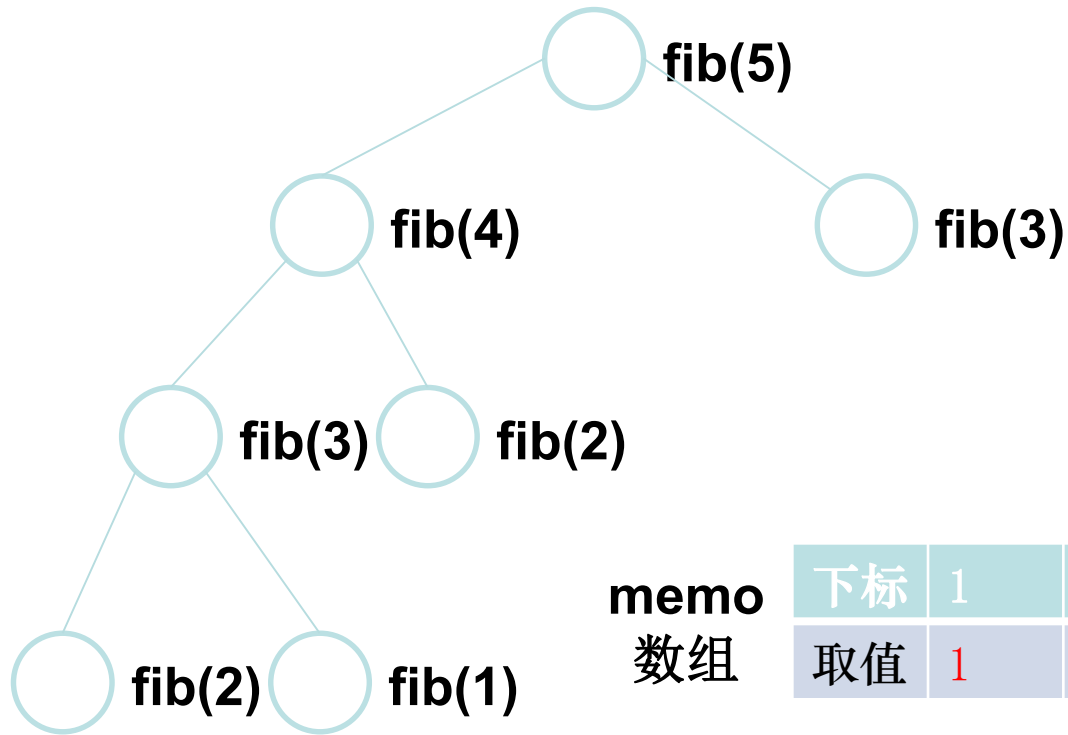


memo
数组

下标	1	2	3	4	5
取值	1	1	2	3	-1

使用memo避免重复计算

- 使用数组存储memo[1]-memo[n]分别存储已经计算出的fib(1)到fib(n)的值
- 数组的初值设为-1,表示“未计算”



memo
数组

下标	1	2	3	4	5
取值	1	1	2	3	5

使用memo避免重复计算

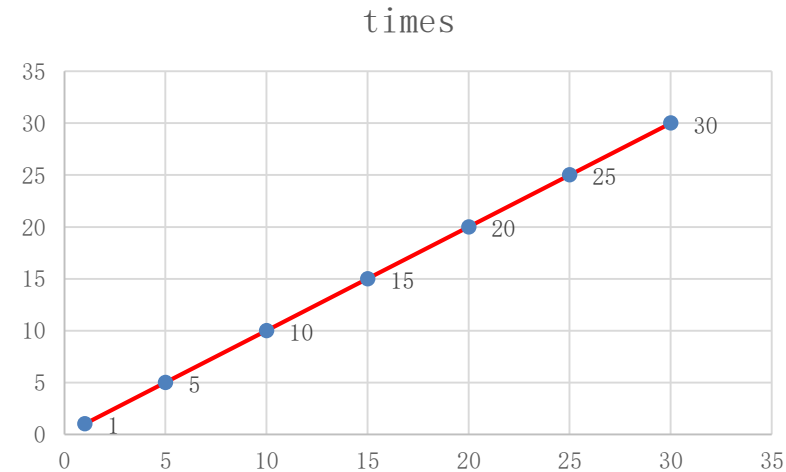
- 分析memo数组的作用
 - 将已经计算过的“中间结果”存起来
 - 直接使用存好的“中间结果”
 - 避免了重复计算

程度设计重要思想

用空间换时间

斐波那契数列

```
#include<iostream>
using namespace std;
int times = 0;
int fib(int n, int memo[]) {
    if (memo[n]!=-1)
        return memo[n];
    else{
        times ++;
        if (n==1||n==2) memo[n]=1;
        else memo[n]=fib(n-1,memo)+fib(n-2,memo);
        return memo[n];}
}
int main () {
    int memo[100];
    cout << fib(5,memo) << endl; for (int i = 0; i < 100; i ++){
memo[i]=-1;
```





中國人民大學
RENMIN UNIVERSITY OF CHINA

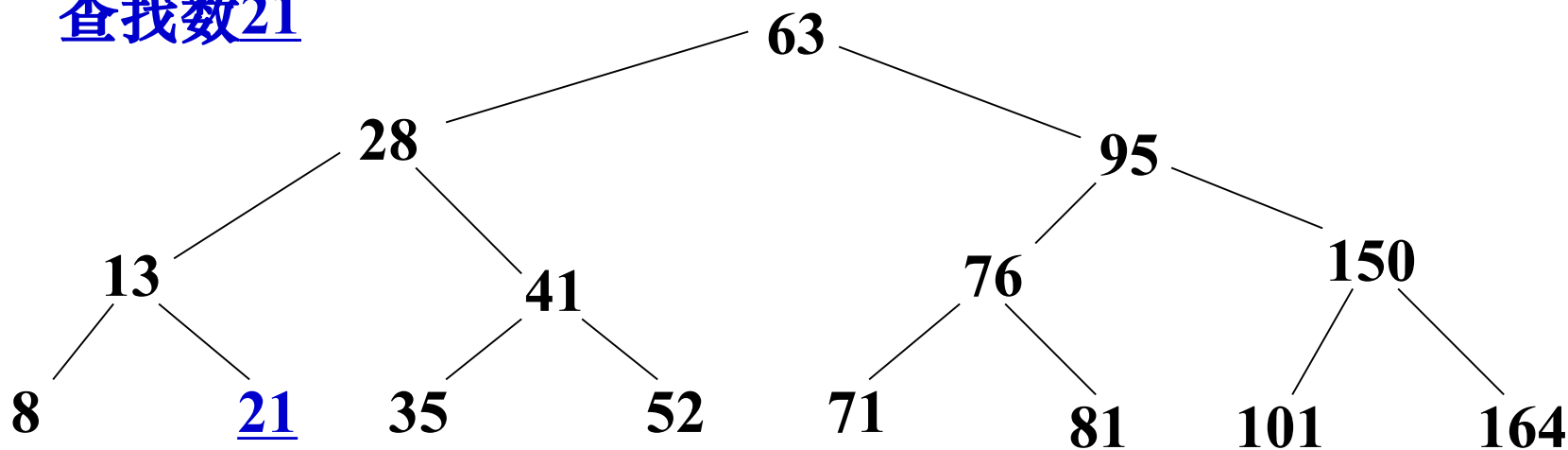


5. 查找排序与递归

二分查找

- 如果把排好序的数据看成一棵树.....

查找数21

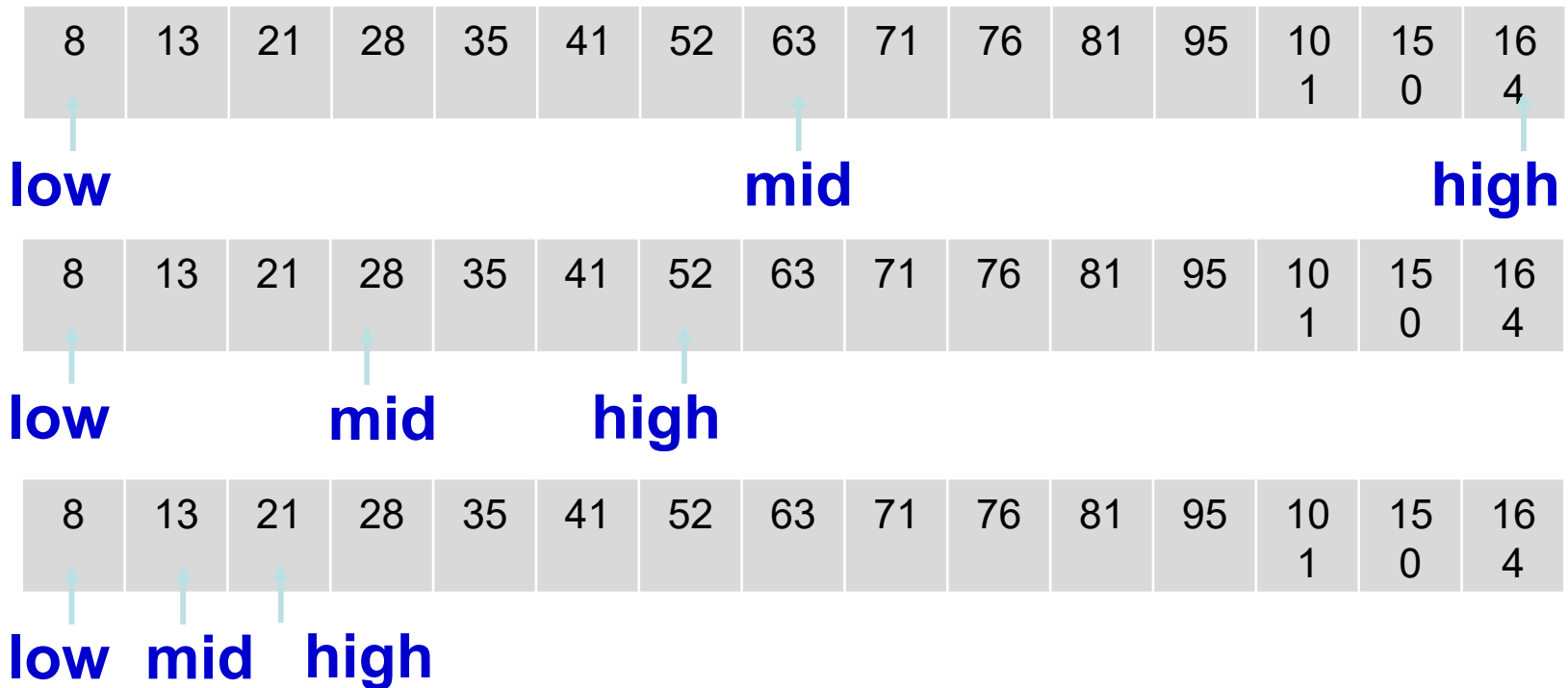


思想： 折半查找

二分查找的循环实现 (1)

- 核心难点：实现 “折半”

➤ 解决：设置下标变量low, high, mid



二分查找的循环实现 (2)

```
int bsearch(int ary[], int low, int high, int m) {  
    while( low <= high ) {  
        int mid = (low+high)/2;  
        if( ary[mid] == m ) //找到m  
            return mid;           //返回  
        else if( ary[mid] > m )   //在数组的左半边  
            high = mid-1;         //更新右边界high  
        else                     //在数组的右半边  
            low = mid + 1;        //更新左边界low  
    }  
    return -1; //没找到  
}
```

二分查找的循环实现 (3)

■ 边界条件

- 可能找到: $low \leq high$!
- 当 $low > high$ 时, 表明查找元素不存在

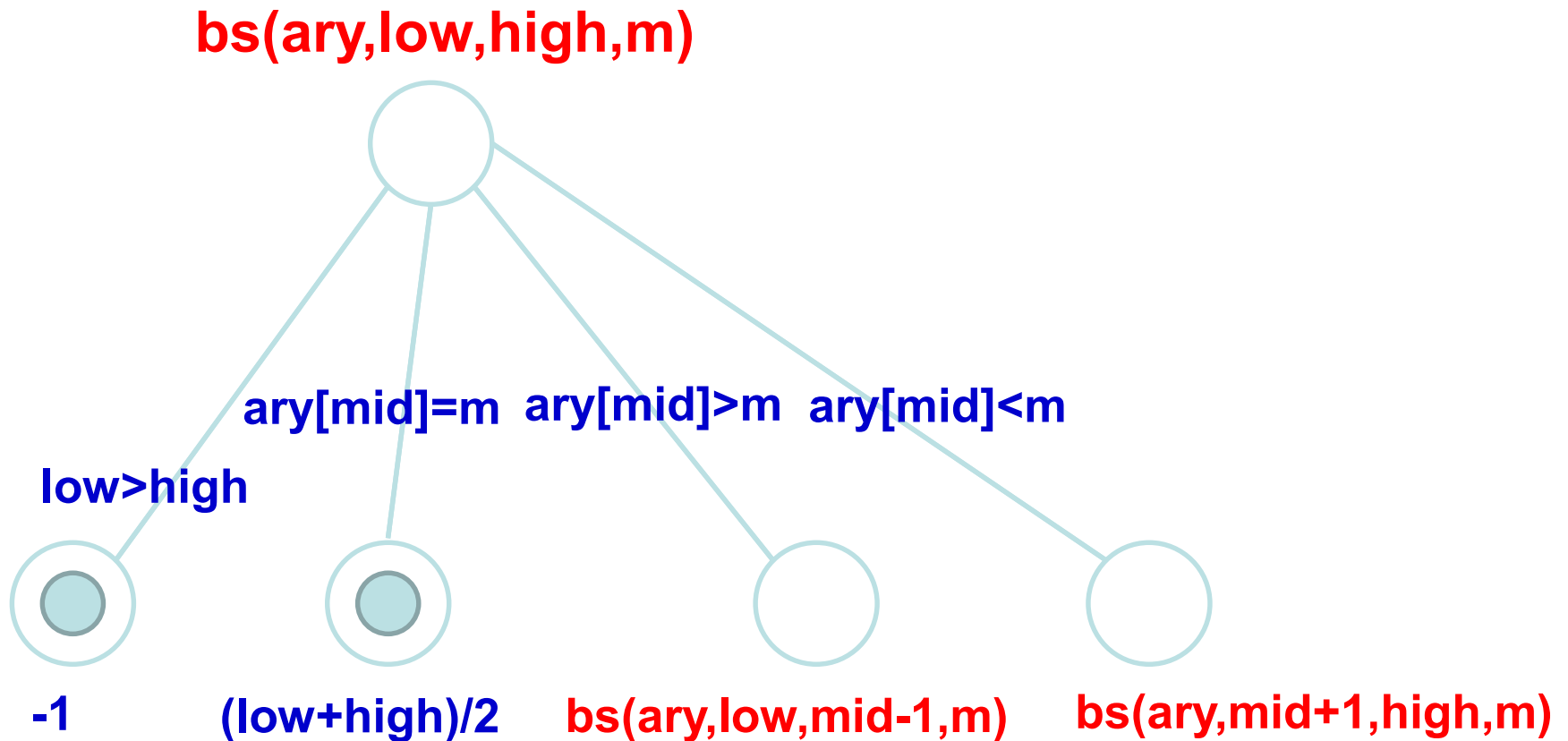
■ 实现“折半”的途径

- 下标操作: $mid = (low + high) / 2$

■ 扩展

- 如果找不到 m , 返回比 m 大的最小数下标
- 例如: 查找 34, 返回 35 的下标 4
- 修改最后一句: `return low`

二分查找的递归实现 (1)



二分查找的递归实现 (2)

```
int bisearch(int ary[], int low, int high, int m) {  
    if ( low > high )        //没找到  
        return -1;  
    int mid = (low + high) / 2;  
    if (ary[mid] == m) //找到  
        return mid;  
    else if ( ary[mid] > m ) //找左半边  
        return bisearch(ary, low, mid - 1, m);  
    else //找右半边  
        return bisearch(ary, mid + 1, high, m);  
}
```

Bisearch(二分查找).cpp

```

int bsearch(int ary[], int low, int high, int m) {
    while( low <= high ) {
        int mid = (low + high) / 2;
        if( ary[mid] == m ) //找到m
            return mid;          //返回
        else if( ary[mid] > m ) //在数组的左半边
            high = mid - 1;      //更新右边界high
        else //在数组的右半边
            low = mid + 1;      //更新左边界low
    }
    return -1; //没找到
}

```

循环方式

```

int bsearch(int ary[], int low, int high, int m) {
    if ( low > high ) //没找到
        return -1;
    int mid = (low + high) / 2;
    if ( ary[mid] == m ) //找到
        return mid;
    else if ( ary[mid] > m ) //找左半边
        return bsearch(ary, low, mid - 1, m);
    else //找右半边
        return bsearch(ary, mid + 1, high, m);
}

```

递归方式

排序问题

■ 问题定义

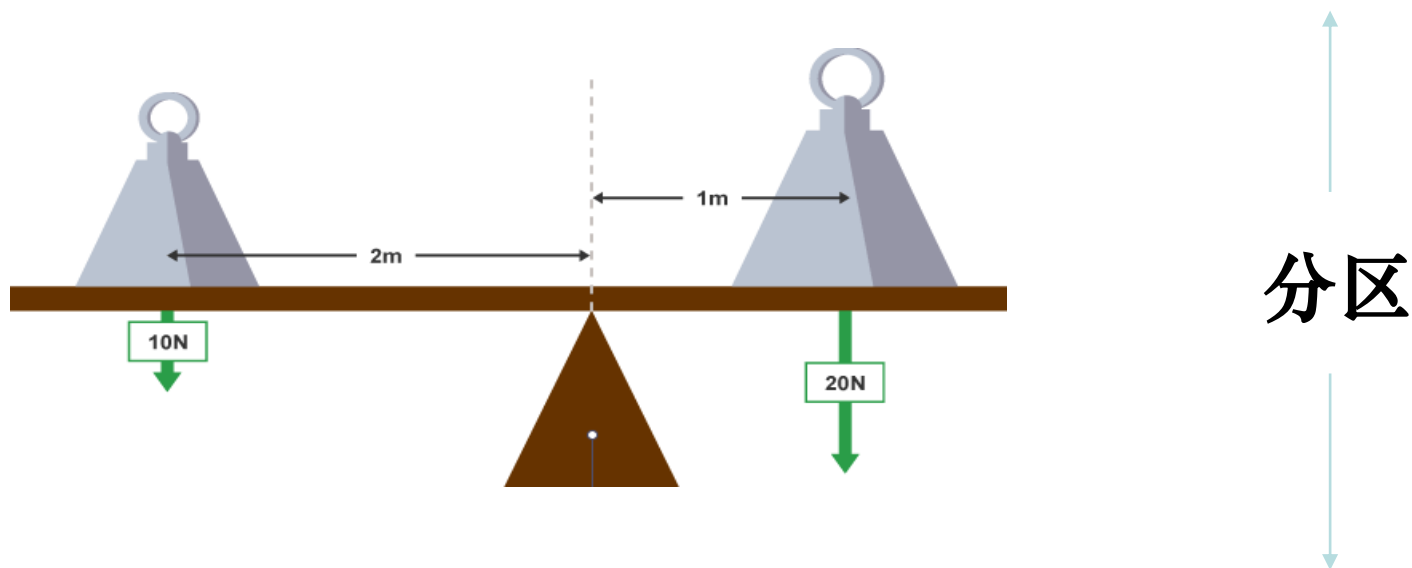
- 数据：给定一组乱序的数 {13, 8, 21, 28, 164, 35, 41, 52, 71, 63, 76, 95, 81, 101, 150}
- 操作：按照数组由小到大排序

■ 学过的排序算法

- 冒泡排序
- 选择排序

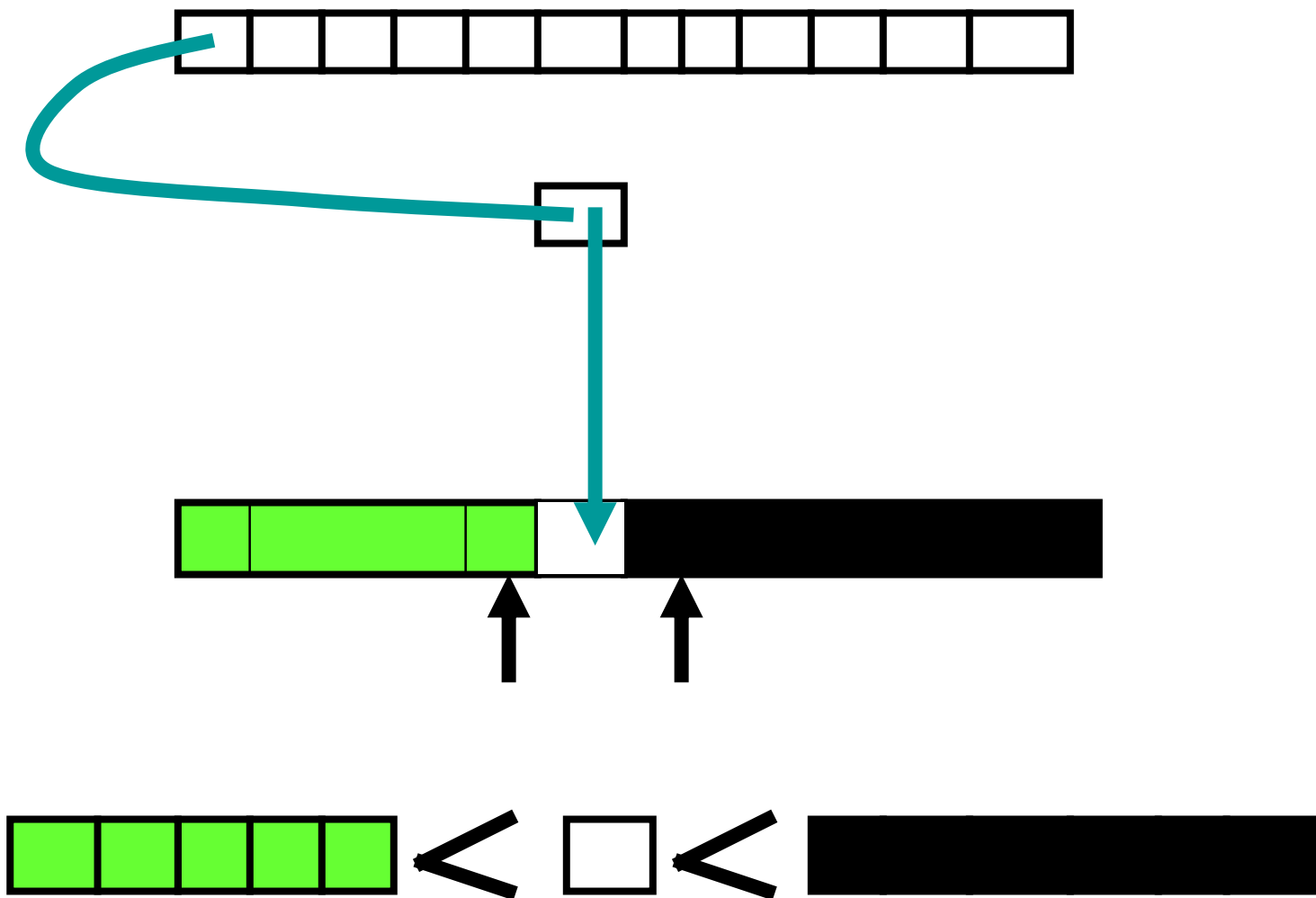
快速排序

1. **选枢纽**：从数组中选择一个元素，称之为枢纽pivot元素



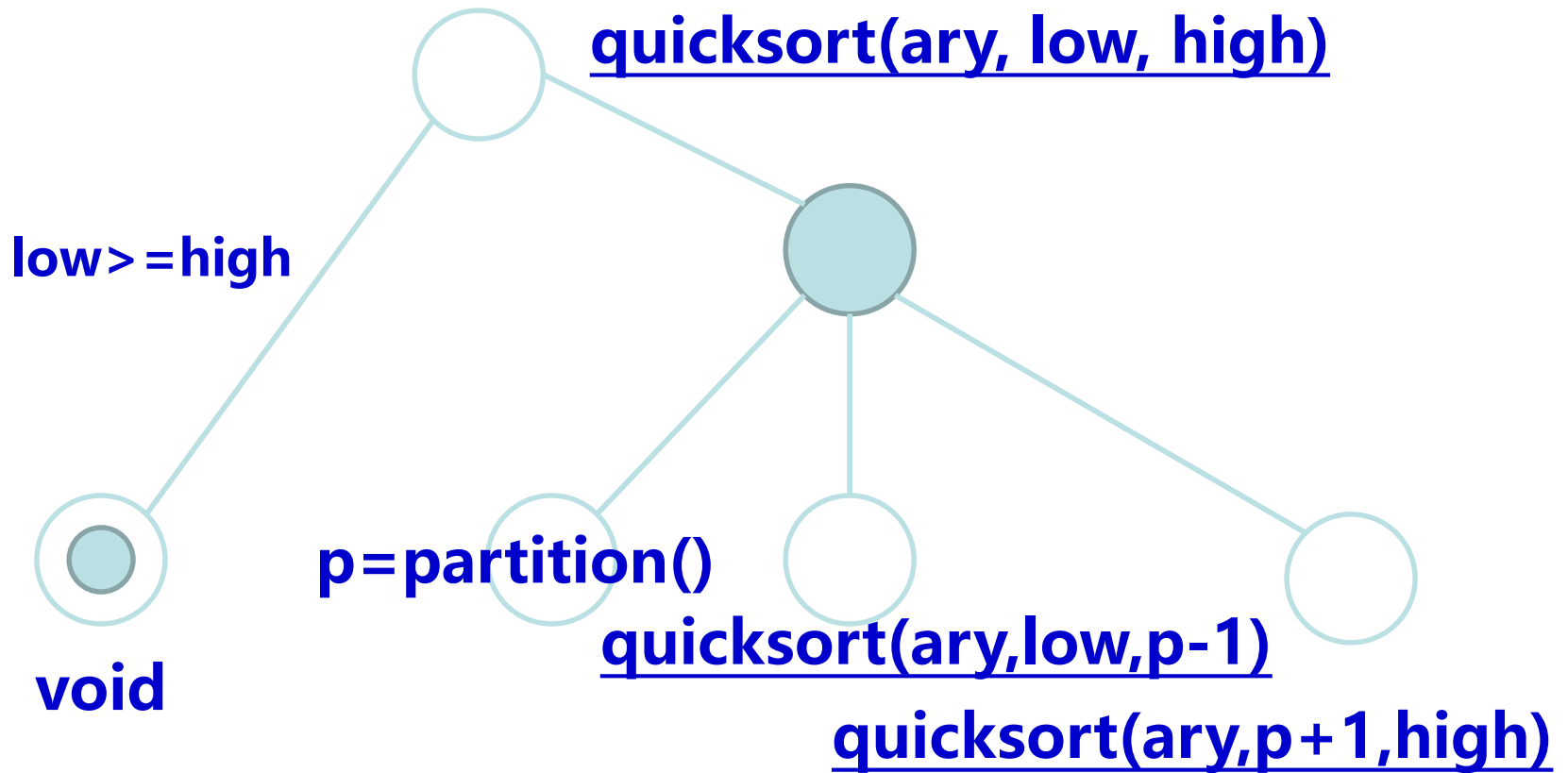
2. **重组织**：重新组织数组，使其满足比pivot小的在左侧，大的在右侧，以此分区。
3. **做递归**：分别对前后两部分执行上述步骤

快速排序基本思想



快速排序与或图设计

- 将问题抽象为quicksort (ary, low, high)



快速排序函数设计

```
void quicksort(int ary[], int low, int high) {  
    if (low >= high)    //无需做任何事  
        return;  
    //分区：找出pivot点，并重组织数组  
    int p = partition(ary, low, high);  
    //递归调用：处理左侧分区  
    quicksort(ary, low, p - 1);  
    //递归调用：处理右侧分区  
    quicksort(ary, p + 1, high);  
}
```

Quicksort(快排).cpp

快速排序partition函数设计 (1)

- **int partition(int ary[], int low, int high)**

- 核心要解决两个问题

1. 选择哪个元素作为pivot? 最左侧

5	2	6	1	7	3	4
0	1	2	3	4	5	6

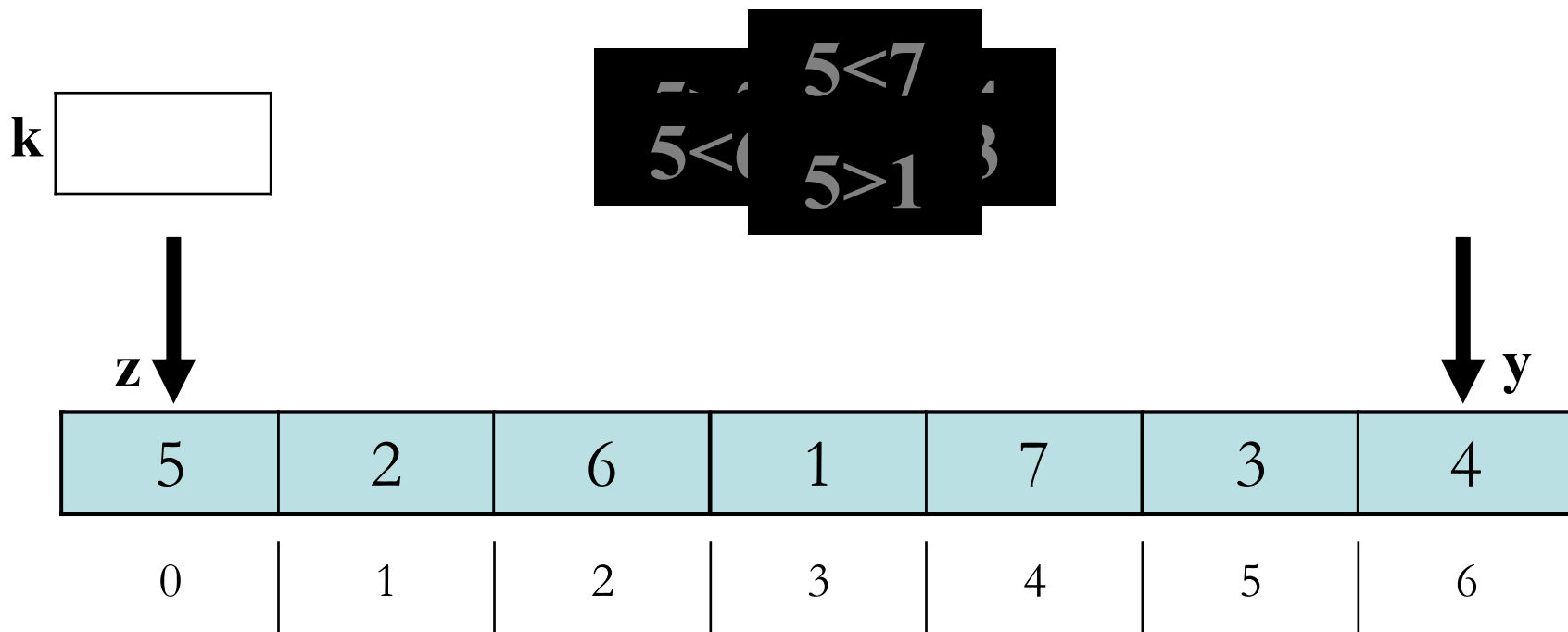
2. 如何基于pivot进行分区?

- 目标很明确: 比5小的放左边、大的放右边
- 你如何实现?

快速排序partition函数设计 (2)

■ 基本思路

- 设置下标变量 i ，从左往右扫描数组
- 设计下标变量 j ，从右往左扫描数组



快速排序partition函数设计 (3)

```
int partition(int ary[], int low, int high)
{
    int p = ary[low]; // left-most as pivot
    while( low < high ) {
        while( low < high && ary[high] >= p ) high--;
        ary[low] = ary[high];
        while( low < high && ary[low] <= p ) low++;
        ary[high] = ary[low];
    }
    ary[low] = p;
    return low;
}
```

如果是从大到小进行排序，应怎样修改？



中國人民大學
RENMIN UNIVERSITY OF CHINA



06. 利用递归思想解题

全排列问题

■ 问题定义

- 从数组中的 n 个元素中任取 m ($m \leq n$) 个元素, 按照一定的顺序排列起来, 叫做从 n 个不同元素中取出 m 个元素的一个排列。

■ 当 $m = n$ 时所有的排列情况叫全排列

■ 以字符元素为例, 如{A,B,C}的全排列:

A B C

B C A

A C B

C A B

B A C

C B A

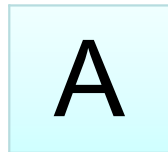
问题分析

■ 问题定义

- 给定字符数组`ary[n]`，输出数组元素的全排列

■ 简单情况：

- 数组只包含一个数，即 $n=1$
- 例如`char ary[n] = {A}`



直接输出：

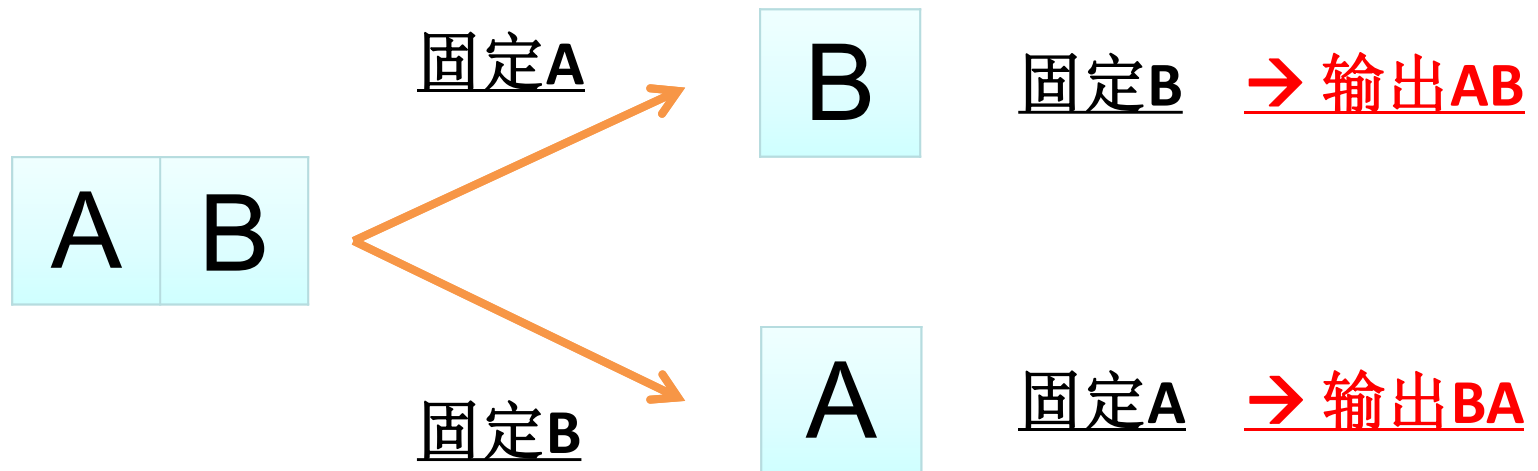
A

1. 注：为了表述方便，省略掉字符的单引号，下同。

问题分析

■ 略复杂情况：

- 数组包含2个数，即 $n=2$
- 例如`char ary[n] = {A, B}`

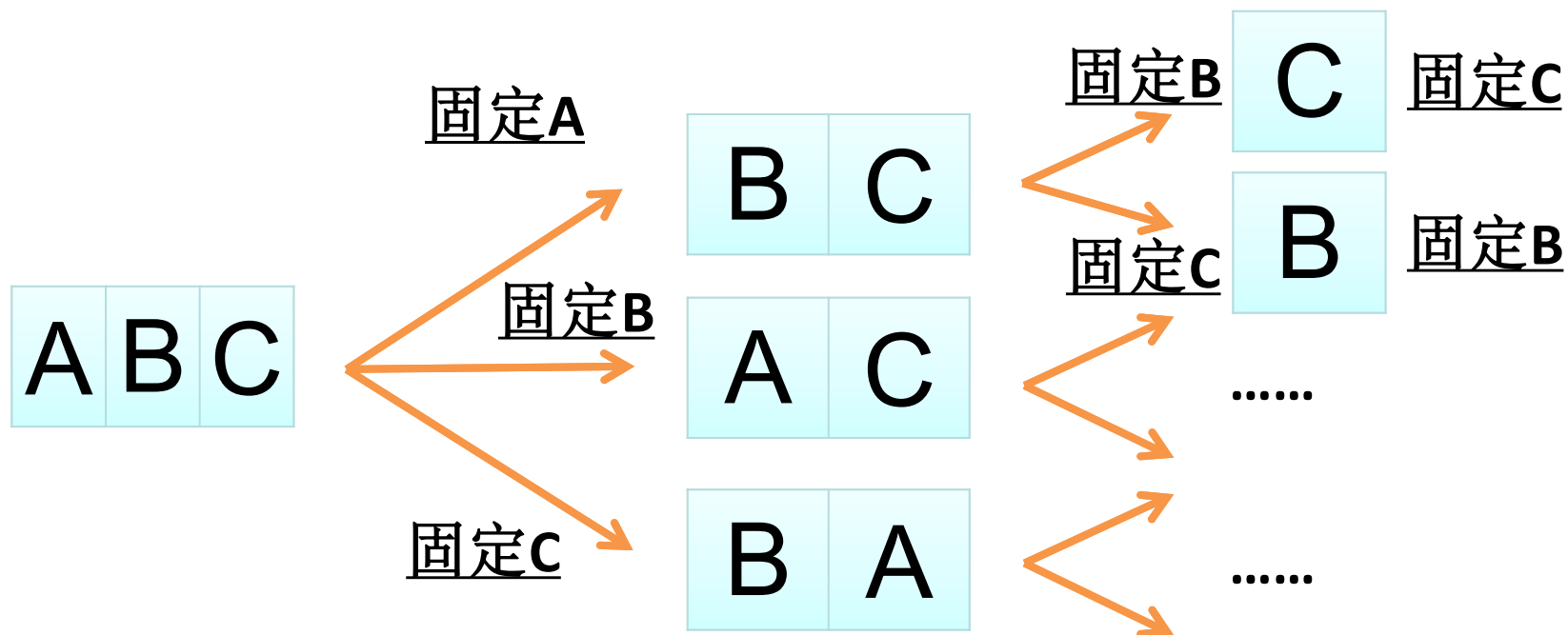


问题分析

■ 略复杂情况：

- 数组包含3个数，即 $n=3$
- 例如`char ary[n] = {A, B, C}`

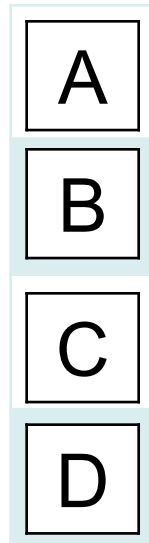
能否利用**递归**
进行描述解决



问题分析

- 引入selected[n]数组记录待输出的元素
 - 将已经“固定”的元素存储起来
 - 当selected数组放满n个元素时，找到一组全排列，将selected数组进行输出

数组
ary[n]



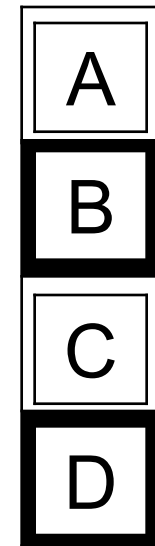
数组
selected [n]



利用递归解题

- 问题抽象：设计函数perm(s,n)表示输出ary数组从下标s到下标n-1的元素的全排列
- 分析递归的基本要素
 - Basic Case: selected 数组填满时，则输出

数组
ary[n]

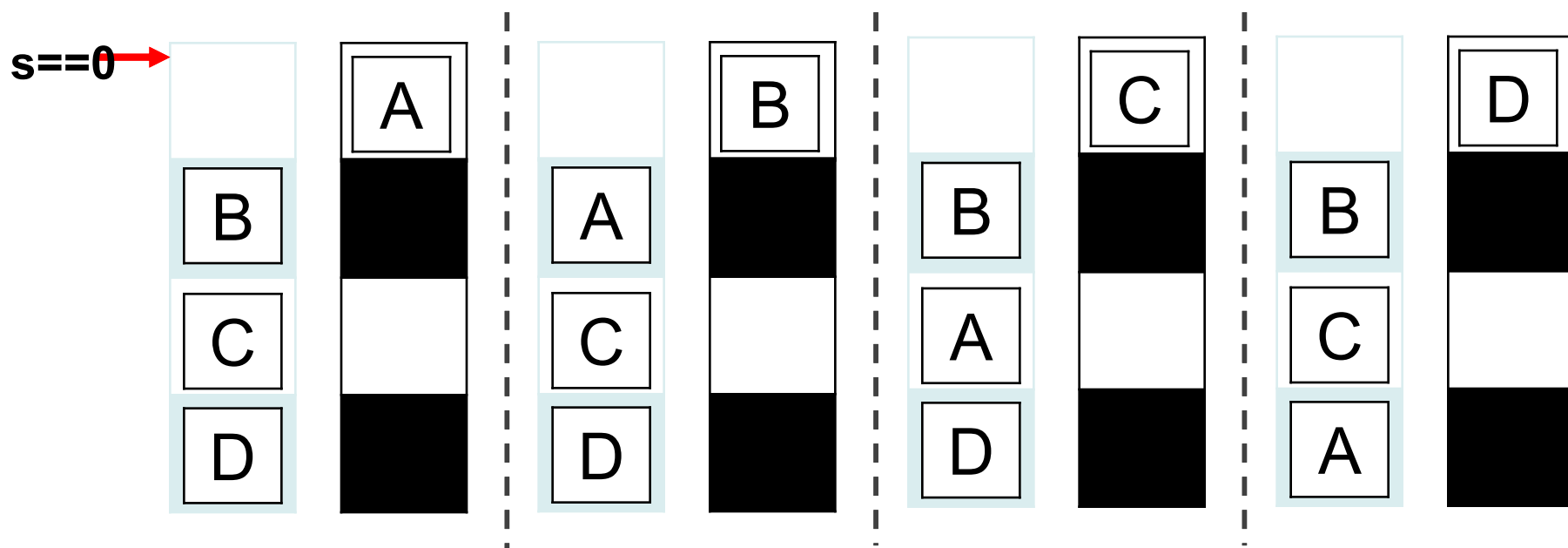


数组
selected [n]

利用递归解题

■ 分析递归的基本要素

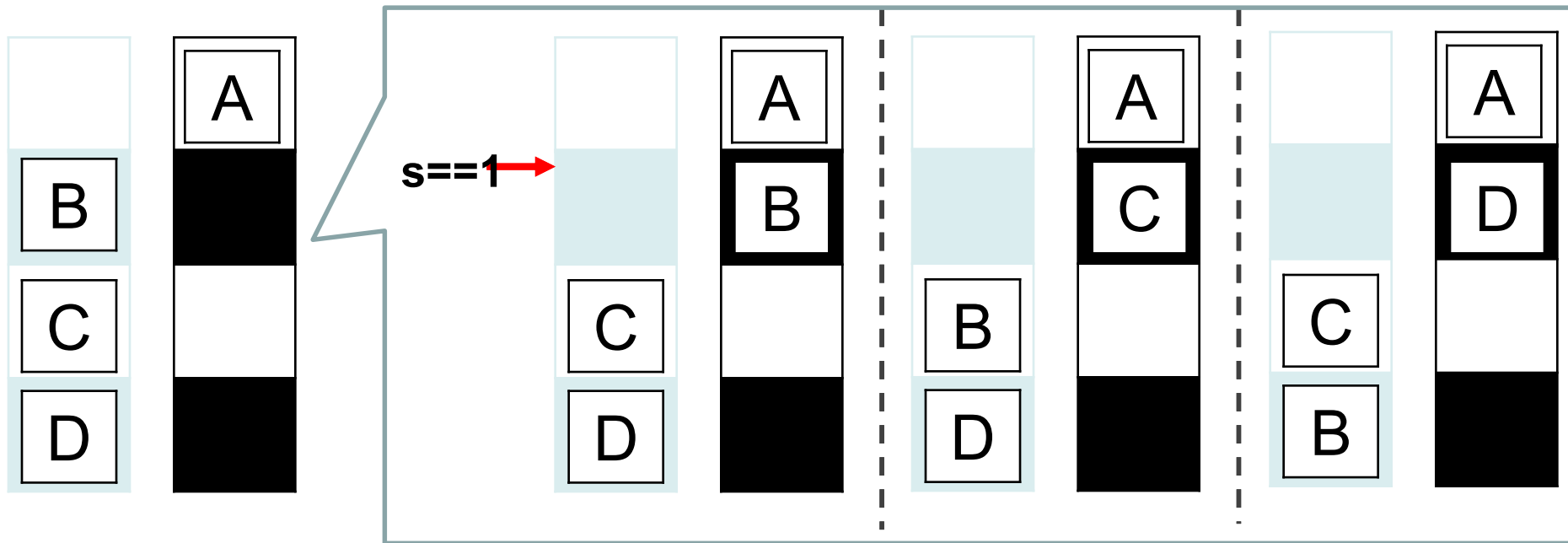
- Inductive Case: 当 $s < n$ 时
- 例：当 $s == 0$ 时，将第0个元素固定在selected数组中；递归调用子问题perm($s+1, n$)



利用递归解题

■ 分析递归的基本要素

- Inductive Case: 当 $s < n$ 时
- 例：当 $s = 1$ 时，将第1个元素固定在selected数组中；递归调用子问题 $\text{perm}(s+1, n)$



与或图:



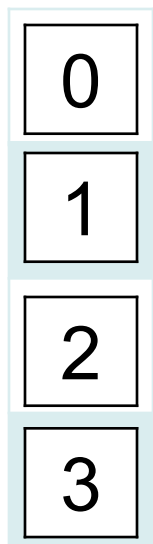
全排列解决方案1

```
void perm_impl1 (int ary[ ], int selected[ ], int k, int n) {  
    if (k == n) {  
        for (int i = 0; i < n; i++)  
            cout << selected[i] << " ";  
        cout << endl;  
    } else  
        for (int i = k; i < n; i++) {  
            selected[k] = ary[i];  
            swap (ary, k, i);  
            perm_impl1(ary, selected, k+1, n);  
            swap (ary, k, i);  
        }  
}
```

运行结果

1	2	3	4	5
1	2	3	5	4
1	2	4	3	5
1	2	4	5	3
1	2	5	4	3
1	2	5	3	4
1	3	2	4	5
1	3	2	5	4
1	3	4	2	5
1	3	4	5	2
1	3	5	4	2
1	3	5	2	4
1	4	3	2	5
1	4	3	5	2
1	4	2	3	5
1	4	2	5	3
1	4	5	2	3
1	4	5	3	2
1	5	3	4	2

Original array



selected



输出0123全排列的部分过程

Print:

0 1 2 3

0 1 3 2

perm(0,4);

perm(1,4);

perm(2,4);

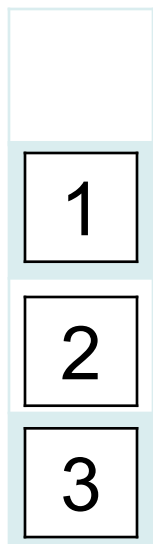
perm(3,4);

perm(3,4);

框表示正在运行的函数

为了清晰起见，将以确定、准备输出的数放在右边数组中，未确定的数放在左边数组中

Original array



selected



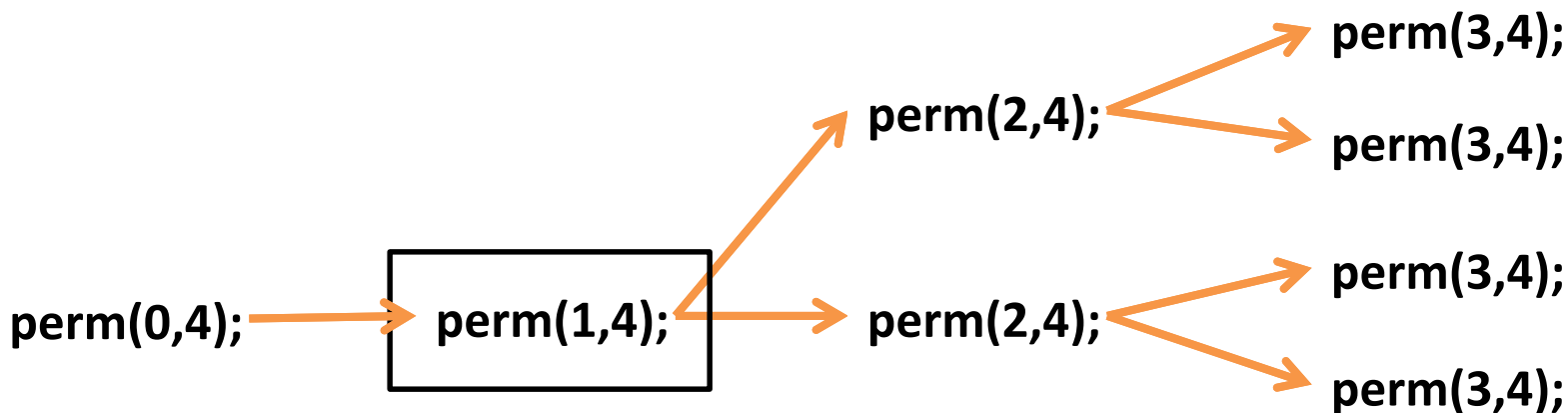
Print:

0 1 2 3

0 1 3 2

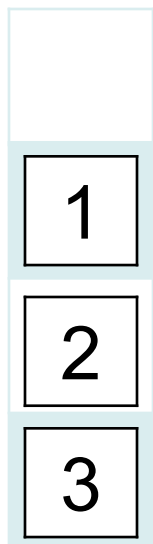
0 2 1 3

0 2 3 1



框中的是正在运行的函数

Original array



selected

Print:

0 1 2 3

0 1 3 2

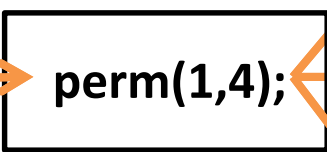
0 2 1 3

0 2 3 1

0 3 2 1

0 3 1 2

perm(0,4);



perm(1,4);

perm(2,4);

perm(2,4);

perm(2,4);

perm(3,4);

perm(3,4);

perm(3,4);

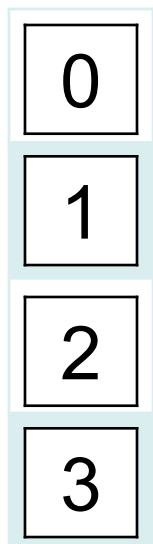
perm(3,4);

perm(3,4);

perm(3,4);

框中的是正在运行的函数

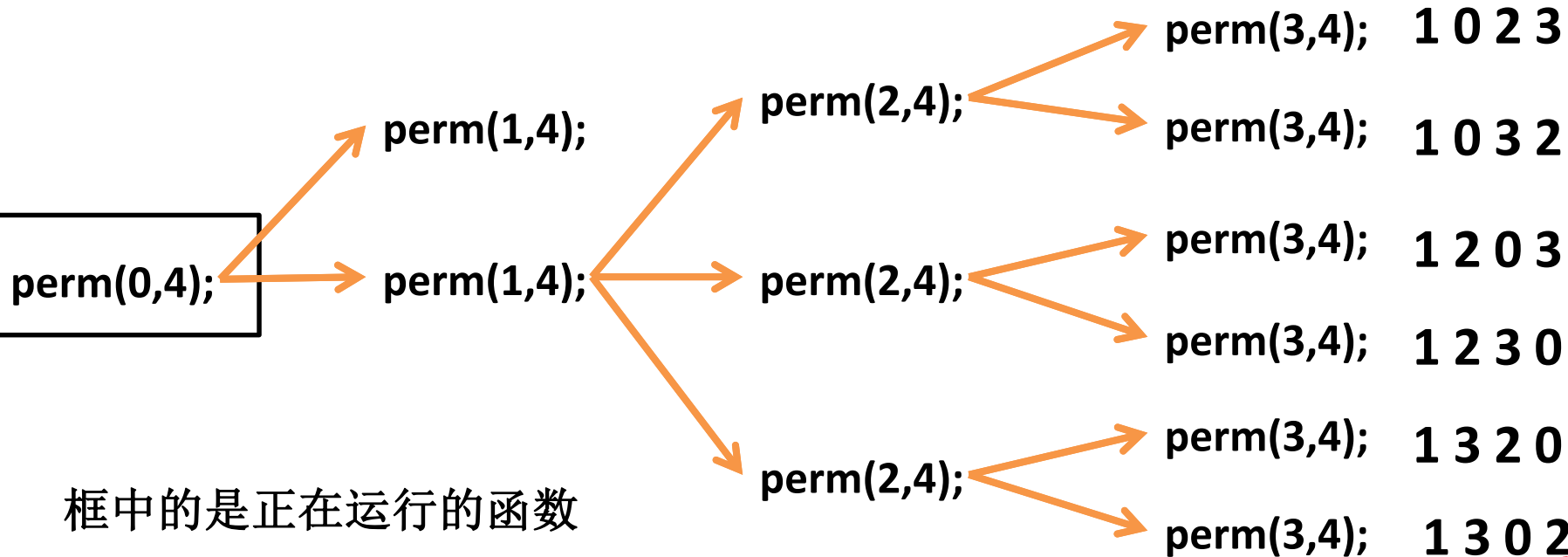
Unused number



selected

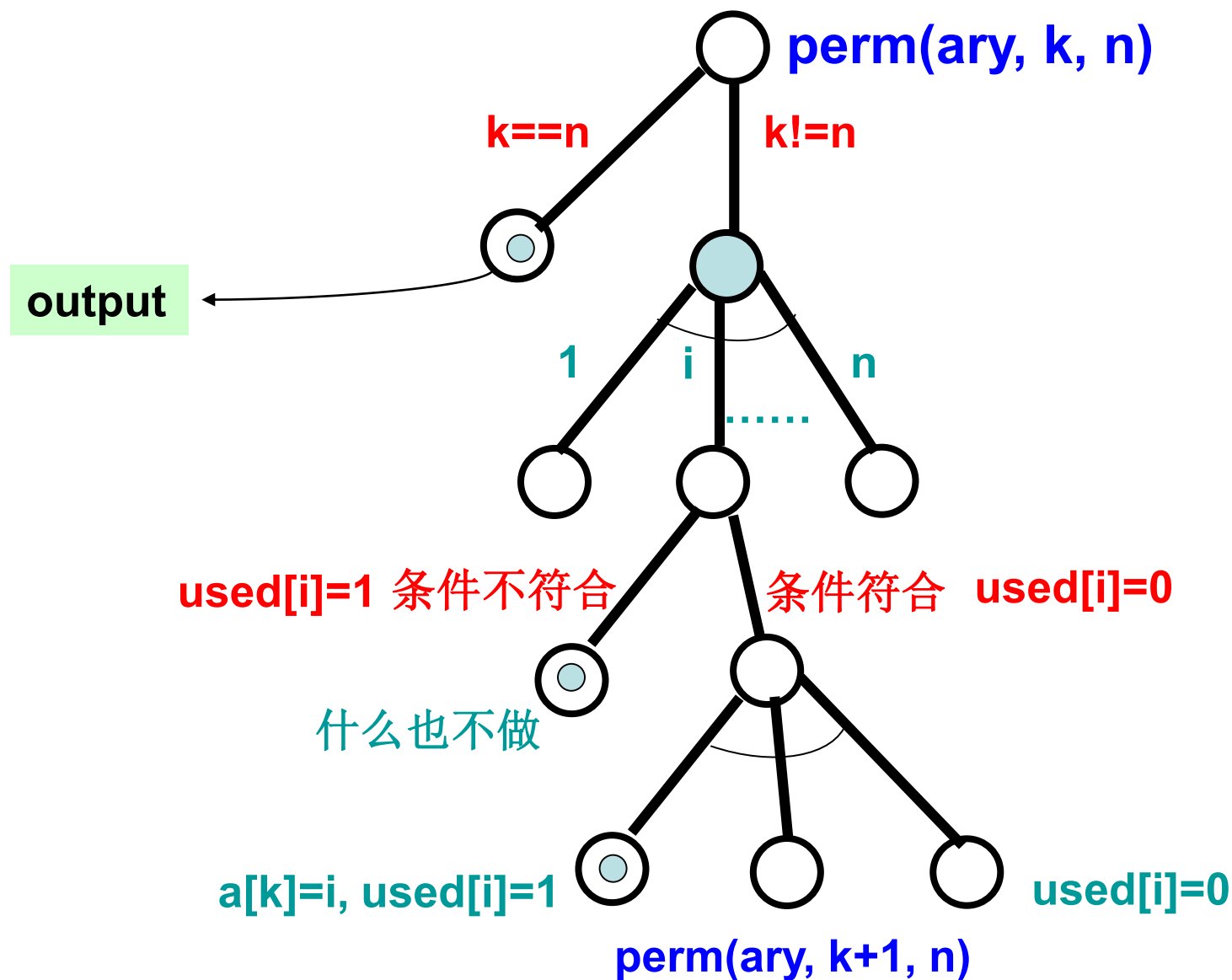


Print:



框中的是正在运行的函数

全排列解决方案2



全排列解决方案2

```
void perm_impl2 (int ary[ ], int indices[ ], int used[ ], int k, int n) {  
    if (k == n) {  
        for (int i = 0; i < n; i++)  
            cout << ary[indices[i]] << " ";  
        cout << endl;  
    }  
    else  
        for (int i = 0; i < n; i++) {  
            if (used[i] == 1) continue;  
  
            used[i] = 1;  
            indices[k] = i;  
            perm_impl2(ary, indices, used, k+1, n);  
            used[i] = 0;  
        }  
}
```

运行结果

1	2	3	4	5
1	2	3	5	4
1	2	4	3	5
1	2	4	5	3
1	2	5	4	3
1	2	5	3	4
1	3	2	4	5
1	3	2	5	4
1	3	4	2	5
1	3	4	5	2
1	3	5	4	2
1	3	5	2	4
1	4	3	2	5
1	4	3	5	2
1	4	2	3	5
1	4	2	5	3
1	4	5	2	3
1	4	5	3	2
1	5	3	4	2

#109 全排列问题.cpp



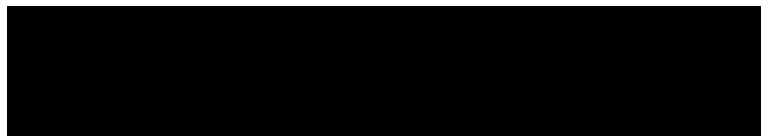
中國人民大學
RENMIN UNIVERSITY OF CHINA



07. 递归搜索问题

分书问题

- 有编号分别为 0, 1, 2, 3, 4 的五本书, 准备分给 A, B, C, D, E 五个人, 每个人阅读兴趣用一个二维数组加以描述:



希望你写一个程序, 输出所有分书方案, 让人人皆大欢喜。

- 假定 5 个人对 5 本书的阅读兴趣如下表：

		书				
		0	1	2	3	4
人	A	0	0	1	1	0
	B	1	1	0	0	1
	C	0	1	1	0	1
	D	0	0	0	1	0
	E	0	1	0	0	1

解题思路

- 1、定义一个整型的二维数组，将表中的阅读喜好用初始化方法赋给这个二维数组。可定义

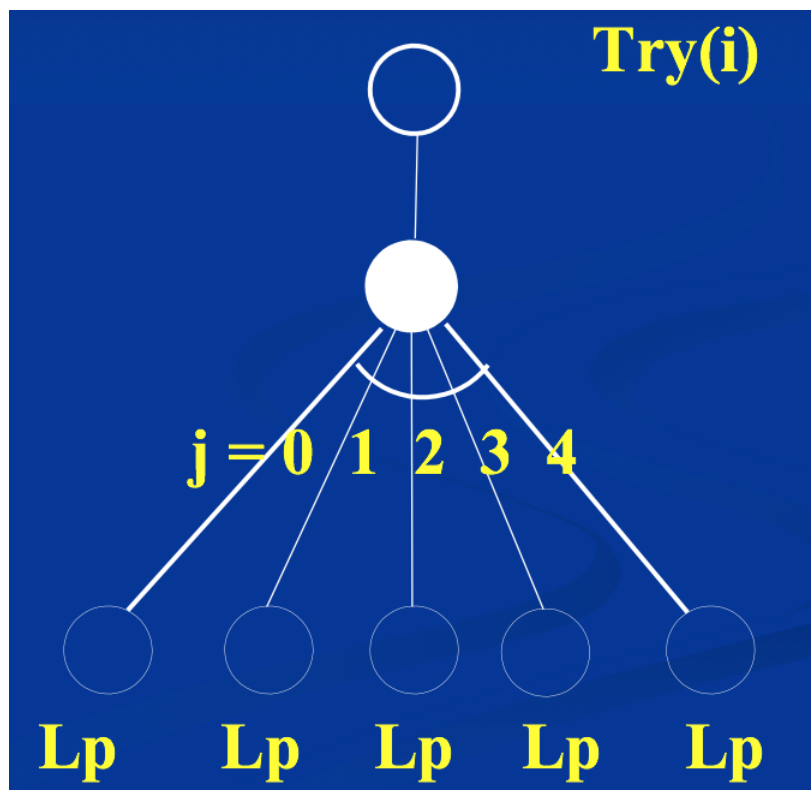
```
int like[5][5] = { {0,0,1,1,0} , {1,1,0,0,1} ,  
                  {0,1,1,0,1} , {0,0,0,1,0} , {0,1,0,0,1} };
```

- 2、定义一个整型一维数组book[5]用来记录书是否已被选用。用下标作为五本书的标号，被选过元素值为1，未被选过元素值为0，初始化皆置0。

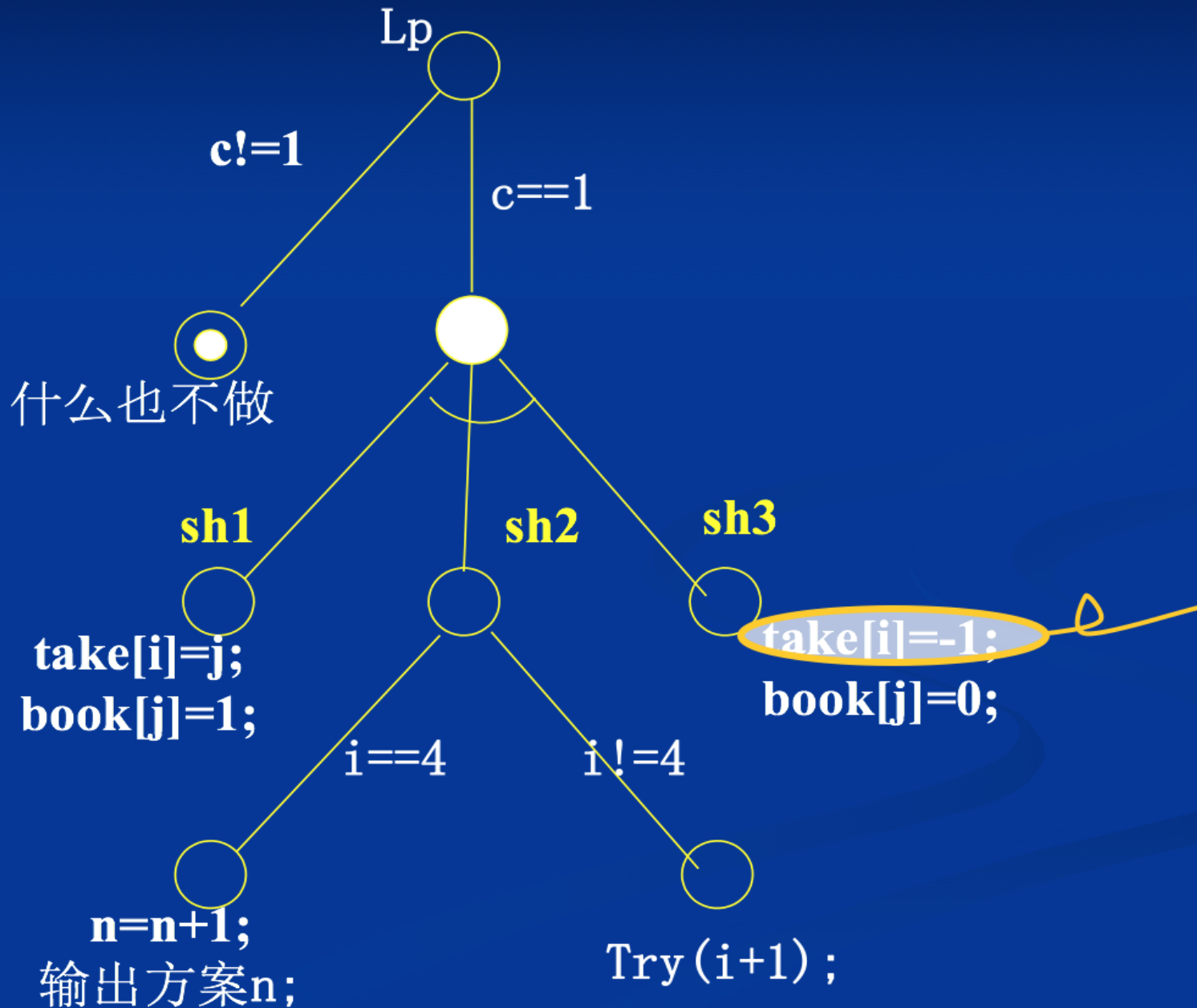
```
int book[5]={ 0,0,0,0,0 };
```

解题思路

- 定义函数 $\text{Try}(i)$ ，表示给第 i 个人分书
- 画出与或图



条件 $C=(like[i][j]>0)\ \&\&\ (book[j]==0)$



条件 C 是由两部分 “与” 起来的。“第 i 个人喜欢 j 书，且 j 书尚未被分走”。满足这条件是第 i 人能够得到 j 书的条件。

- 如果不满足 C 条件，则什么也不做，这是直接可解结点。
- 满足 C 条件，做三件事。

第一件事：

将 j 书分给 i ，用一个数组 $take[i]=j$ ，

记住书 j 给了 i ，

同时记录 j 书已被选用， $book[j] = 1$ 。

第二件事：

查看 i 是否为 4，如果不为 4，表示尚未将所有 5 个人所要的书分完，这时应递归再试下一人，即 $\text{Try}(i+1)$ 。

如果 $i == 4$ ，则应先使方案数 $n = n+1$ ，然后输出第 n 个方案下的每个人所得之书。

第三件事：回溯

让第 i 人退回 j 书，恢复 j 书尚未被选的标志，即 $\text{book}[j] = 0$ 。这是在已输出第 n 个方案之后，去寻找下一个分书方案所必需的

回溯法Backtracking

- Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons each partial candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

大胆往前走，错了就回头

根据与或图写程序

```
#include<iostream>
using namespace std;
#define PNUM 5
#define BNUM 5

int take[5],n;
int like[5][5] = {{0,0,1,1,0},{1,1,0,0,1},
    {0,1,1,0,1}, {0,0,0,1,0}, {0,1,0,0,1} };
int book[5]={0,0,0,0,0};

void Try(int i);

int main () {
    Try (0);
}
```



```
void Try(int i) {
```

```
    for (int j = 0; j < BNUM; j++) { // for each book
```

```
        if (book[j] == 1) continue; // already taken
```

```
        if (like[i][j] == 0) continue; // not like
```

```
        take[i] = j; // take the book
```

```
        book[j] = 1; // update the flag
```

```
    if (i < PNUM -1 )
```

```
        Try(i+1);
```

```
    else {
```

```
        n++;
```

```
        cout << "Assignment Plan #" << n << endl;
```

```
        for(int k=0; k < PNUM; k++)
```

```
            cout << "Person " << char(k+'A') << " takes Book " << take[k] << endl;
```

```
        cout << endl;
```

```
    }
```

```
    take[i] = -1; // return the book
```

```
    book[j] = 0; // update the flag
```

```
}
```

```
}
```

边搜索，边判断

```

void Try(int i) {

    if (i == PNUM) {
        n++;
        cout << "Assignment Plan #" << n << endl;
        for(int k=0; k < PNUM; k++)
            cout << "Person " << char(k+'A') << " takes Book " << take[k] << endl;
        cout << endl;
    } else

        for (int j = 0; j < BNUM; j++) { // for each book
            if (book[j] == 1) continue; // already taken
            if (like[i][j] == 0) continue; // not like

            take[i] = j; // take the book
            book[j] = 1; // update the flag
            Try(i+1);
            take[i] = -1; // return the book
            book[j] = 0; // update the flag

        }

}

```

先Base, 后搜索

#202 分书问题.cpp

八皇后问题

- 在 8×8 的棋盘上，放置8个皇后（棋子），使两两之间互不攻击。所谓互不攻击是说任何两个皇后都要满足：
 - (1) 不在棋盘的同一行；
 - (2) 不在棋盘的同一列；
 - (3) 不在棋盘的同一对角线上。

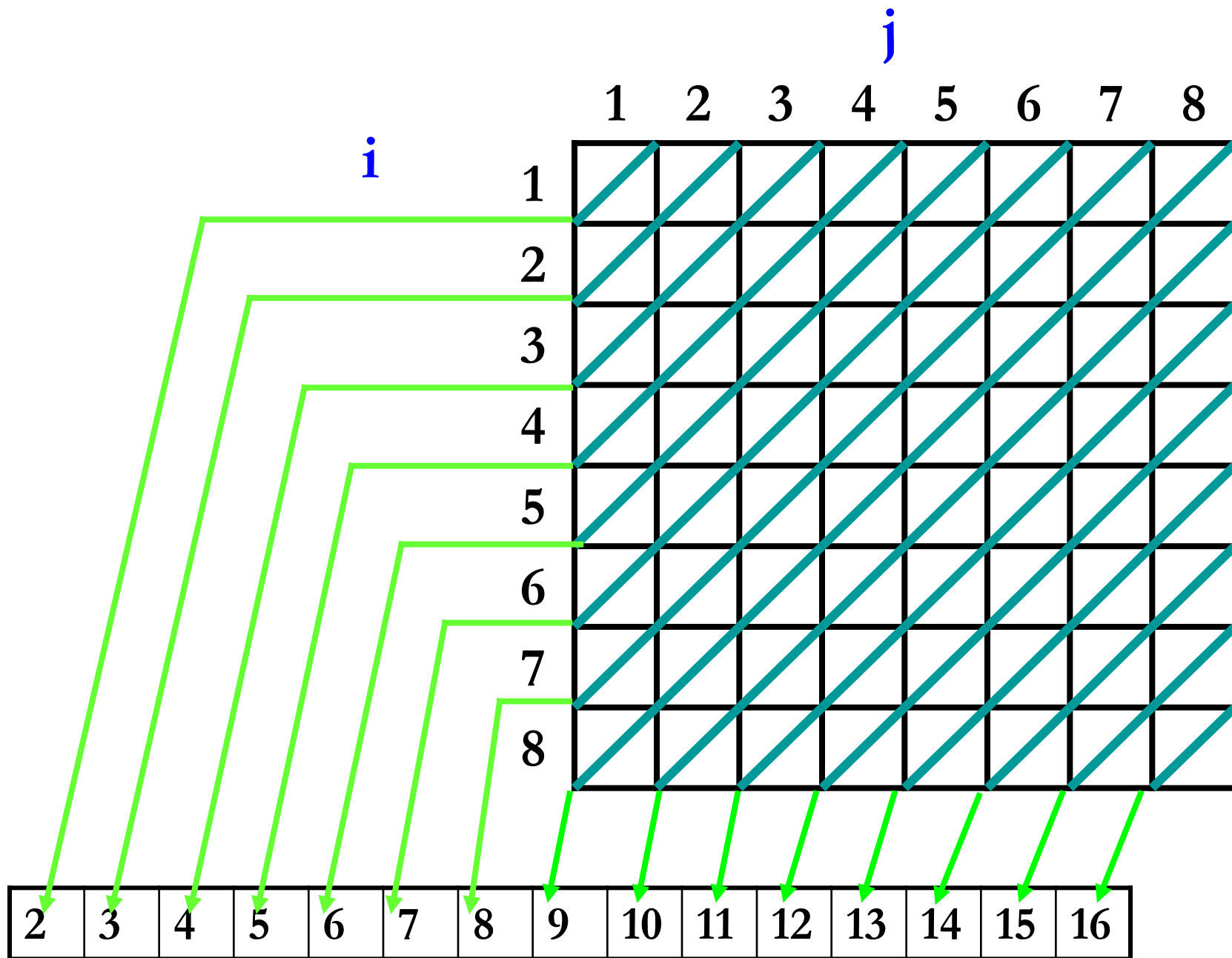
因此可以推论出，棋盘共有8行，每行有且仅有一个皇后，故至多有8个皇后。

这8个皇后每个应该放在哪一列上是解该题的任务。

解题思路 —— 回溯法

- 我们还是用试探的方法“向前走，碰壁回头”的策略
- 1、定义Try(i)——试探放第 i 行上的皇后。
- 2、讨论将第 i 行上的皇后放在 j 列位置上的安全性。
 - 我们可以逐行地放每一个皇后，因此，在做这一步时，假定第 i 行上还没有皇后，不会在行上遭到其它皇后的攻击。只考虑来自列和对角线的攻击。我们定义 $q(i) = j$ 表示第 i 行上的皇后放在第 j 列，一旦这样做了，就要考虑第 i 个皇后所在的列不安全了，这时让 $C[j] = 0$ ，同时，要考虑通过 (i, j) 位置的两条对角线也不安全了。分析看出从左上到右下的对角线上的每个位置都有 $i - j = \text{常数}$ 的特点；从左下到右上的对角线上的每个位置都有 $i + j = \text{常数}$ 的特点。比如两条对角线上的点，一条有 $i - j = -1$ ，一条有 $i + j = 7$ 的特点。

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



- 定义 $\text{int } R[17]$

$R[k] \quad k = 2, 3, \dots, 16$

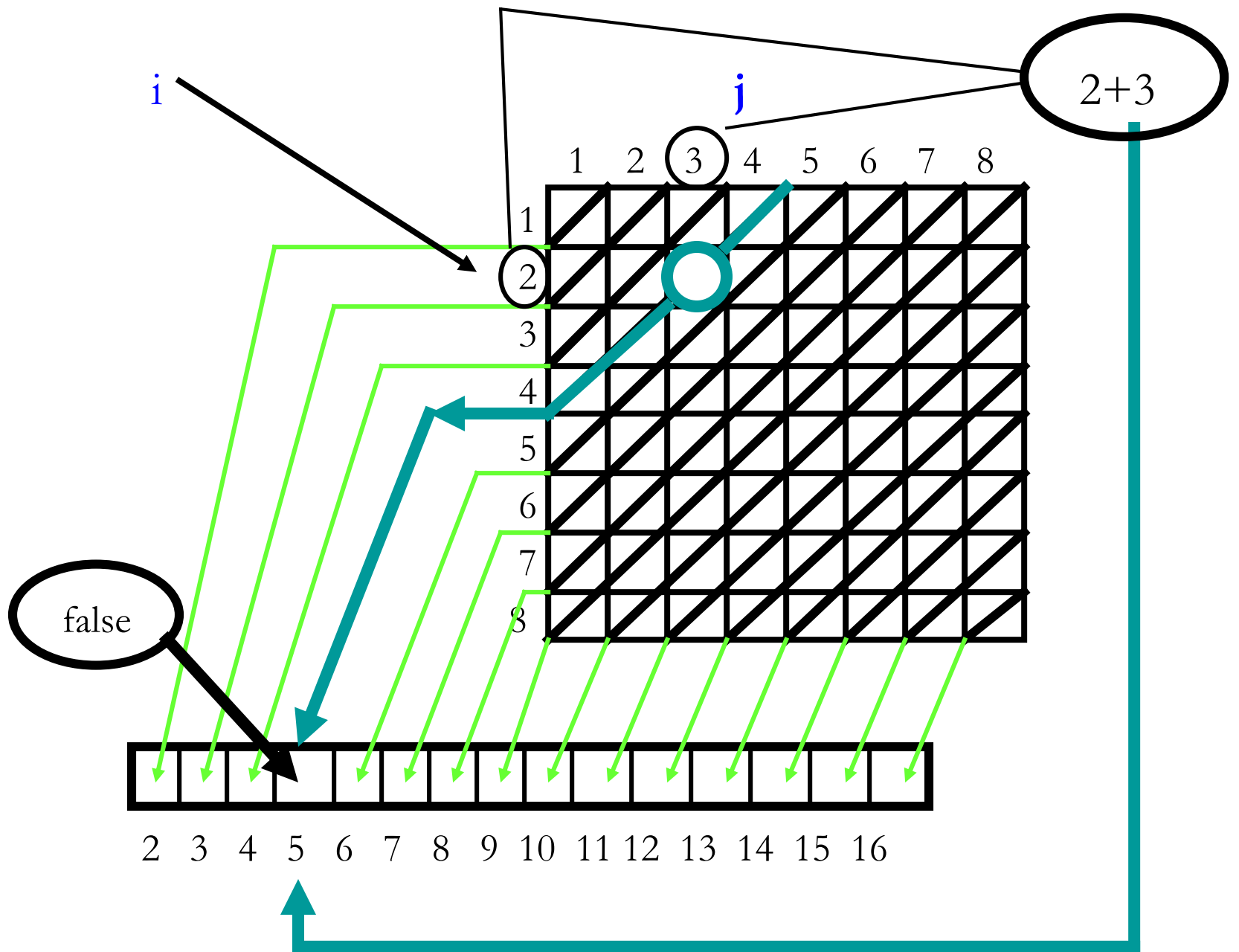
$$k = i + j$$

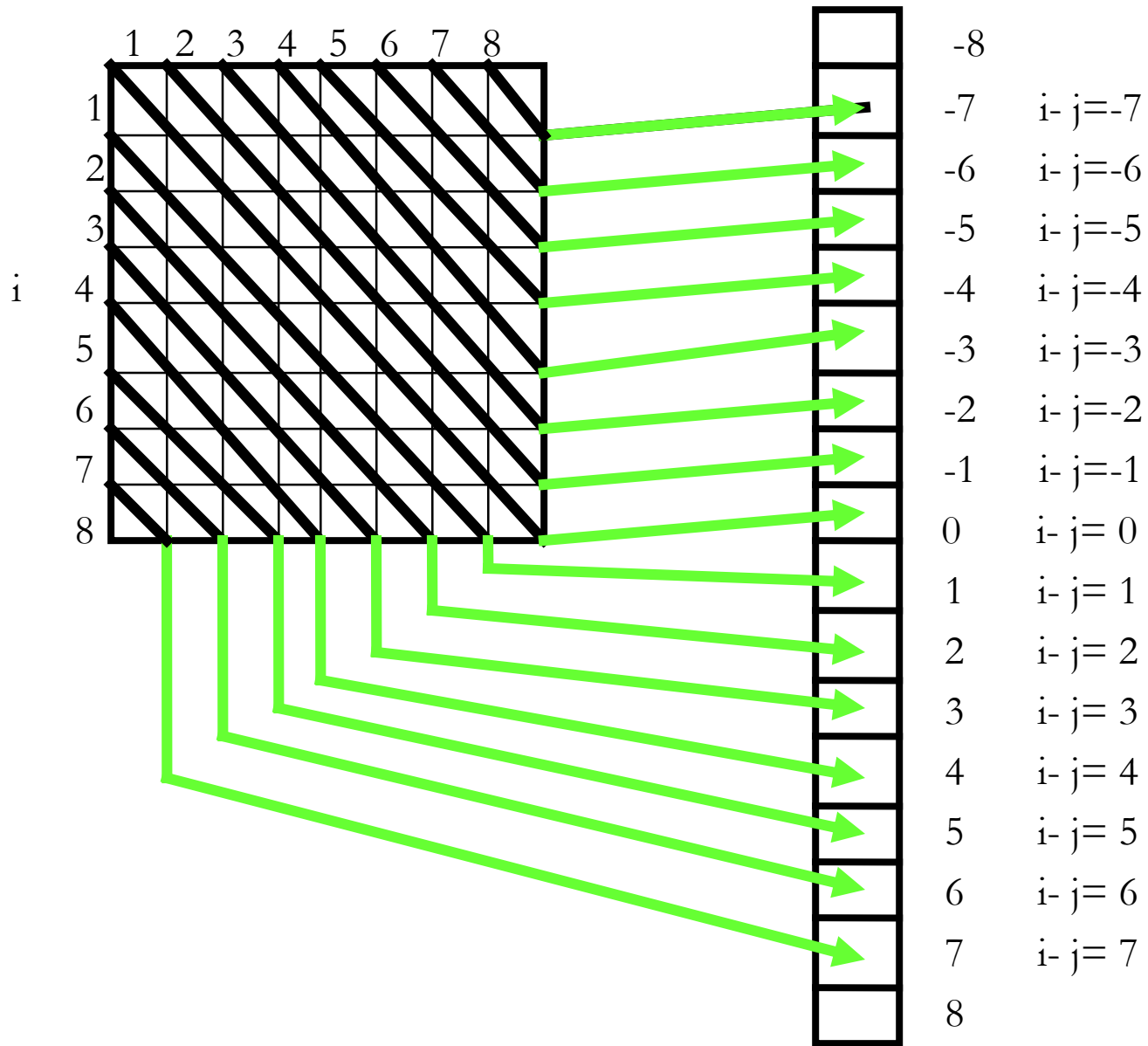
$$i = 1, 2, \dots, 8 \quad j = 1, 2, \dots, 8$$

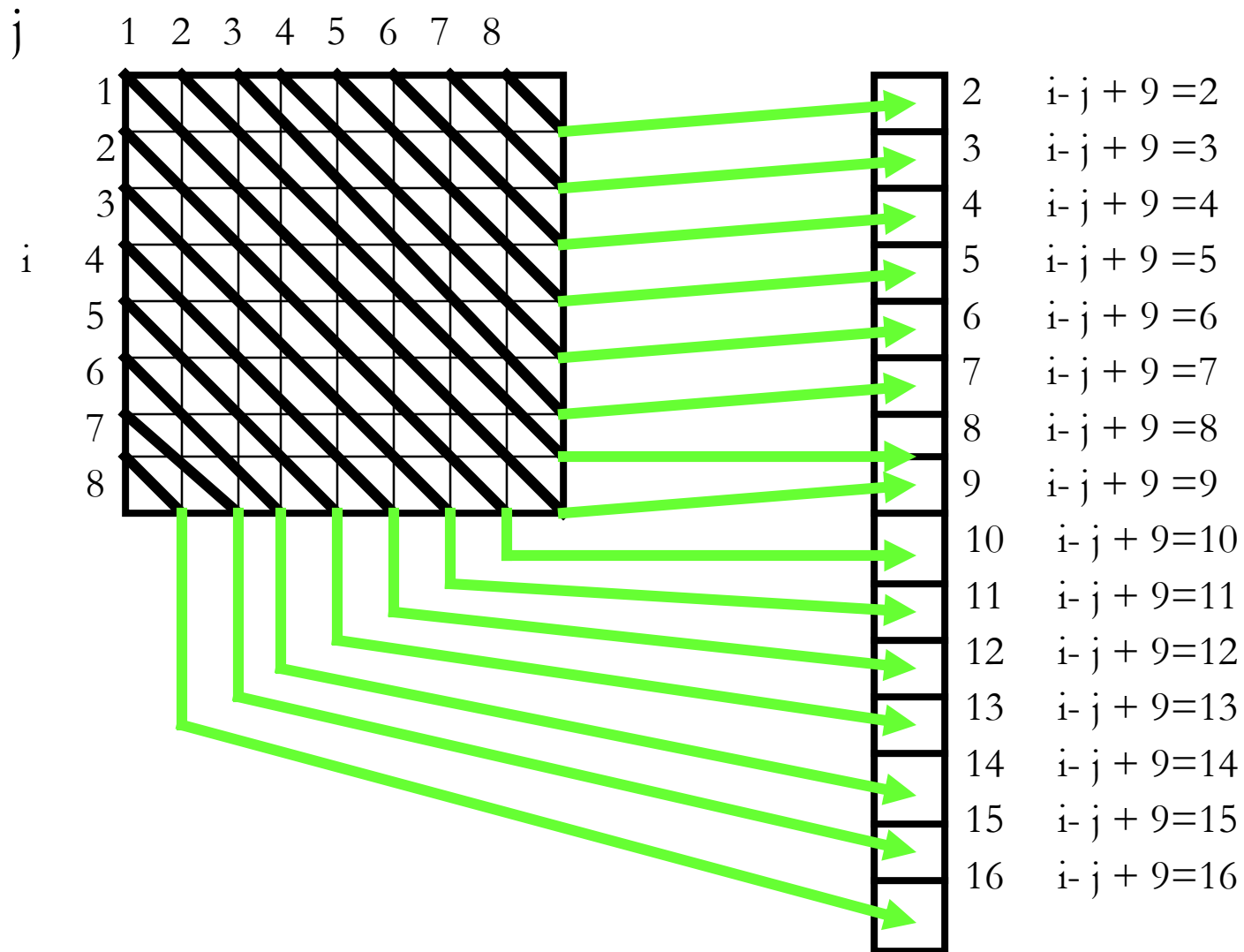
- 描述从右上至左下的对角线是否安全
- 数据类型为整型

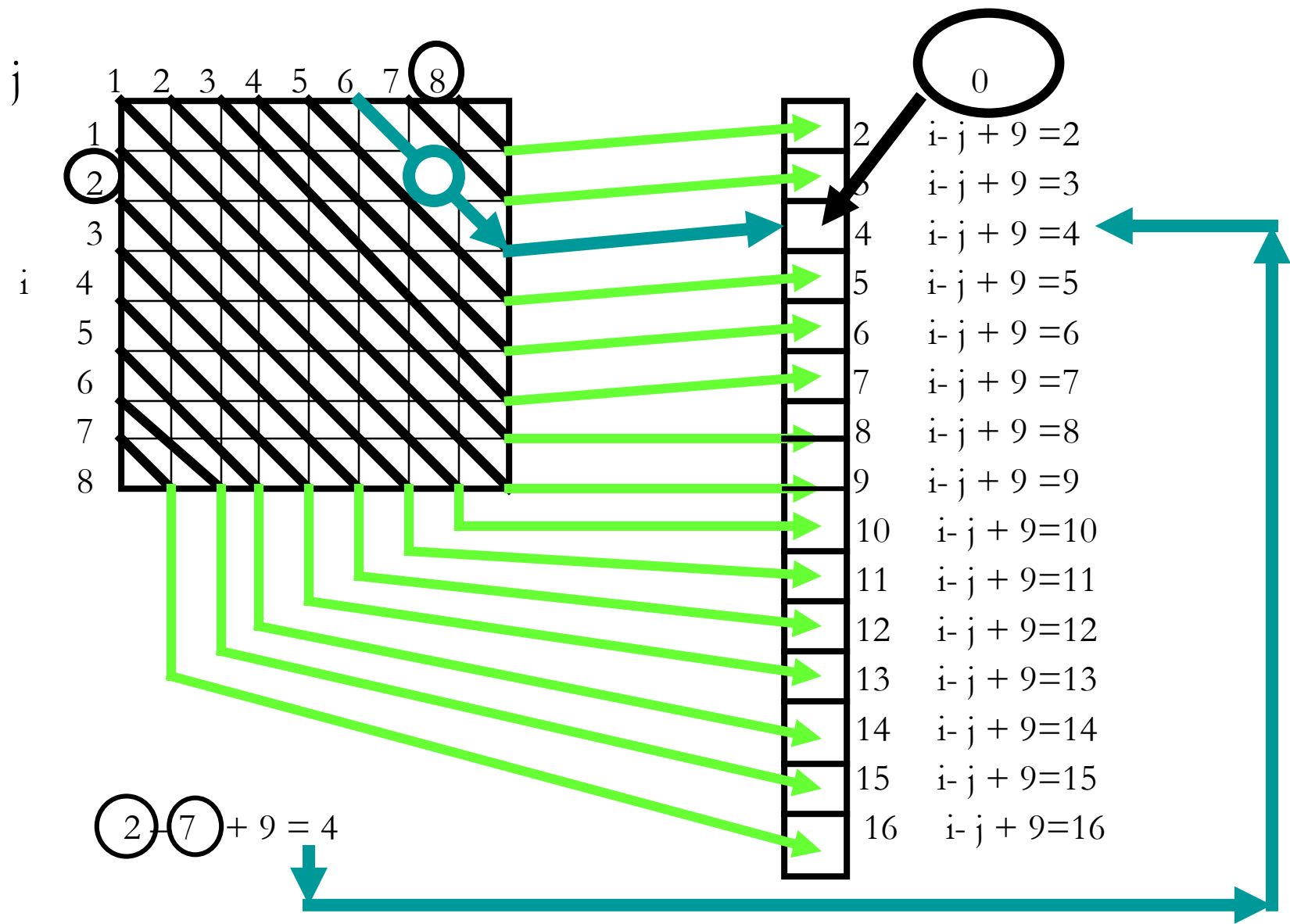
1-----安全

0-----不安全









- 定义 `int L[17]`

$$L[k] \quad k = 2, 3, \dots, 16$$

$$k = i - j + 9 \quad i = 1, 2, \dots, 8 \quad j = 1, 2, \dots, 8$$

- 描述从左上至右下的对角线是否安全
- 数据类型为整型：1----安全 0----不安全
- 利用这个特点，我们可以令

$$L[i - j + 9] = 0;$$

$$R[i + j] = 0;$$

来表示在 (i, j) 位置放皇后之后，通过该位置的两条对角线上不安全。

- 这样我们得出了在 (i, j) 位置放皇后的安全条件为：

$$nq = C[j] \ \&\& \ L[i-j+9] \ \&\& \ R[i+j]$$

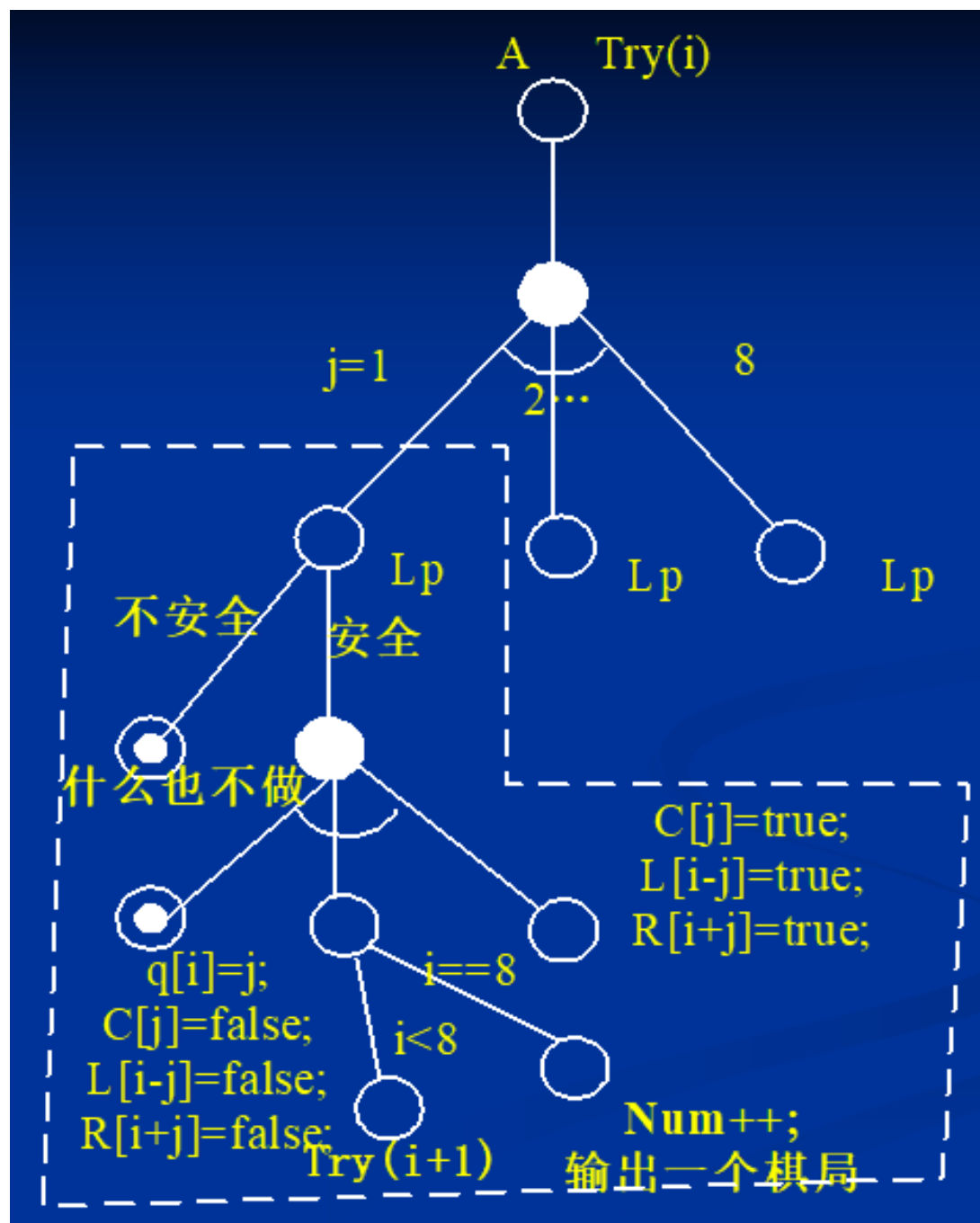
- 为了判断安全条件，在程序中要用到三个数组：

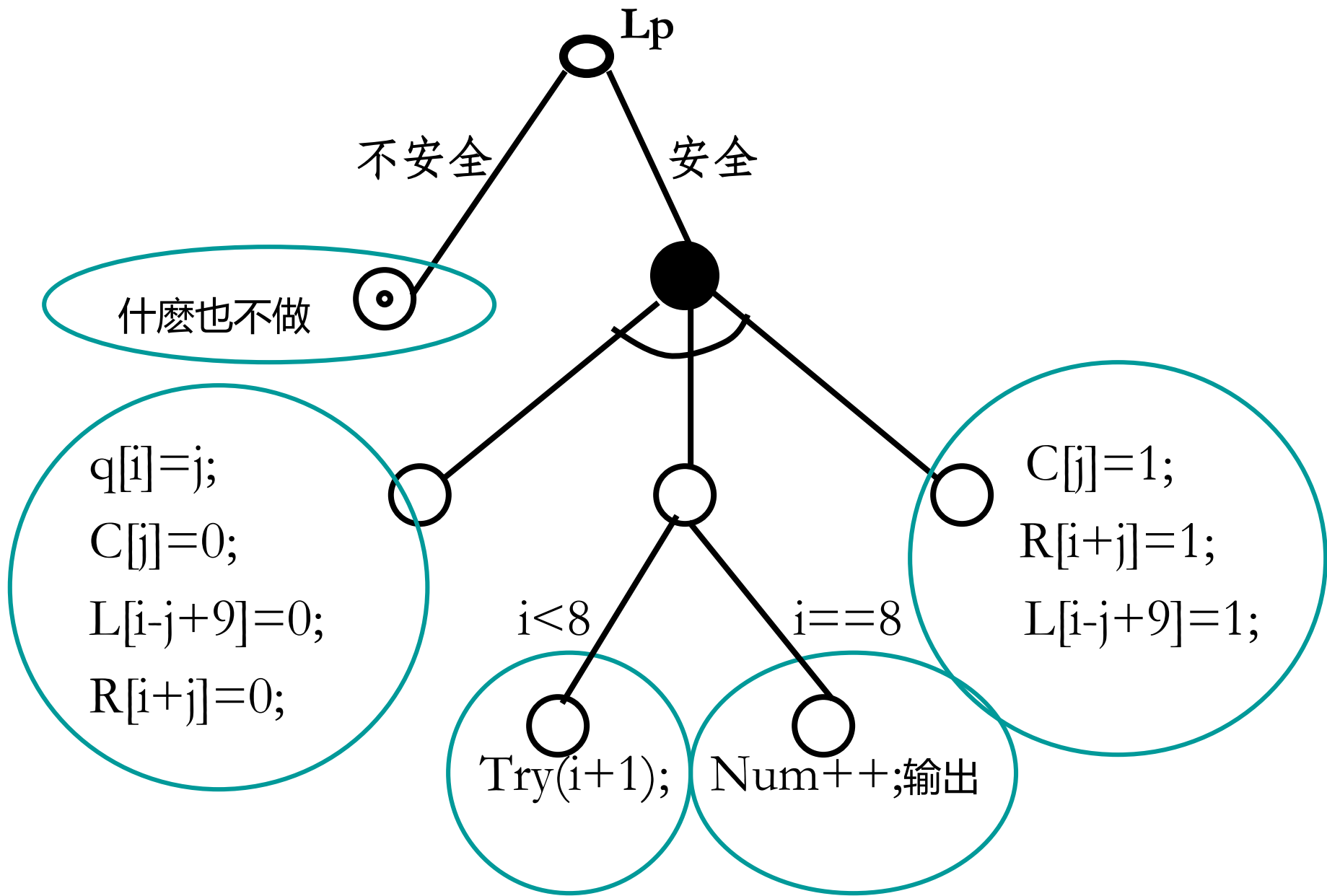
(1) $C[j]$ 为整型的， $j = 1, 2, \dots, 8$ ，初始化时全部置为 1；

(2) $L[k]$ 为整型的， $k = i - j + 9$ ， $k = 2, 3, \dots, 16$ ，初始化时全部置为 1；

(3) $R[m]$ 为整型的， $m = i + j$ ， $m = 2, 3, \dots, 16$ ，初始化时全部置为 1；

- 3、从思路，在放第 i 个皇后时（当然在第 i 行），选第 j 列，当 nq 为 1 时，就可将皇后放在 (i, j) 位置，做如下 3 件事
- (1) 放皇后 $q[i] = j$ ，同时让第 j 列和过 (i, j) 位置的两条对角线变为不安全。即让 $C[j] = 0$; $L[i-j+9] = 0$; $R[i+j] = 0$;
 - (2) 之后查一下 i 是否为 8，如果为 8，则表明已经放完 8 个皇后，这时让方案数 Num 加 1，输出该方案下 8 个皇后在棋盘上的位置。否则，未到 8 个，还要让皇后数 i 加 1 再试着放，这时还要递归调用 $Try(i+1)$ 。
 - (3) 是为了寻找不同方案用的。当着一个方案输出之后，要回溯，将先前放的皇后从棋盘上拿起来，看看还有没有可能换一处放置。这时要将被拿起来的皇后的所在位置的第 j 列，和两条对角线恢复为安全的。我们用与或图来描述 8 皇后问题的解题思路。






```

#include <stdio.h>    // 预编译命令
const int Normalize = 9;    // 定义常量, 用来统一数组下标
int Num;                // 整型变量, 记录方案数
int q[9];                // 记录8个皇后所占用的列号
int C[9];                // C[1]~C[8], 布尔型变量, 当前列是否安全
int L[17];                // L[2]~L[16], 布尔型变量, (i-j)对角线是否安全
int R[17];                // R[2]~R[16], 布尔型变量, (i+j)对角线是否安全

```

```

void Try(int i)                // 被调用函数

```

```

{  int j;                // 循环变量, 表示列号

```

```

    int k;                // 临时变量

```

```

    for (j=1; j<=8; j++)    // 循环

```

```

    {  if ( C[j]==1 && R[i+j]==1 && L[i-j+Normalize]==1 )

```

```

        // 表示第i行, 第j列是安全的
    }
}

```

```

{   q[i] = j;           // 第一件事, 占用位置(i,j)
    C[j] = 0;           // 修改安全标志, 包括所在列和两个对角线
    L[i-j+Normalize] = 0;
    R[i+j] = 0;
    if ( i<8 )          // 第二件事, 判断是否放完8个皇后
        Try(i+1);        // 则继续放下一个
    else                // 已经放完8个皇后
    {
        Num++;           // 方案数加1
        printf("方案%d: ", Num); // 输出方案号
        for ( k=1; k<=8; k++)
            printf("%d ", q[k]); // 输出具体方案
        printf ("\n" );
    }
    C[j] = 1;           // 第三件事, 修改安全标志, 回溯
    L[i-j+Normalize] = 1;
    R[i+j] = 1;
}

```

```

    }                // 循环结束
}                  // Try函数结束

int main()          //主函数
{
    int i;           // 循环变量
    Num = 0;          // 方案数清零
    for(i=0; i<9; i++) // 置所有列为安全
        C[i] = 1;

    for(i=0; i<17; i++) // 置所有对角线为安全
        L[i] = R[i] = 1;

    Try(1);          // 递归放置8个皇后，从第一个开始放

    return 0;
}

```

共92组解，部分答案如下：

方案1：1 5 8 6 3 7 2 4

方案2：1 6 8 3 7 4 2 5

方案3：1 7 4 6 8 2 5 3

方案4：1 7 5 8 2 4 6 3

方案5：2 4 6 8 3 1 7 5

方案6：2 5 7 1 3 8 6 4

方案7：2 5 7 4 1 8 6 3

方案8：2 6 1 7 4 8 3 5

方案9：2 6 8 3 1 4 7 5

方案10：2 7 3 6 8 5 1 4

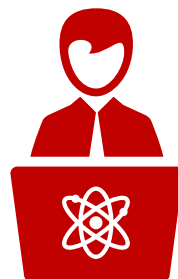
```

void Try(int i) {
    int j, k;
    if (i == 9) {                // 已经放完8个皇后
        Num++;                    // 方案数加1
        printf( "方案%d: ", Num); // 输出方案号
        for ( k = 1; k <= 8; k++)
            printf("%d ", q[k]); // 输出具体方案
        printf("\n"); }
    else
        for (j = 1; j <= 8; j++) // 循环
            if ( C[j] == 1 && R[i + j] == 1 && L[i - j + Normalize] == 1 ) {
                q[i] = j;        // 第一件事, 占用位置(i,j)
                C[j] = 0;        // 修改安全标志, 包括所在列和两个对角线
                L[i - j + Normalize] = 0;
                R[i + j] = 0;
                Try(i + 1);      // 则继续放下一个
                C[j] = 1;        // 第三件事, 修改安全标志, 回溯
                L[i - j + Normalize] = 1;
                R[i + j] = 1;
            }
}

```



中國人民大學
RENMIN UNIVERSITY OF CHINA



谢谢大家!

