中国人民大学
RENMIN UNIVERSITY OF CHINA

C程序设计

第8讲 递归

余力

buaayuli@ruc.edu.cn

内容

1. 初步递归

2. 再理解函数变量

3. 递归分析工具-与或图

4. 经典递归应用

中国人民大学
RENMIN UNIVERSITY OF CHINA

1. 初步递归

阶乘n!的求解 – 枚举

```
int fact(int n)
{   int m = 1;
    for (int i=1; i<=n; i++)
        m = m * i;
    return m;
}
```

循环变量m

承上启下

主动准备

- 根据公式，利用循环，进行枚举
  - 从数字1开始，一直枚举到数据n
  - 将枚举的数字乘起来得到结果

阶乘n!的求解 – 递推

```
int fact(int n) {
  int m[10];  // 假设n不超过10
  m[1] = 1;   // 递推的起始值
  for (int i=2; i<=n; i++)
      m[i] = m[i-1] * i;
  return m[n]; }
```

数组m

逐个求值

主动准备

- 递推数列
  - 从某一项起，任何一项都可以用它前面的若干项来确定，这样的数列称为递推数列
  - 阶乘n!，可以再求得前一项(n-1)!后求得

阶乘n!的求解 – 递归

```
int fact(int n) {
    if (n == 1)  return 1; // 终止条件
    else return n*fact(n-1);  // 直接返回
}  // 自己调用自己：递归
```

分工协作

多次调用fact

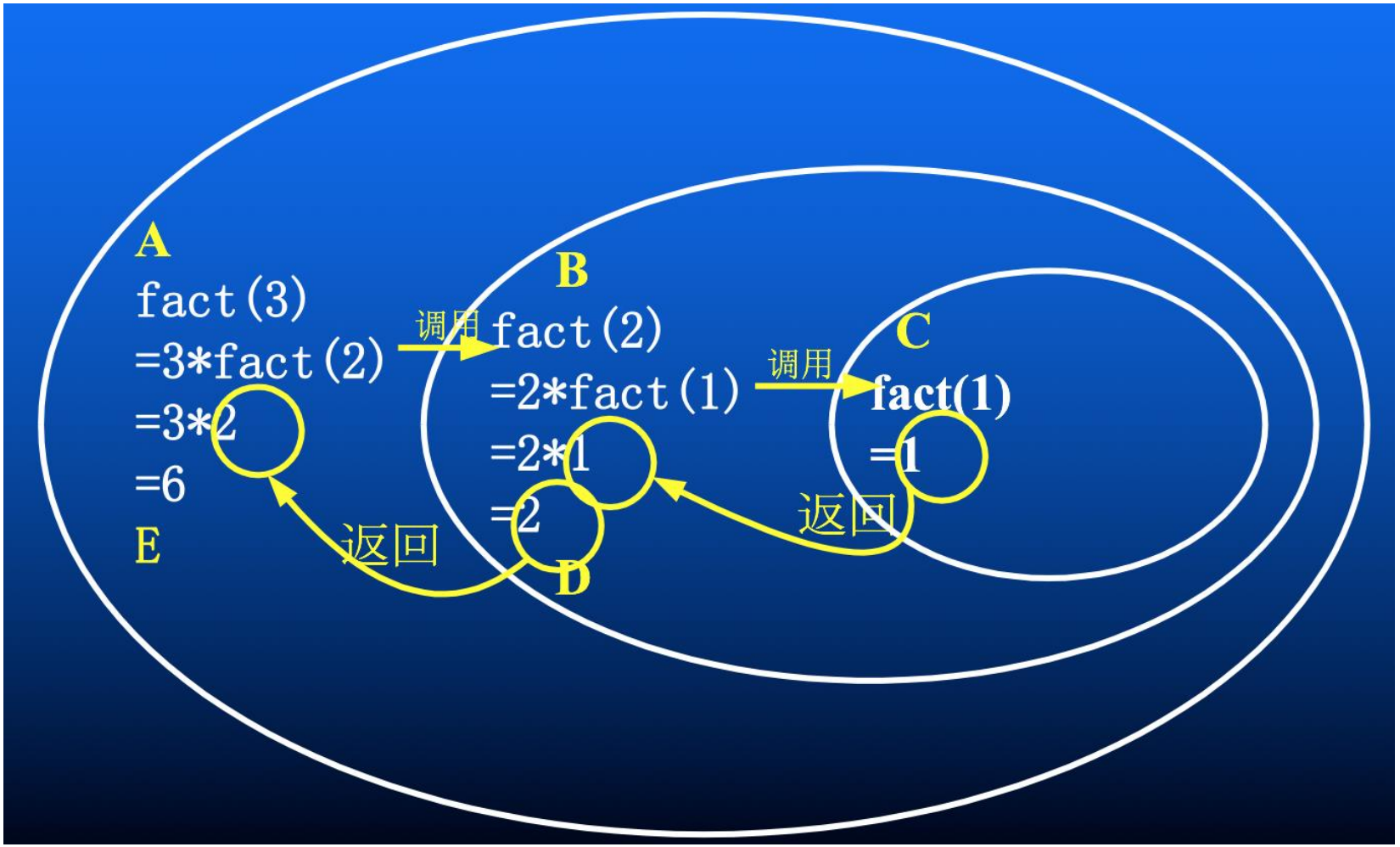
临时委托别人

等待结果

被动准备

- **递归：**
  - 核心：自己调用自己
  - 欲求阶乘n!，先调用 (n-1)!的结果

下面是fact(3)的调用和返回的示意图



斐波那契数列（枚举法）

```
int FibEnum (int n) {
  if (n==1 || n == 2) return 1;
  int fn1 = 1, fn2 = 1;
  int fn;
  for (int i = 3; i <= n; i ++) {
    fn = fn1 + fn2;
    fn2 = fn1;
    fn1 = fn;  }
  return fn;
}
```

斐波那契数列（递推法）

```
int FibIter (int n) {
  int fib[MAX_N];
  for（int i = 0; i < n; i ++)
    if (i == 0 || i == 1) fib[i] = 1;
    else fib[i] = fib[i-1] + fib[i-2];
return  fib[n-1];
}
```

- 在计算第i项的数值之前，已经计算出i-1项和i-2项，并且存储在数组中。

斐波那契数列（递归法）

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

- 把问题分解更小规模的问题
  - FibRecur(n-1) + FibRecur(n-2)
- 给出边界情况的解
  - if (n == 1 || n == 2) return 1

理解递归

- 递归并不符合我们的思维习惯
  - 日常生活中我们习惯一步接一步地做事
- 递归是一种基于函数调用的表达方式
  - 将解决某一问题抽象为函数
  - 把规模大的问题转化为规模小的子问题来解决
  - 函数自己调用自己
- 递归往往能以很简洁的代码，来解决非常复杂的问题

递归的关键要素

- 递归的组成要素
  - Base Case：问题最简单的情况，可以直接得到答案
  - Recursive/Inductive case：将复杂情况变成更简单的情况

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

Base Case

- if (n == 1 || n == 2) return 1

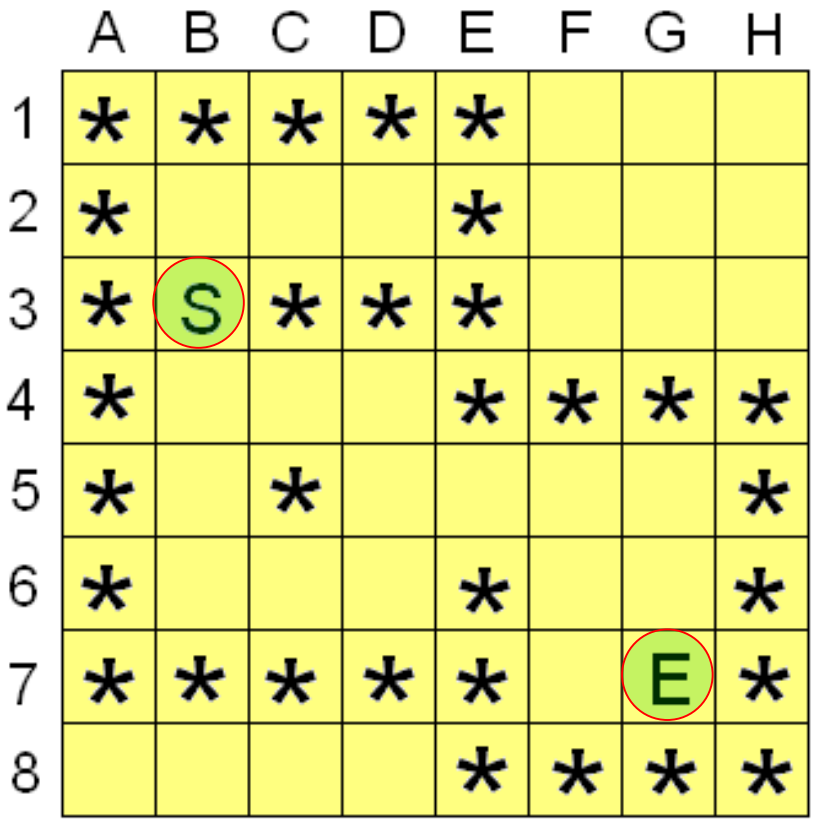
Inductive Case

- FibRecur(n-1) + FibRecur(n-2)

判断回文

- 回文例子
  - Able was I ere I saw Elba
- 如何使用递归实现
  - **Base Case:** 空字符串/长度为1的字符串
  - **Inductive Case:** 一个字符串是回文，当且仅当：
    - 该字符串第一个与最后一个字符相同
    - 中间的字符串是回文字符串

迷宫(Maze)



从S点走到E点，不能经过任何的*符号的方格

迷宫(Maze)

- 做一个判定性问题：迷宫走得通吗？
  - isMazeSolveable
- 迷宫递归解法思路
  - **Base Cases:**
    - 走到了E点，返回1
    - 走到*点，返回0
  - **Inductive Cases**
    - 从当前点开始能走通，只要东南西北四个方向只好有一个能走通

汉诺塔问题 (1)

- 在世界中心贝拿勒斯（在印度北部）的圣庙里，一块黄铜板上插着三根宝石针。印度教的主神梵天在创造世界的时候，在其中一根针上从下到上地穿好了由大到小的64片金片，这就是所谓的汉诺塔。不论白天黑夜，总有一个僧侣在按照下面的法则移动这些金片：一次只移动一片，不管在哪根针上，小片必须在大片上面。僧侣们预言，当所有的金片都从梵天穿好的那根针上移到另外一根针上时，世界就将在一声霹雳中消灭，而梵塔、庙宇和众生也都将同归于尽。

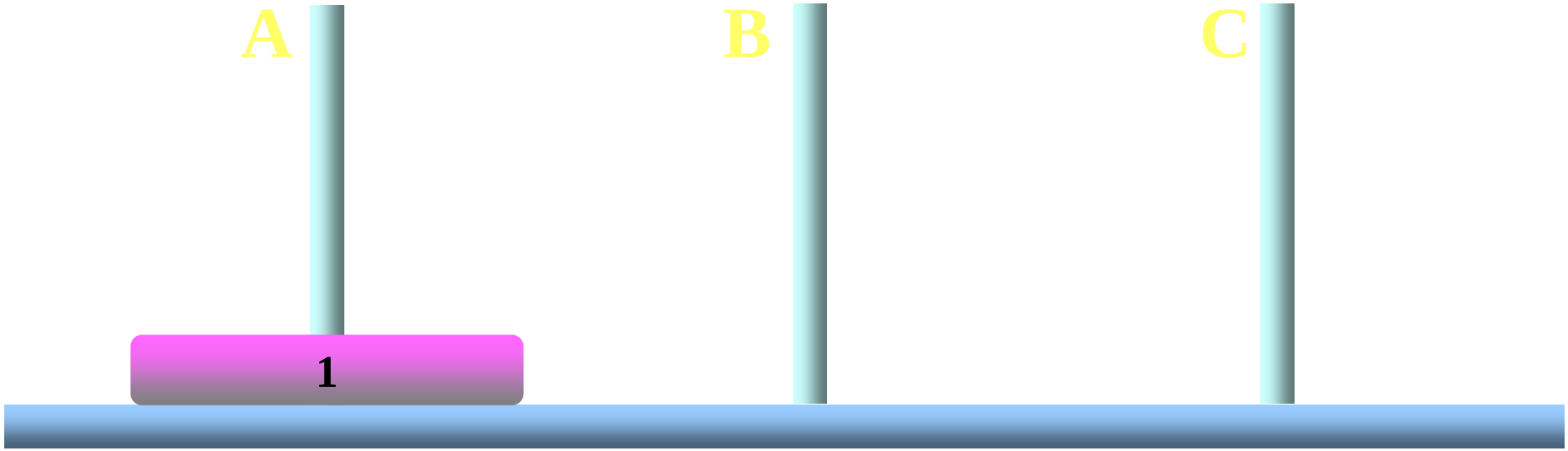
汉诺塔问题 (2)

¥9.80 包邮　　800+人付款

汉诺塔木制10层8层十层益智儿童汉罗塔玩具小学生逻辑思维训练

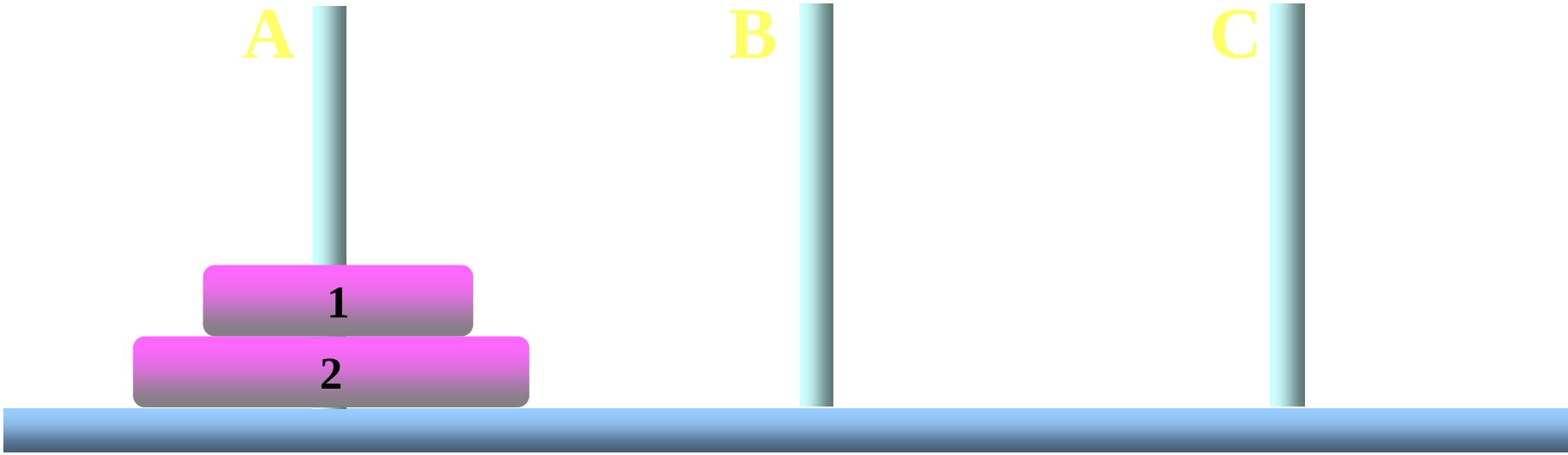
一个盘子的情况

- Base Case: 只有一个盘子的情况
  - 在A柱上只有一只盘子，假定盘号为 1，这时只需将该盘从 A 搬至 C，一次完成，记为 move 1 from A to C



二个盘子的情况

- 在 A 柱上有二只盘子，1 为小盘，2 为大盘
  - 将1号盘从A移至B，这是为了让 2号盘能移动 → **move 1 from A to B;**
  - 将 2 号盘从A 移至 C → **move 2 from A to C;**
  - 将 1 号盘从 B 移至 C → **move 1 form B to C;**



三个盘子的情况

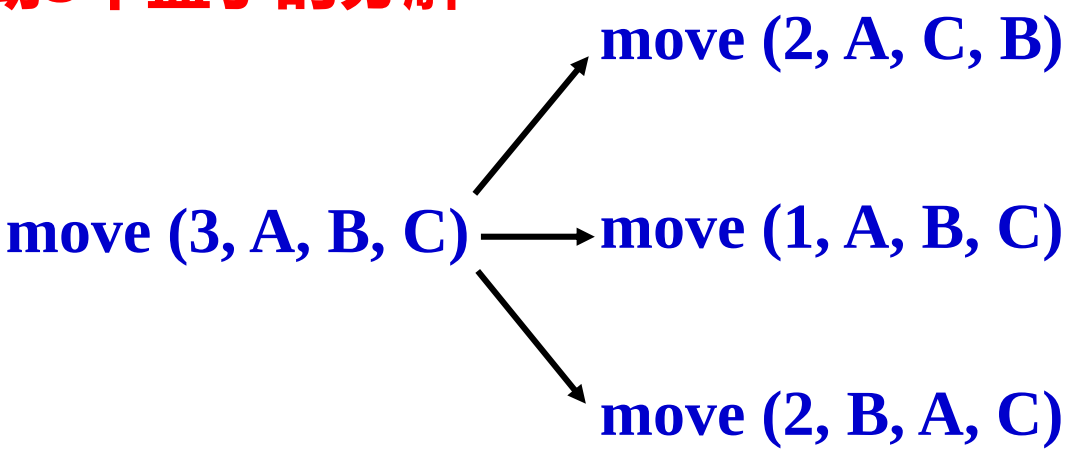
- 假设在 A 柱上有三只盘子，1 为小盘，2 为中盘，3为大盘，怎么处理？

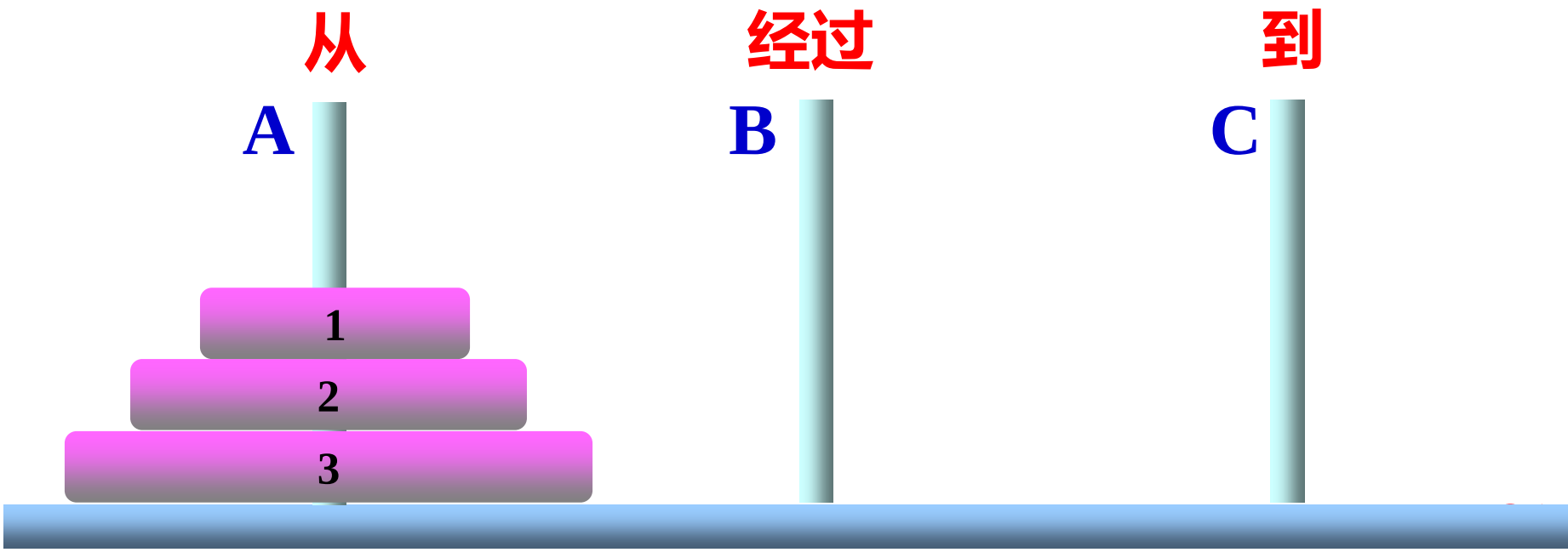
假设规模小的盘子移动问题

已经解决！

- 现将A柱上的1、2号盘子移到B柱上
- 把3号盘子移动到C柱上
- 再把1、2号盘子从B柱上移动到C柱上

演示：移动3个盘子的分解





move (3, A, B, C)

move (2, A, C, B)

move (1, A, B, C)

move (1, A, C, B)

move (1, C, A, B)

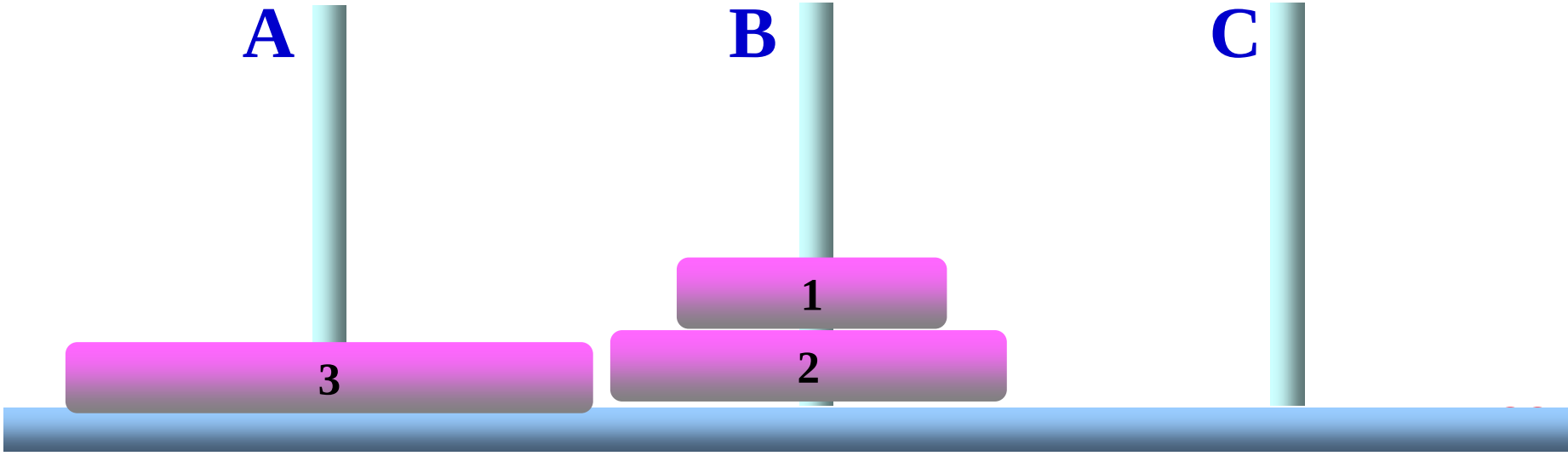
move (1, A, B, C)

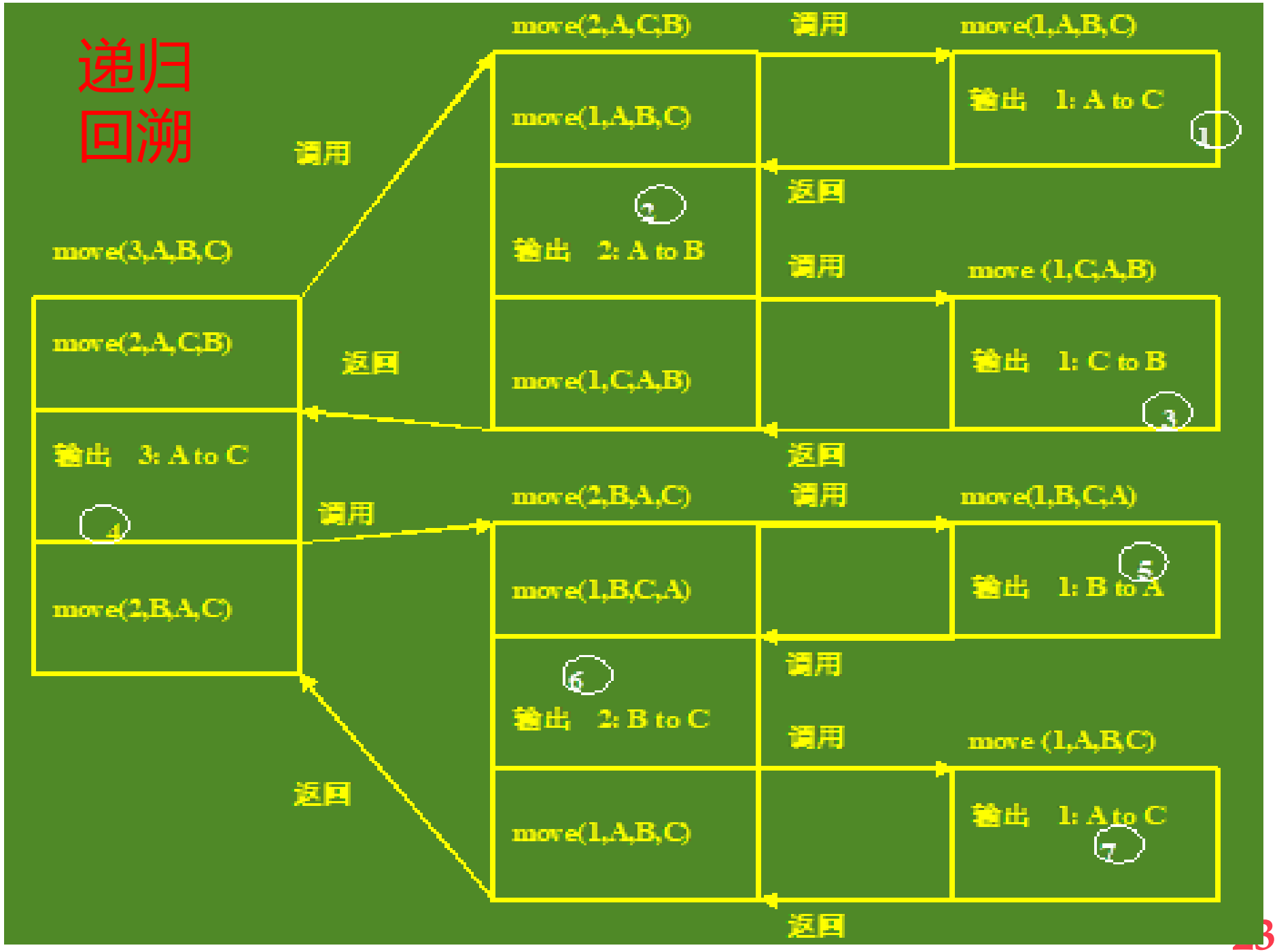
move (2, B, A, C)

move (1, B, C, A)

move (1, B, A, C)

move (1, A, B, C)




递归
回溯
move(3,A,B,C)
move(2,A,C,B)
输出 3: A to C
4
move(2,B,A,C)
调用
返回
move(2,A,C,B)
move(1,A,B,C)
2
输出 2: A to B
move(1,C,A,B)
调用
返回
move(1,A,B,C)
输出 1: A to C
1
调用
返回
move (1,C,A,B)
输出 1: C to B
3
调用
返回
move(2,B,A,C)
move(1,B,C,A)
6
输出 2: B to C
move(1,A,B,C)
调用
返回
move(1,B,C,A)
输出 1: B to A
5
调用
返回
move (1,A,B,C)
输出 1: A to C
7
调用
返回

```
#include <stdio.h>
int main()
{  void hanoi(int n,char one, char two,char three);
    int m;
    printf(  “the number of diskes:");
    scanf("%d",&m);
    printf("move %d diskes:\n",m);
    hanoi(3, '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",count, x,y); }
```

```
the number of diskes:3
move 3 diskes:
A-->C
A-->B
C-->B
A-->C
B-->A
B-->C
A-->C
```

+Chp08_汉诺塔.cpp

中国人民大学
RENMIN UNIVERSITY OF CHINA

2. 再理解函数变量

动态变量

- 动态变量是在程序执行的某一时刻被动态地建立并在另一时刻被动态地撤销的一种变量。它们存在于程序的局部，也只在这个局部中可以使用。
- 动态变量有两种：
  - 自动变量（auto）
  - 寄存器变量（register）

自动变量（auto）

[auto]数据类型 变量名[=初值表达式]，…；

main函数

```
x = 1
        x = 3
        call Prt
        printf( "2nd x= %d \n", x );
printf( "1st x = %d \n", x );
```

Prt函数

```
x = 5
printf( "3th x = %d \n", x );
```

寄存器变量（register）

- 当把一个变量指定为寄存器存储类别时，系统就将它存放在CPU的一个寄存器中。
- 各CPU寄存器的个数和长度不同，因此，C标准对寄存器存储类别只给出建议，不作硬性规定。

大概了解一下就行！

静态变量（static）

- static变量的存储空间在程序的整个运行期间是固定的、有效的，而不像动态变量是在程序执行中被动态建立和动态撤销的。
- static变量在编译时就为其分配存储空间，程序一开始执行便被建立，直到程序执行结束都是存在的。
- static变量的初始化是在编译时进行的。在定义时只能使用常量或常量表达式进行初始化。未显示初始化时，编译将把它们初始化为0（对int型）或0.0（对float型）。
- 在函数多次调用的过程中，静态局部变量的值具有继承性。但其值只能在本函（或分程序）中使用。

重点理解！

static变量

```
void increment( void )
{
      static int x=0;
      x++;
      printf("%d\n",x);
}

int main( )
{
      increment();
      increment();
      increment();
      return 0;
}
```

运行结果:

1

2

3

```
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); }
```

如何

统计移动次数？

+Chp08_汉诺塔.cpp

外部变量（extern）

- 定义在所有函数之外的变量称为外部变量。
- 外部变量是全局变量，它的作用域是从定义的位置开始到本文件的结束。在一个函数中改变外部变量的值，那么其后引用该变量时，得到的值是已被改变的值。
- 外部变量可以不出现在文件的起始部分，而出现在函数之间。在此之前的函数是不能引用该外部变量的。
- 外部变量的初始化，可以不在的定义处。

存储类别小结

- 变量的定义需要指定两种属性：
  - 数据类型：int、float、char、double、long、short、unsigned
  - 存储类别：static、**auto**、register、extern
- 变量的有效性从两方面：
  - 作用域（空间有效）：全局 + 局部
  - 生存期（时间有效）：静态 + 动态

作用域（空间）

局部变量
- auto变量，动态局部变量（离开函数，变量消失）
- static局部变量（离开函数，变量仍保留）
- register变量（离开函数，变量消失）
- 形式参数（或定义为auto或register变量）

全局变量
- 静态外部变量（只限本源程序文件引用）
- 外部变量（允许其它源程序文件引用）

生存期（时间）

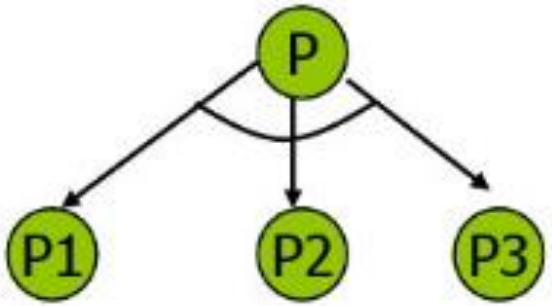
- 动态存储
  - auto变量（本函数内有效）
  - register变量（本函数内有效）
  - 形式参数
- 静态存储
  - 静态局部变量（函数内有效）
  - 静态外部变量（本文件内有效）
  - 外部变量（其它文件可以引用）

中国人民大学
RENMIN UNIVERSITY OF CHINA

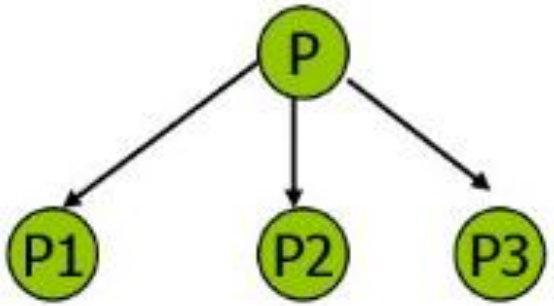
3. 递归分析工具-与或图

与或图

- **与图**：把一个原问题**分解**为若干个子问题，P1，P2，P3，…可用“与图”表示；P1，P2，P3，…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，P1，P2，P3，…可用“或图”表示；P1，P2，P3，…对应的子问题节点称为“或节点”。



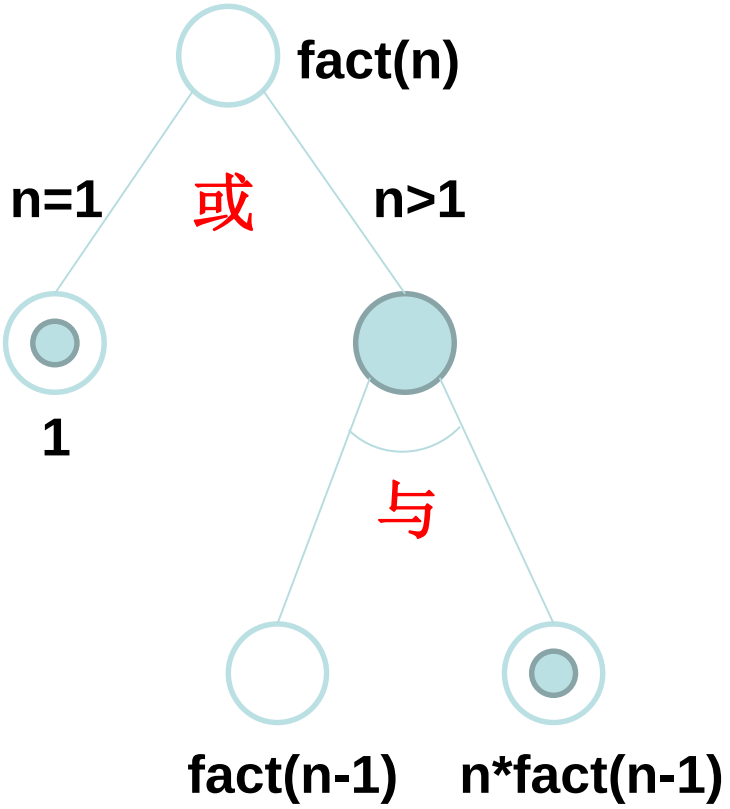
与



或

阶乘

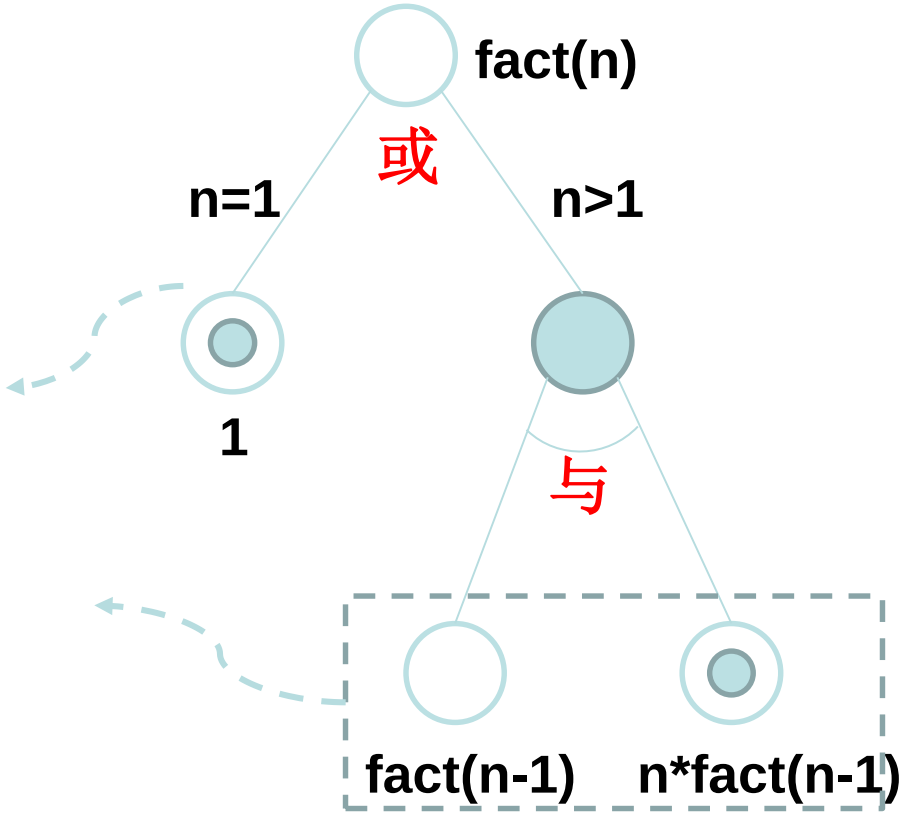
- 阶乘n!的计算
  - Base case：
    - n=1时，返回结果1
  - Inductive case:
    - n>1时，返回n!=n*(n-1)!



如何用图表形式表示?

与或图与递归程序编写

```
int fact (int n) {
    if (n==1) return 1;
    else {
        int f = fact(n-1);
        return n*f;
    }
}
int main () {
    printf("%d", fact(3));
}
```

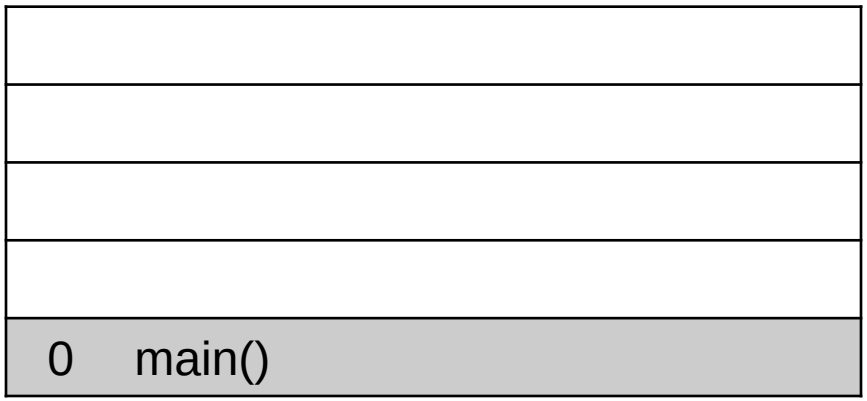


+Chpa08_Fact(阶乘).cpp

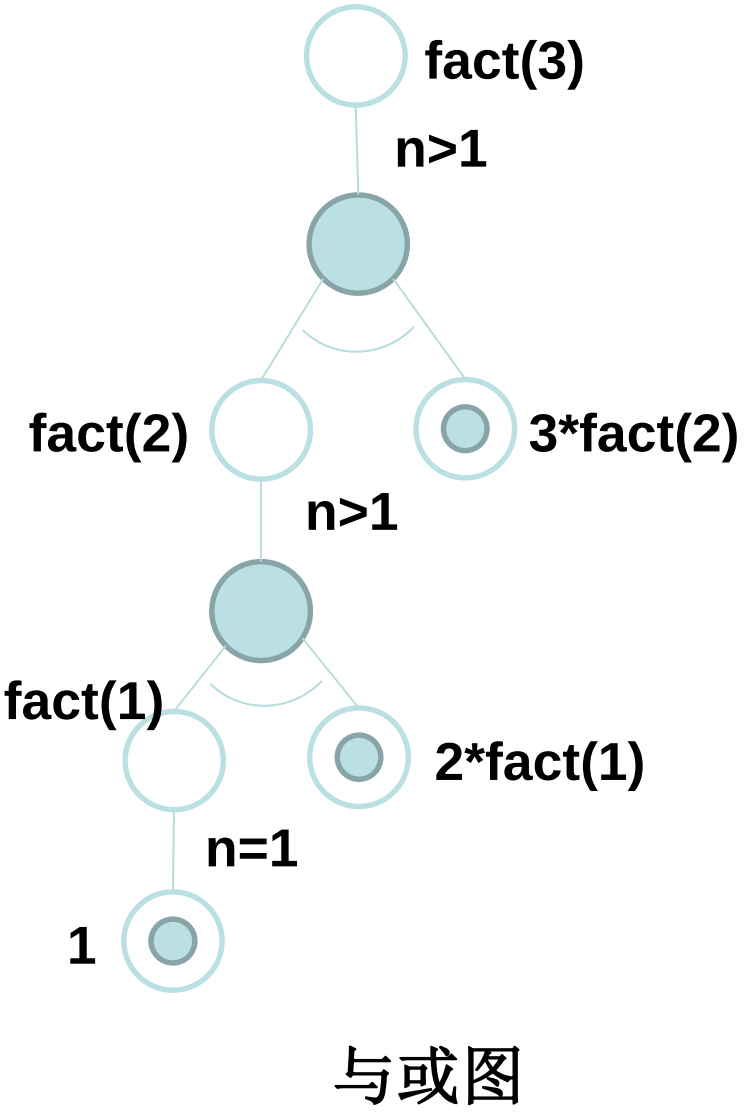
执行过程

```
int fact (int n) {
    if (n==1) return 1;
    else {
        int f = fact(n-1);
        return n*f;
    }
}
int main () {
    printf("%d", fact(3));

}
```



函数调用栈(Call Stack)

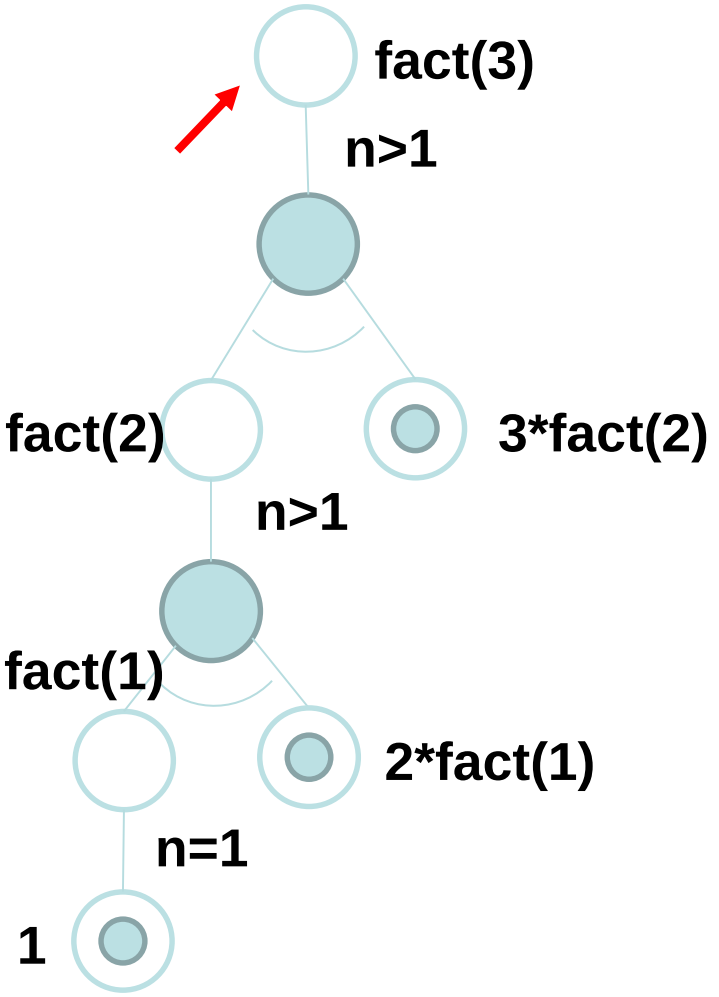


与或图

```
int fact (int n) {
    if (n==1) return 1;
    else {
        int f = fact(n-1);
        return n*f;
    }
}
int main () {
    printf("%d", fact(3));
}
```

1	fact(3)
0	main()

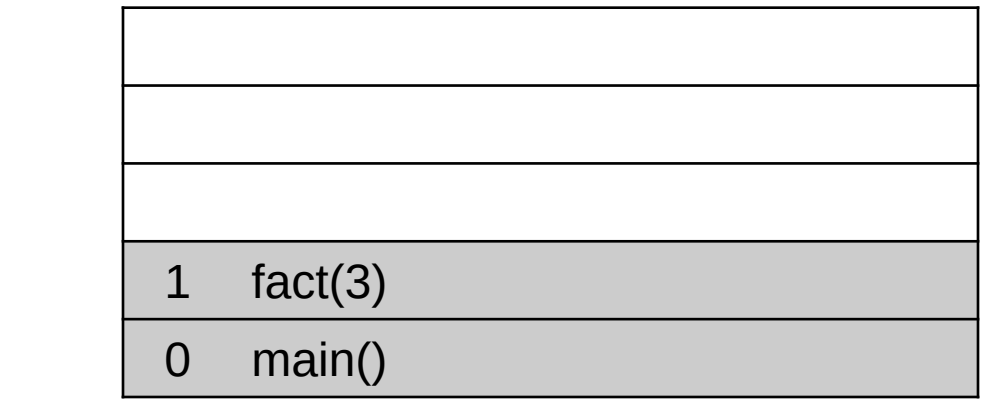
函数调用栈(Call Stack)



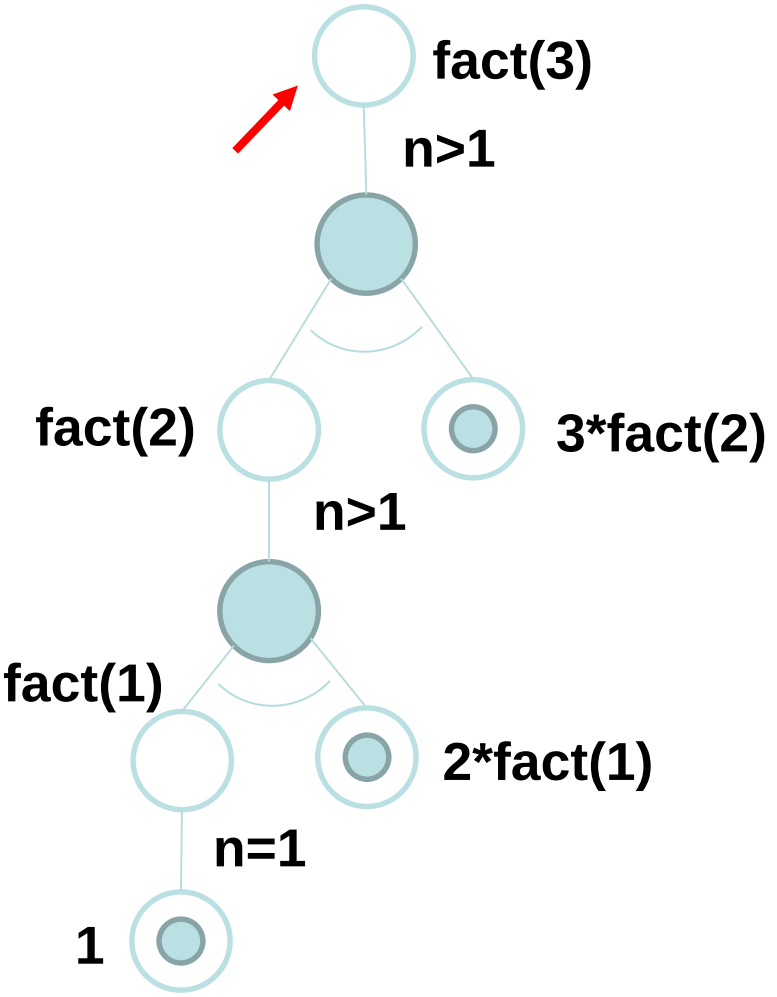
与或图

```
int fact (int n) {
    if (n==1) return 1;
    else {
        int f = fact(n-1);
        return n*f;
    }
}
int main () {
        printf("%d", fact(3));

}
```



函数调用栈(Call Stack)

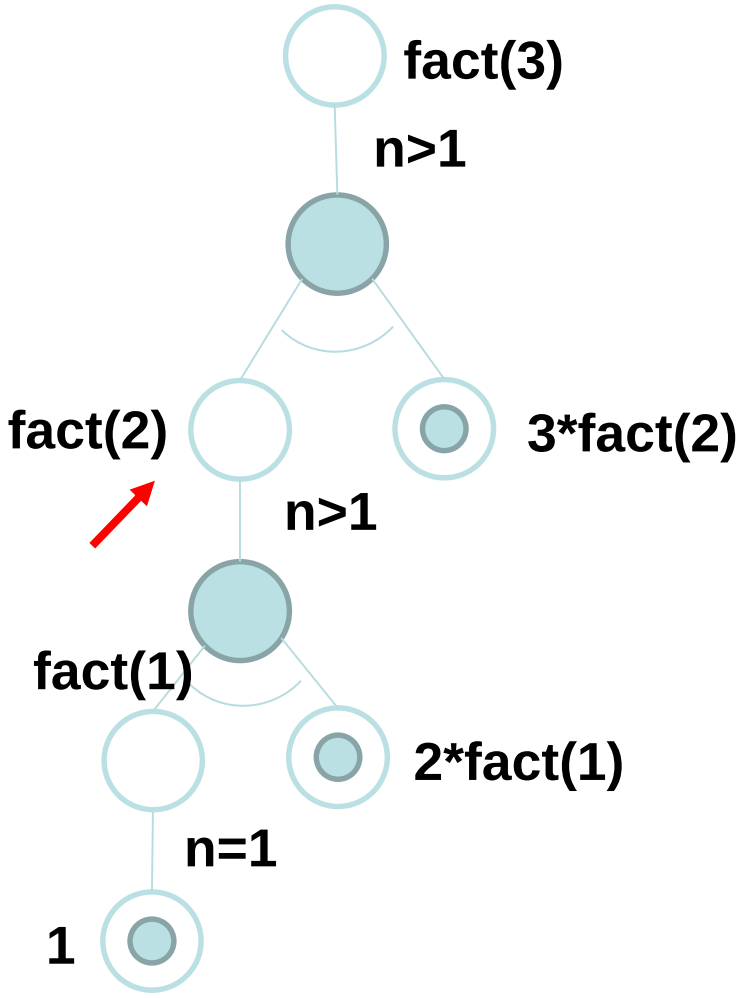


与或图

```
int fact (int n) {
    if (n==1) return 1;
    else {
        int f = fact(n-1);
        return n*f;
    }
}
int main () {
        printf("%d", fact(3));
}
```

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

函数调用栈(Call Stack)



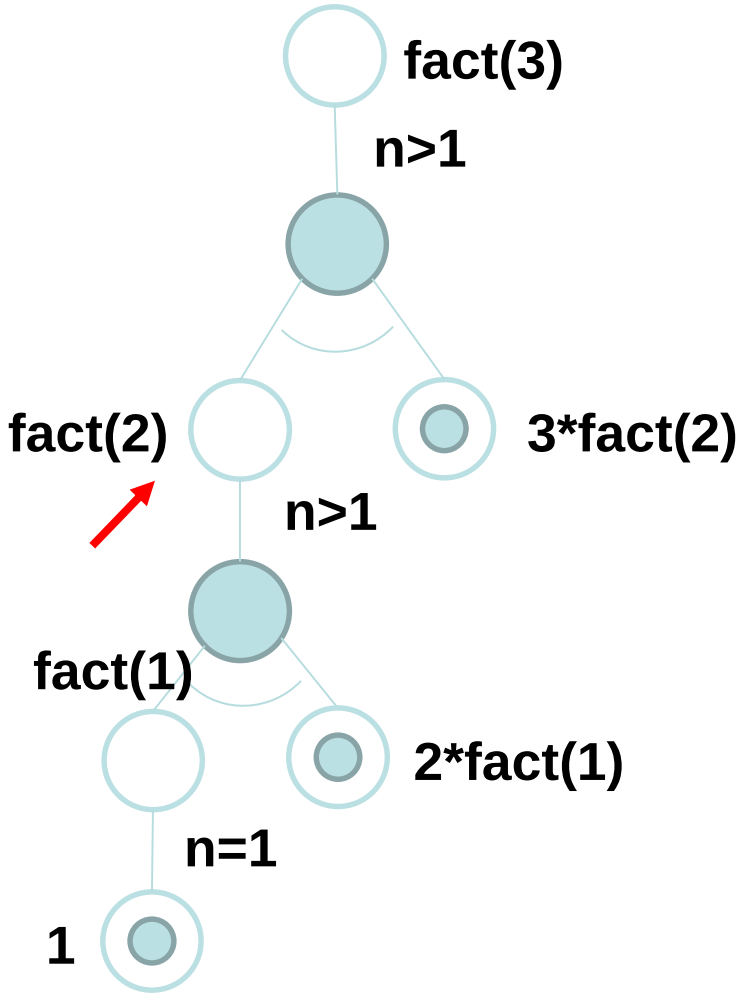
与或图

```
int fact (int n) {
    if (n==1) return 1;
    else {
        int f = fact(n-1);
        return n*f;
    }
}
int main () {
        printf("%d", fact(3));

}
```

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

函数调用栈(Call Stack)

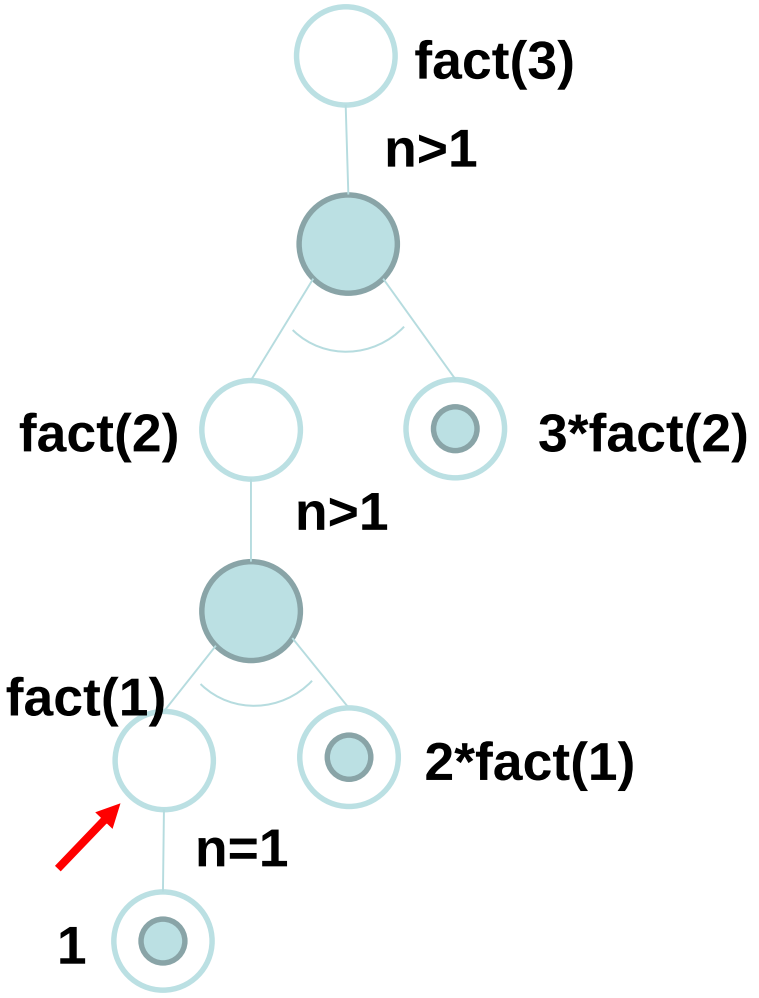


与或图

```
int fact (int n) {
    if (n==1) return 1;
    else {
        int f = fact(n-1);
        return n*f;
    }
}
int main () {
    printf("%d", fact(3));
}
```

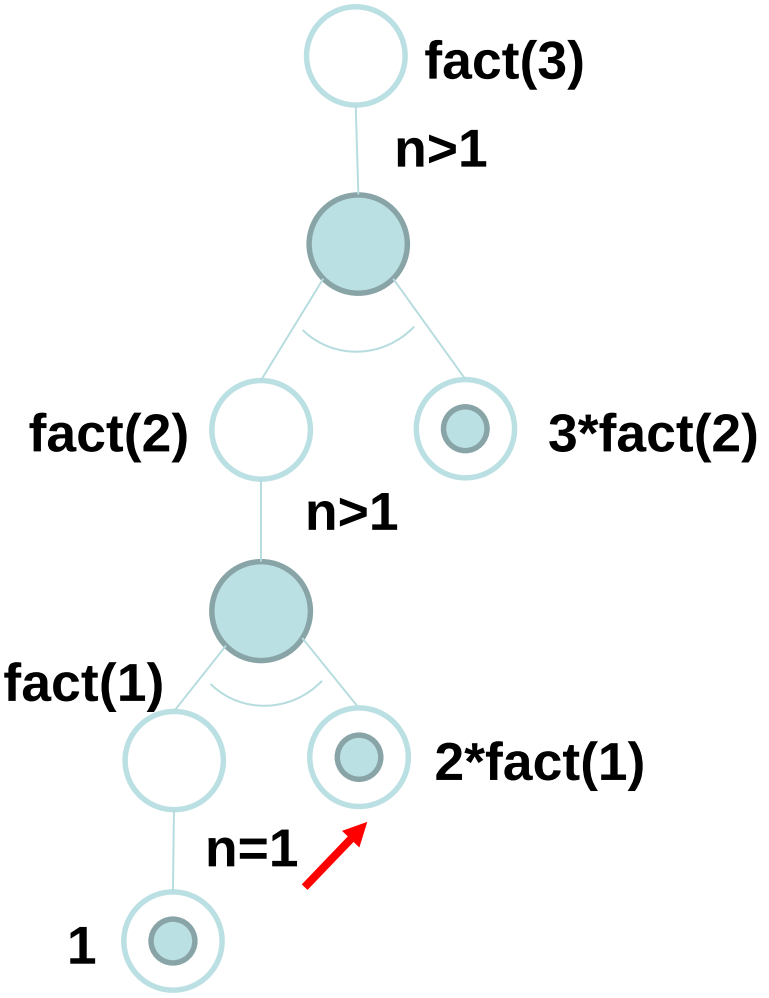
3	fact(1)
2	fact(2)
1	fact(3)
0	main()

函数调用栈(Call Stack)



与或图

```
int fact (int n) {
    if (n==1) return 1;
    else {
        int f = fact(n-1);
        return n*f;
    }
}
int main () {
    printf("%d", fact(3));
}
```



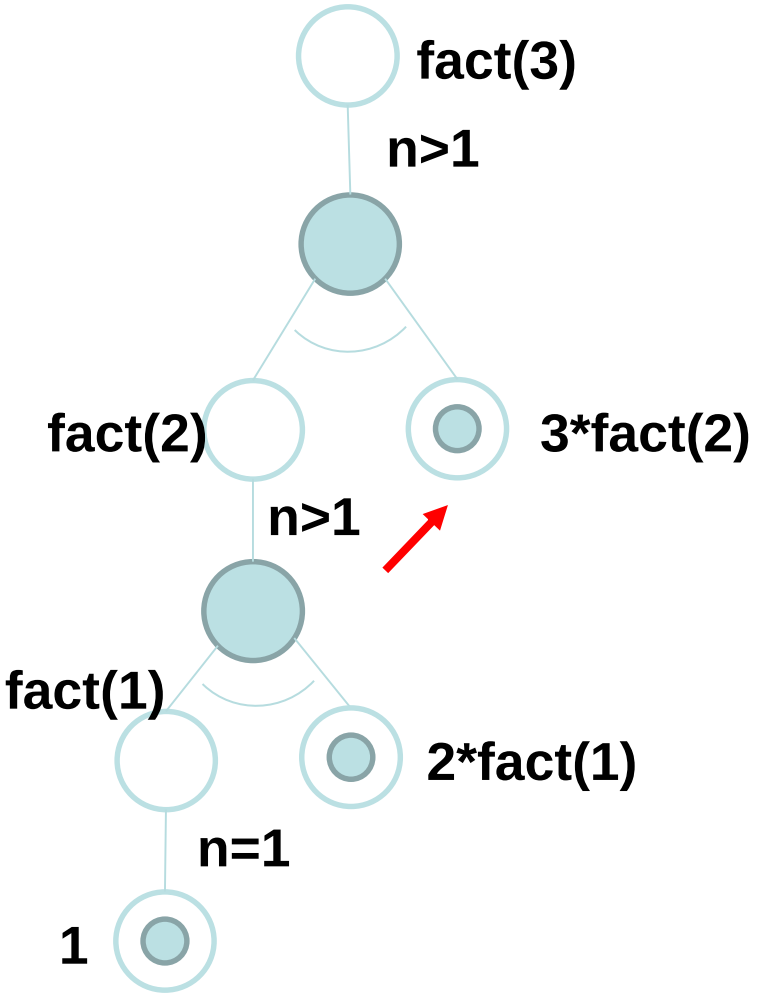
与或图

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

函数调用栈(Call Stack)

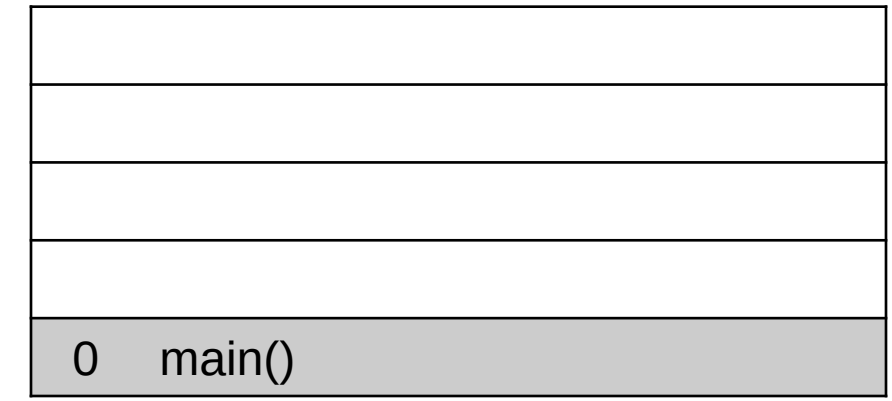
```
int fact (int n) {
    if (n==1) return 1;
    else {
        int f = fact(n-1);
        return n*f;
    }
}
int main () {
    printf("%d", fact(3));
}
```

函数调用栈(Call Stack)

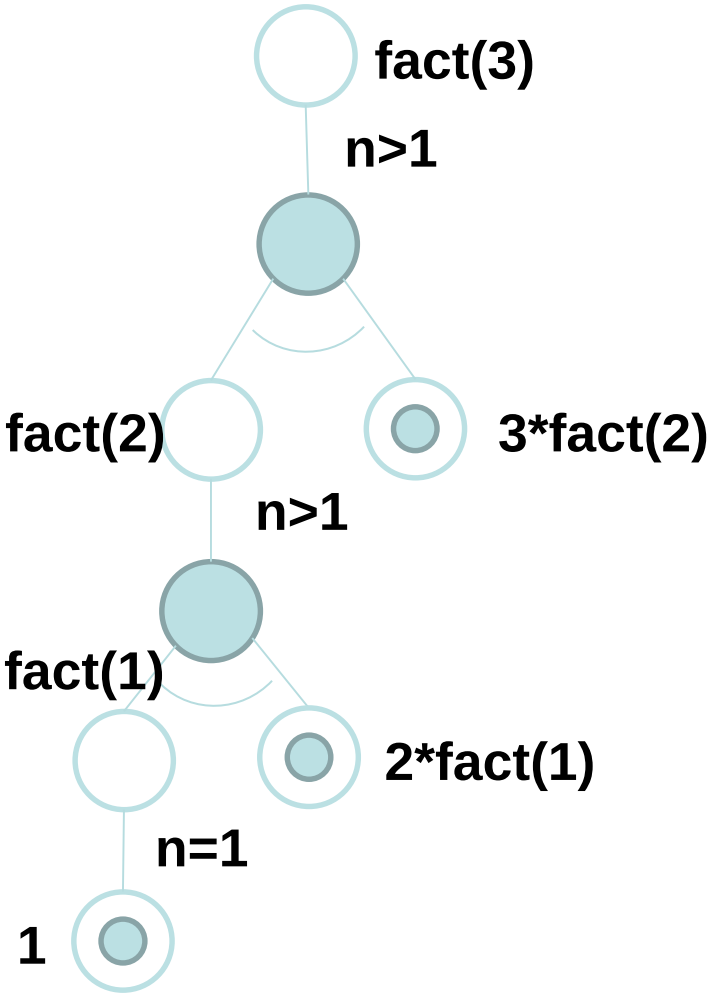


与或图

```
int fact (int n) {
    if (n==1) return 1;
    else {
        int f = fact(n-1);
        return n*f;
    }
}
int main () {
    printf("%d", fact(3));
}
```



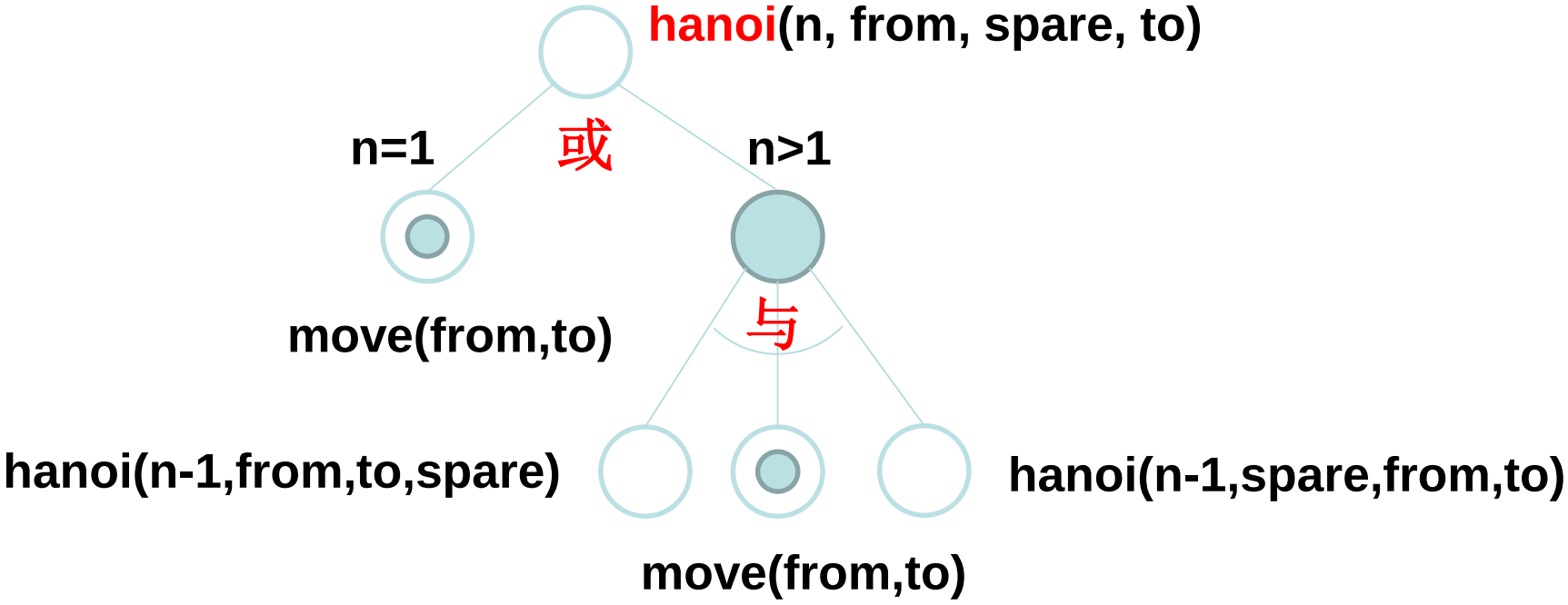
函数调用栈(Call Stack)



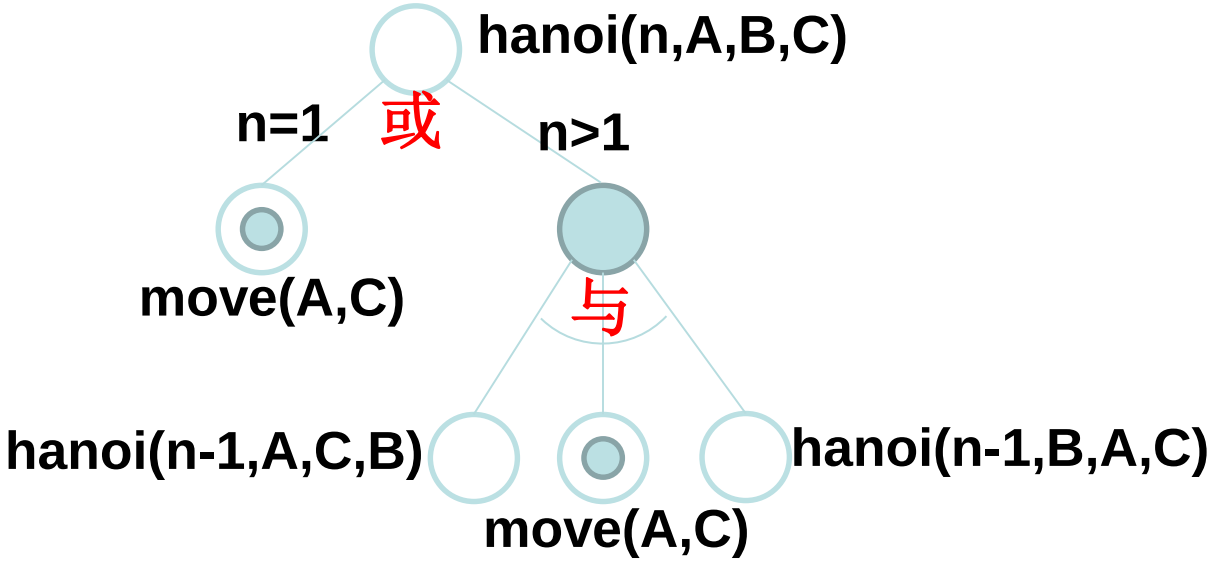
与或图

汉诺塔问题

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



汉诺塔

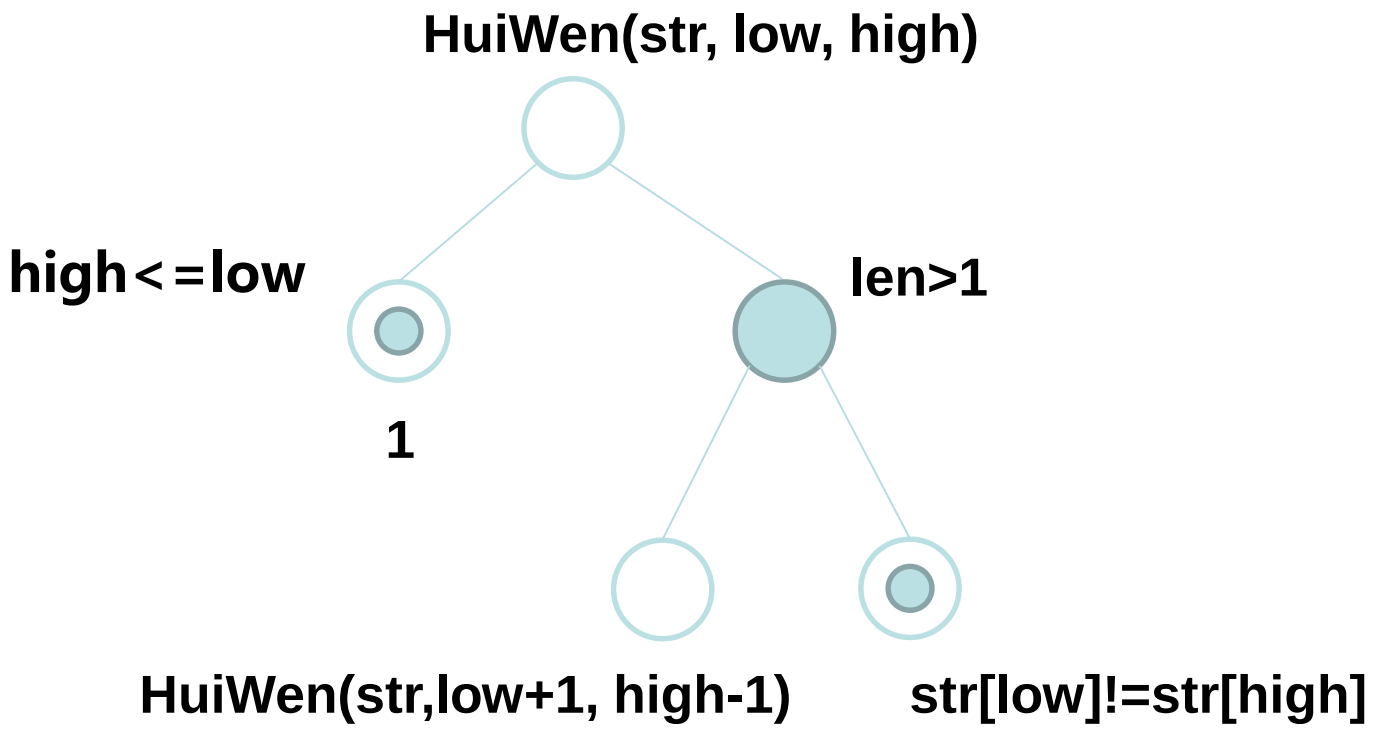



```
void move (int n, char A, char B, char C) {
    if (n == 1)
        cout << "move from " << A << " to " << C << endl;
    else {
        move (n-1, A, C, B);
        cout << "move from " << A << " to " << C << endl;
        move (n-1, B, A, C);
    }
}
int main () {
    int n = 4;
    char A = 'A',B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

+Chp08_汉诺塔.cpp

回文判断

- 问题表示为HuiWen(str, low, high)
  - 画出HuiWen(str,low,high)的与或图
  - 分析HuiWen (str,0,11)的情况



判断回文

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

+Chp08_回文判断(#110).cpp

```c
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;
}
```

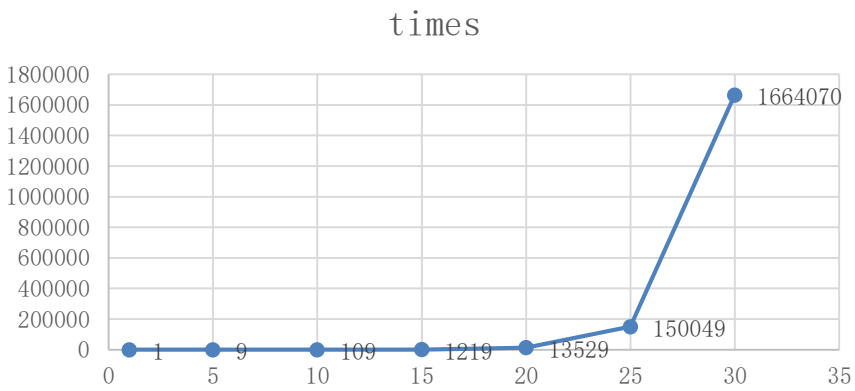
两种对比

```c
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

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

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

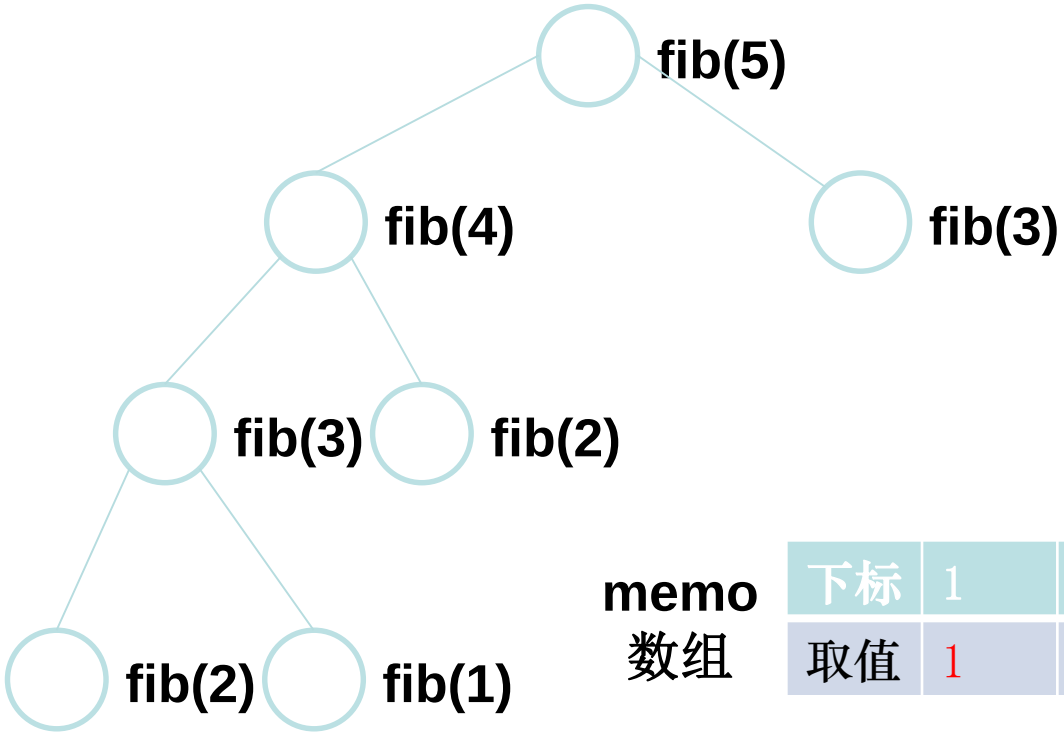


指数级增长！

+Chp08_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”

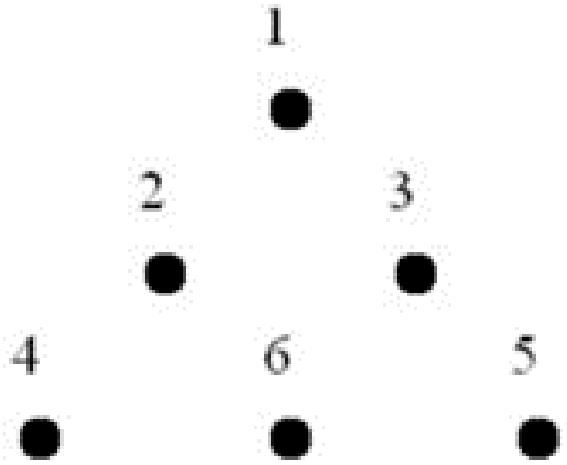


memo数组

下标	1	2	3	4	5
取值	1	1	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



#117 摘桃子

程度设计重要思想

用空间换时间

斐波那契数列

```
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;
    cout << "times: " << times << endl;}
```

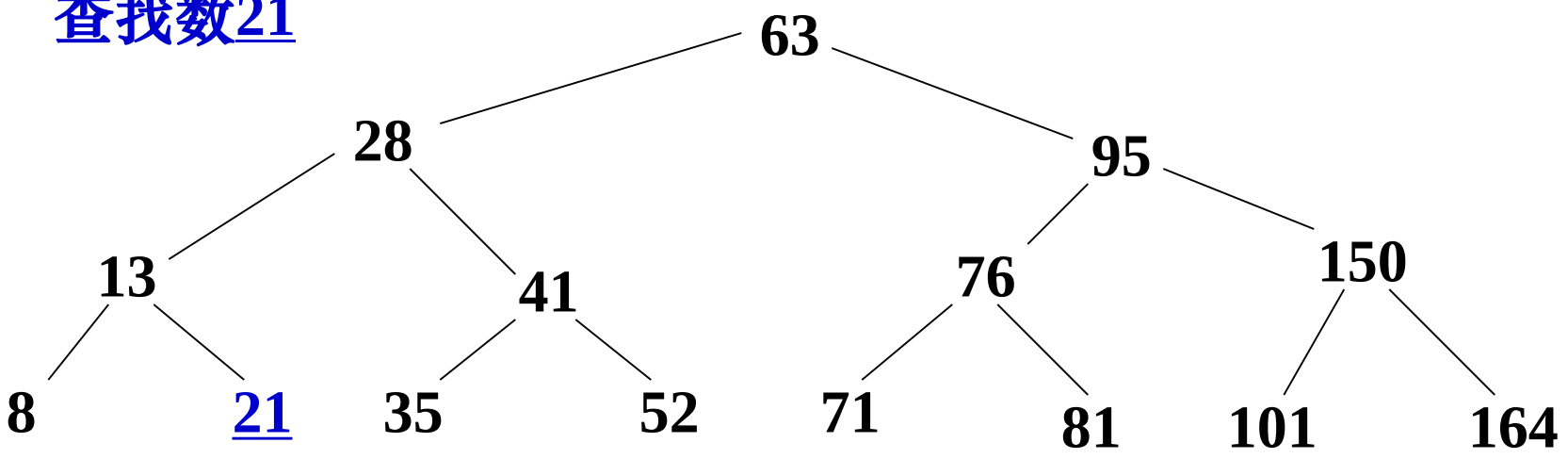
存储一下，

不要浪费了计算

大大节省运算

4. 经典递归应用

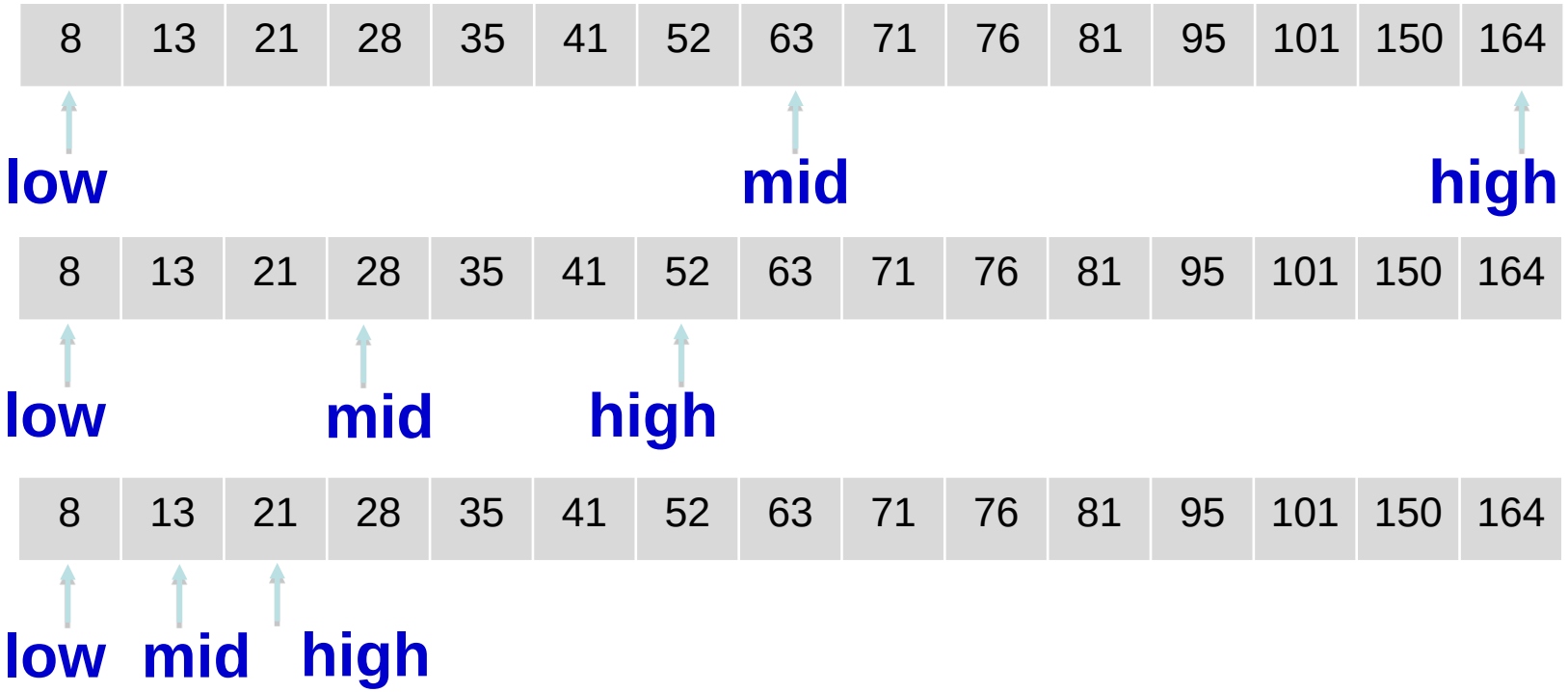
二分查找



思想：折半查找

二分查找的循环实现 (1)

核心难点：实现“折半”



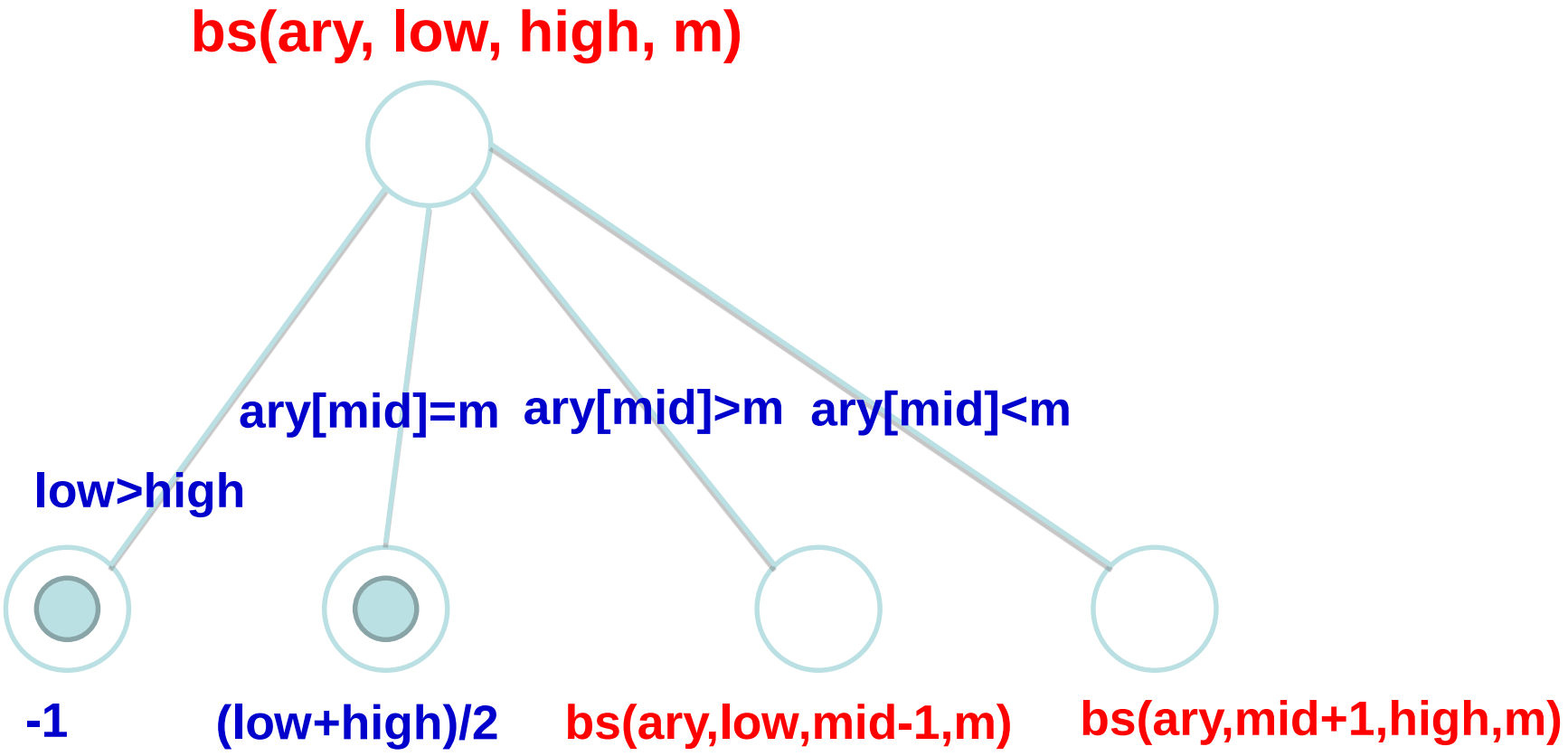
如何设置下标变量low, high, mid ?

二分查找的循环实现 (2)

```
int bisearch(int ary[], int low, int high, int m) {
  while( low<=high ) {
    int mid = (low+high)/2;
    if( ary[mid] == m ) return mid; //找到m
    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
}
```

函数一次性完成，内部循环

二分查找的递归实现 (1)


bs(ary, low, high, m)
ary[mid]=m
ary[mid]>m
ary[mid]<m
low>high
-1
(low+high)/2
bs(ary,low,mid-1,m)
bs(ary,mid+1,high,m)

二分查找的递归实现 (2)

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

多次函数调用

每一次轻松

函数压力减轻

+Chp08_Bisearch(二分查找)

```
int bisearch(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

压力在循环语句

递归方式

压力嵌套调用

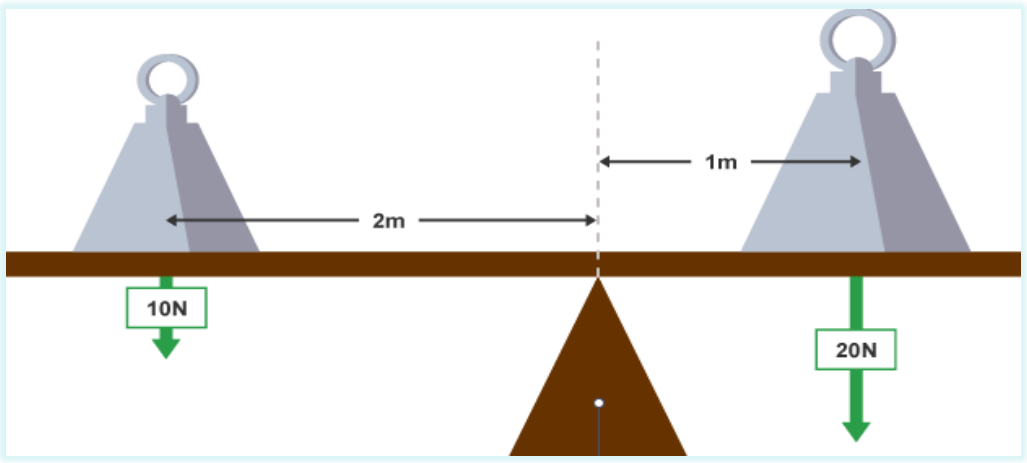
```
int bisearch(int ary[], int low, int high, int m) {
  if ( low > high )        //没找到
      return -1;
  int mid = (low + high) / 2;
  if (ary[mid] == m)  //找到
      return mid;
  else if ( ary[mid] > m )  //找左半边
      return bisearch(ary, low, mid - 1, m);
  else  //找右半边
      return bisearch(ary,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

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

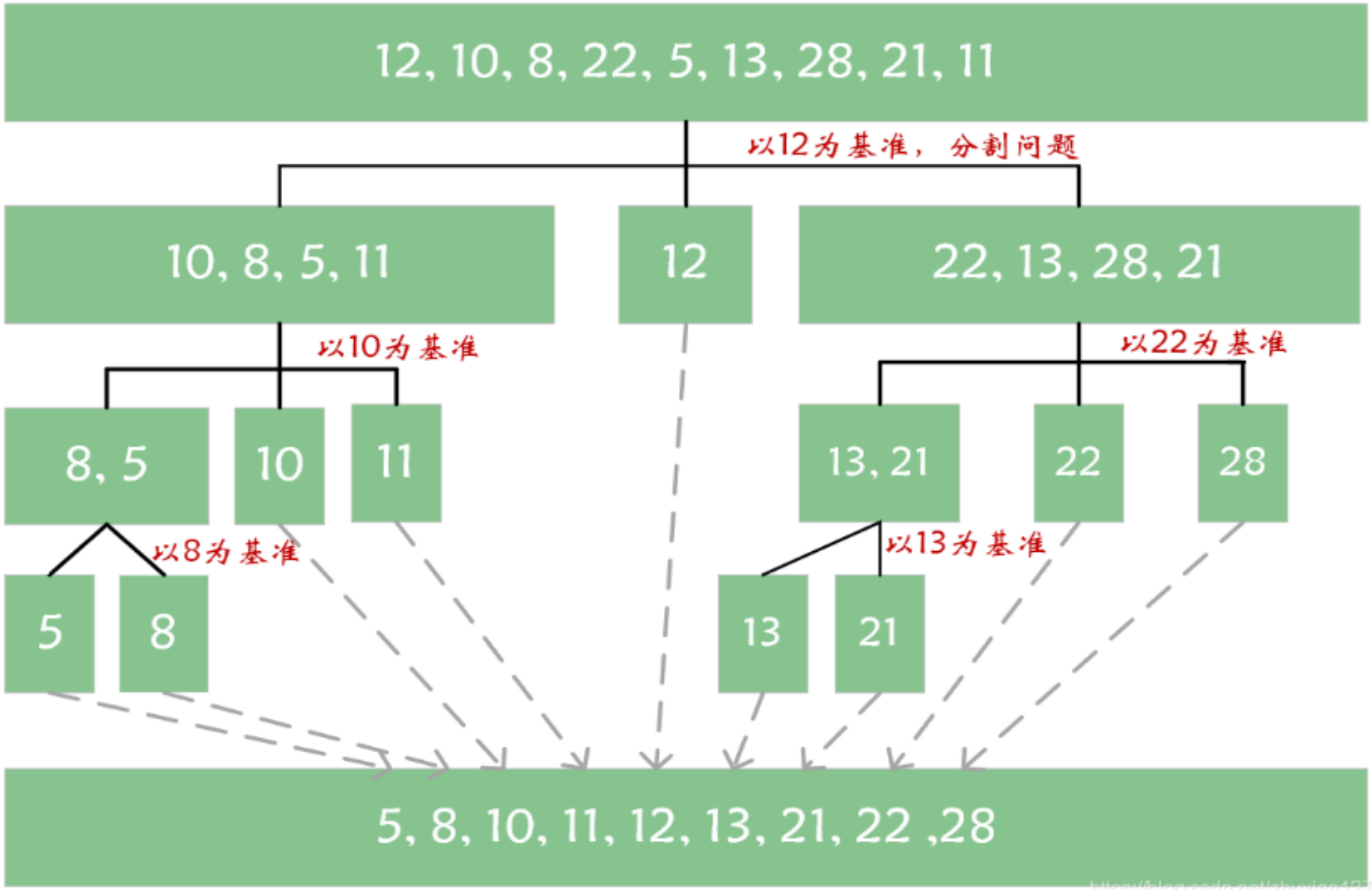
1. 选枢纽：从数组中选择出一个元素，称之为枢纽pivot元素



分区

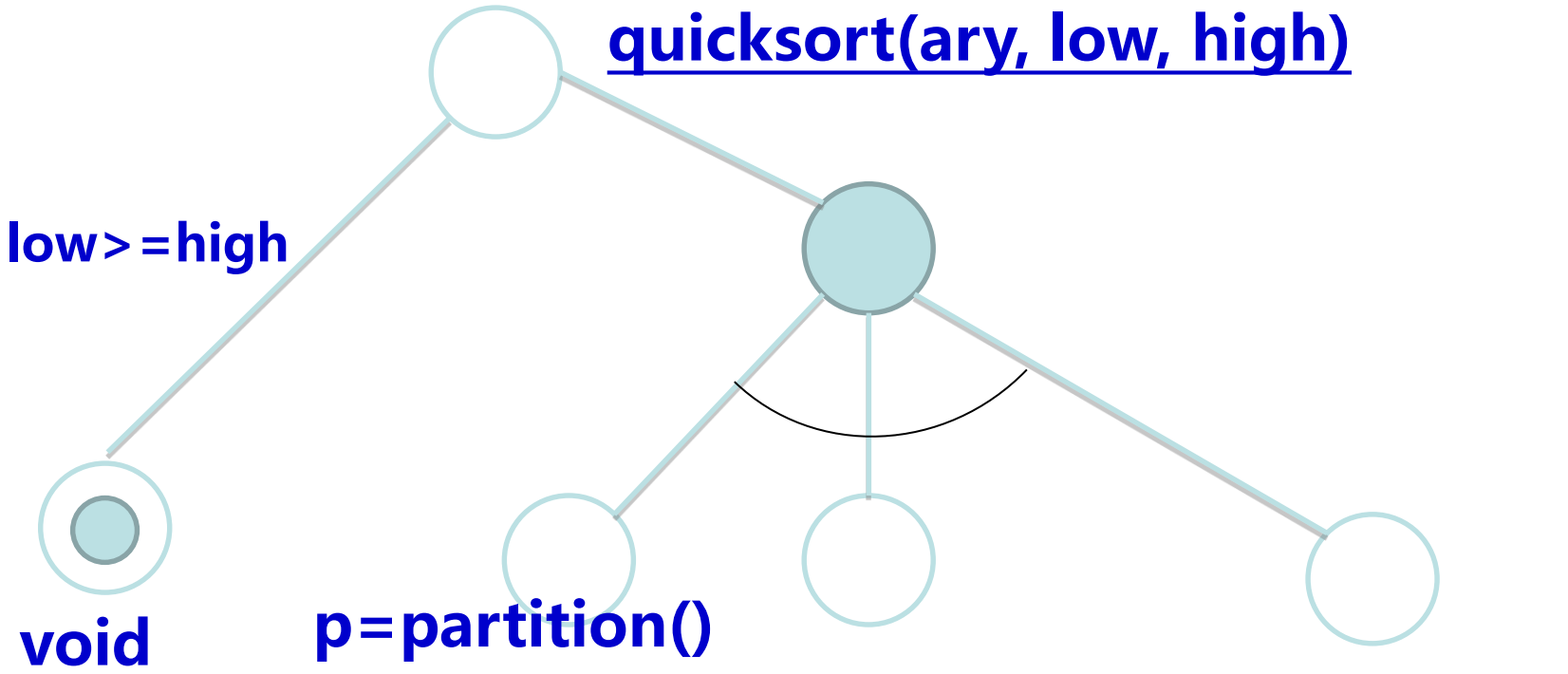
2. 重组织：重新组织数组，使其满足比pivot小的在左侧，大的在右侧，以此分区。
3. 做递归：分别对前后两部分执行上述步骤

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)



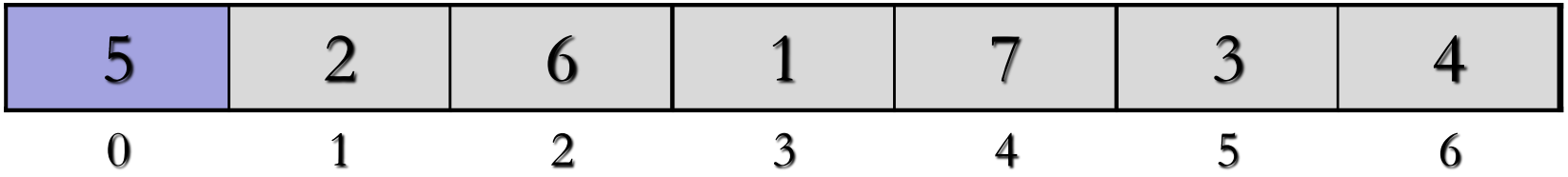
快速排序函数设计

```
void quicksort(int ary[ ], int low, int high) {
  if (low >= high) 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);
}
```

+Chp08_Quicksort(快排).cpp

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

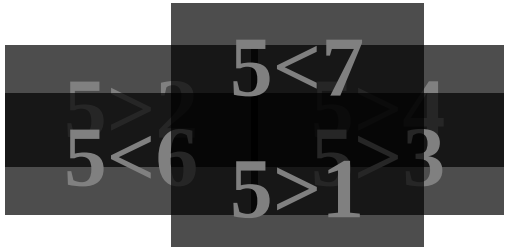


  2. 如何基于pivot进行分区？
     - 目标很明确：比5小的放左边、大的放右边
     - 你如何实现？

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

key



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

partition函数设计 (3)

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

递归搜索：八皇后问题

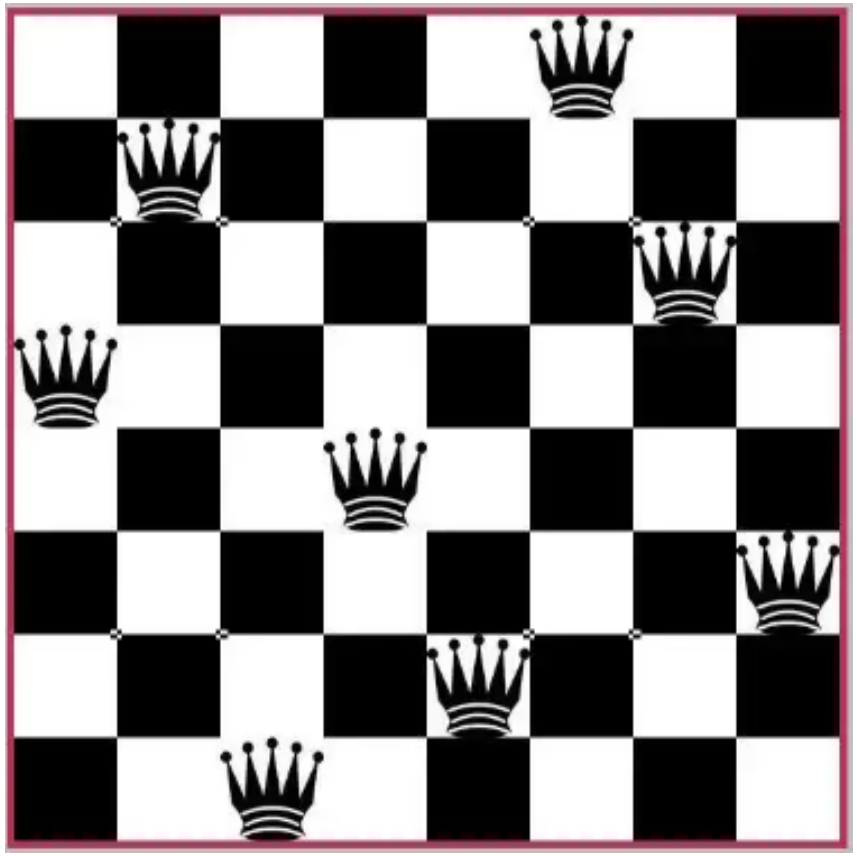
- 在8×8的棋盘上，放置8个皇后（棋子），使两两之间互不攻击。

  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；

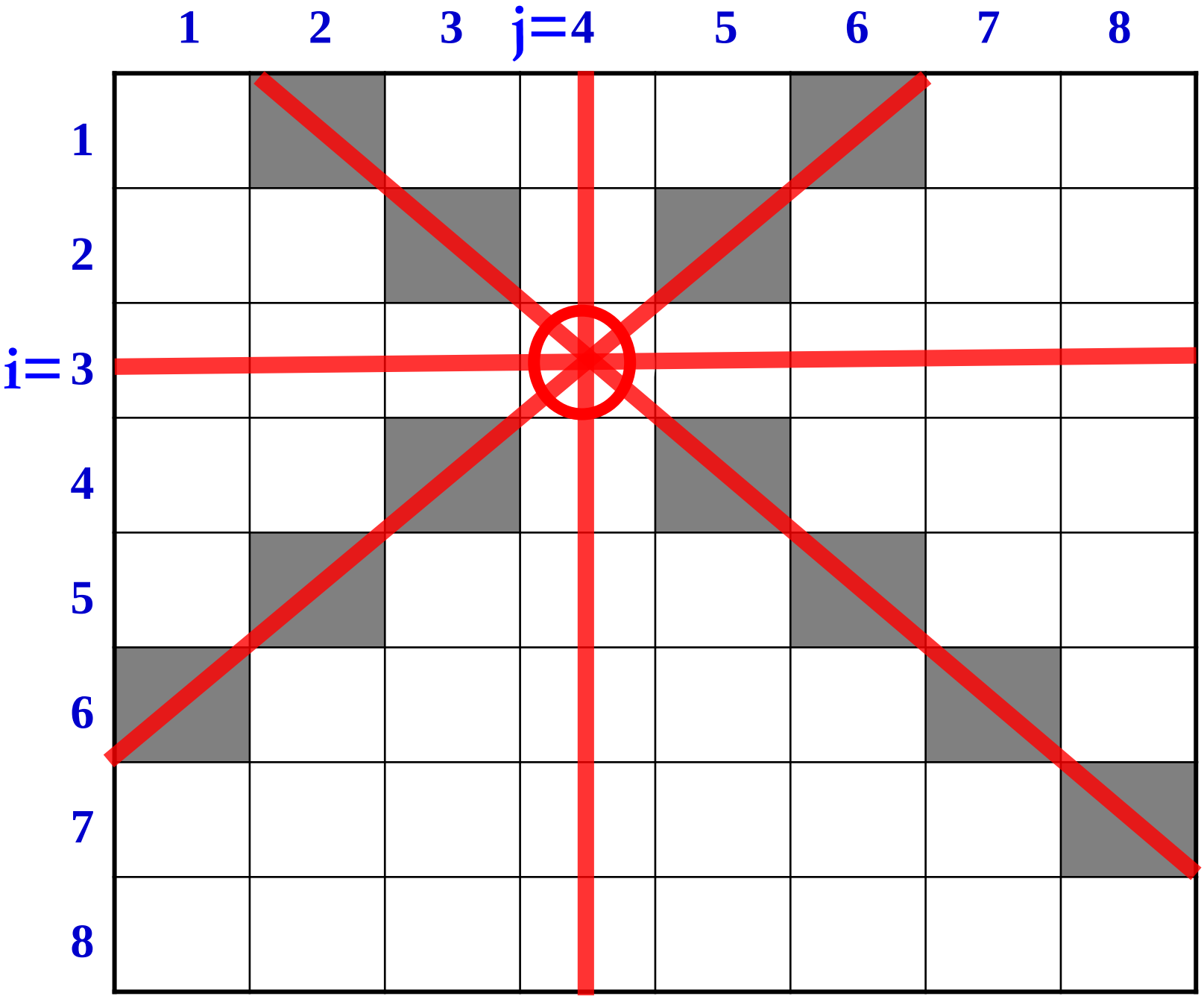
  （2）不在棋盘的同一列；

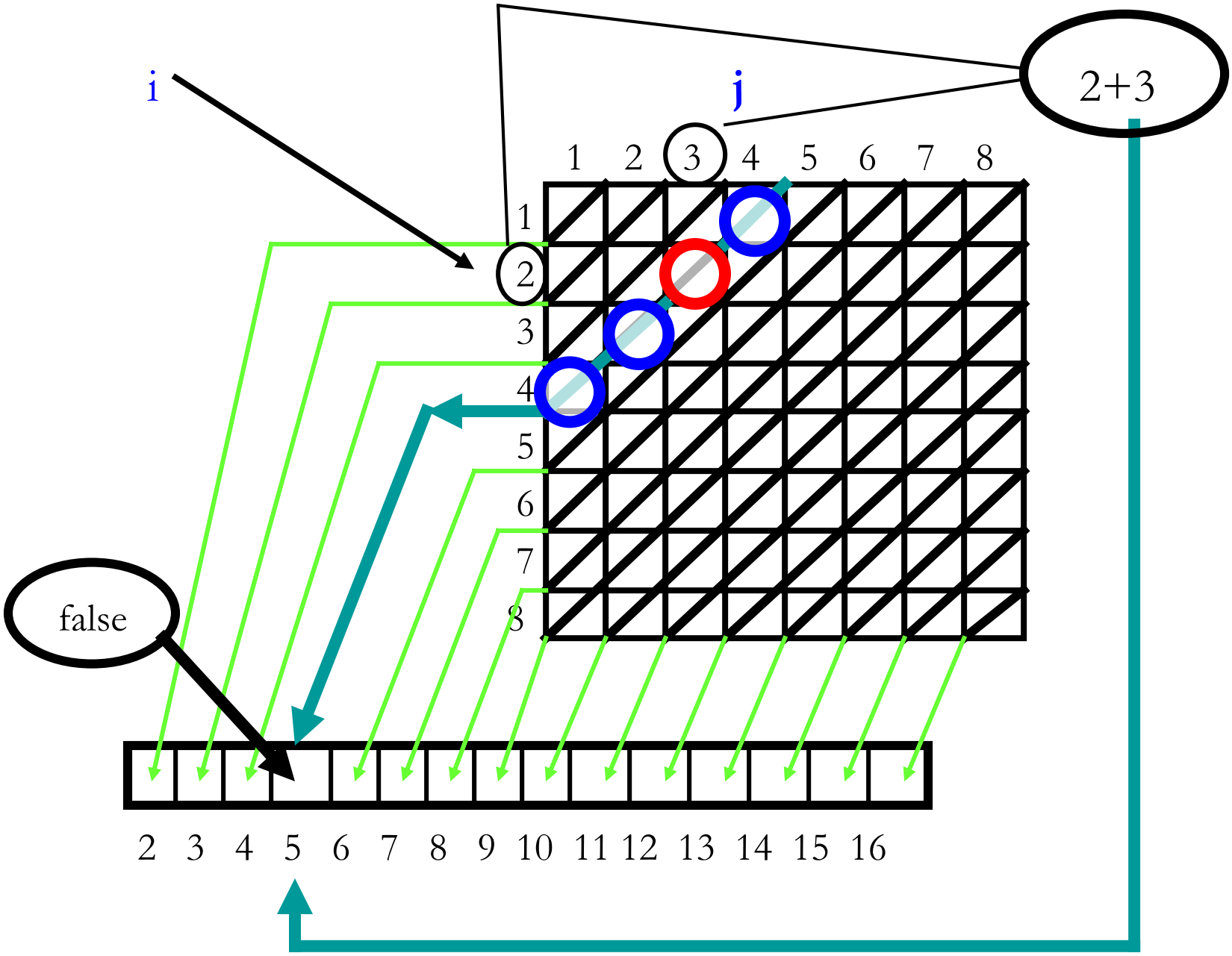
  （3）不在棋盘的同一对角线上。

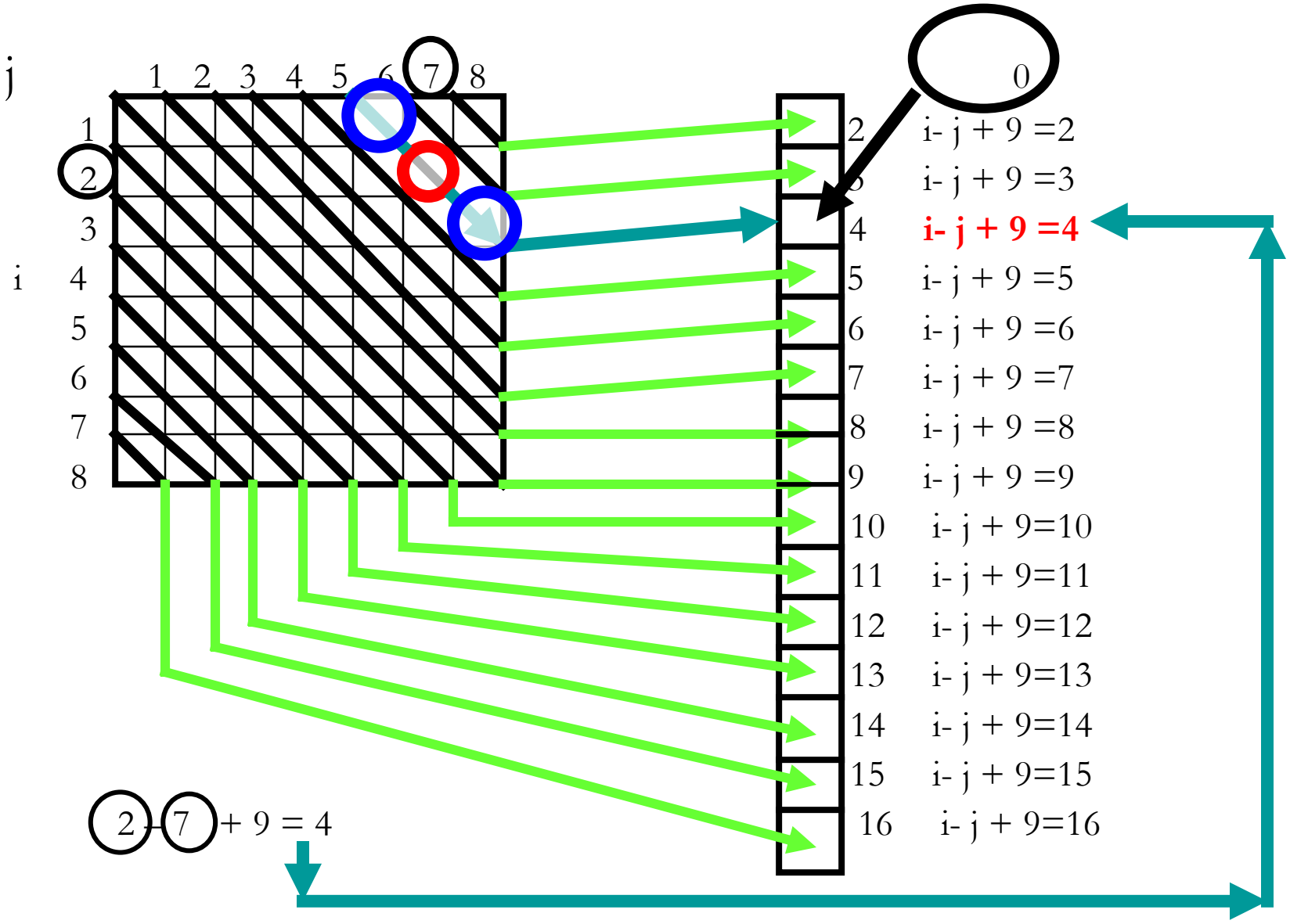


人工放如何放

```
void Try(int i)
{  for (int j=1; j<=8; j++)     // 逐个试每一列
   {  if (如果可以放在第j列)
         第i个皇后
         放在第j列
   }
}
```


1
2
3
j=4
5
6
7
8
i=3
1
2
4
5
6
7
8


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


j
1 2 3 4 5 6 7 8
i
1
2
3
4
5
6
7
8
0
2 i- j + 9 =2
3 i- j + 9 =3
4 i- j + 9 =4
5 i- j + 9 =5
6 i- j + 9 =6
7 i- j + 9 =7
8 i- j + 9 =8
9 i- j + 9 =9
10 i- j + 9=10
11 i- j + 9=11
12 i- j + 9=12
13 i- j + 9=13
14 i- j + 9=14
15 i- j + 9=15
16 i- j + 9=16
2 - 7 + 9 = 4

- 定义　int　L[17]

  L[k]　k= 2,3,…,16

  k = i – j + 9　i = 1,2,…,8　j = 1,2,…,8

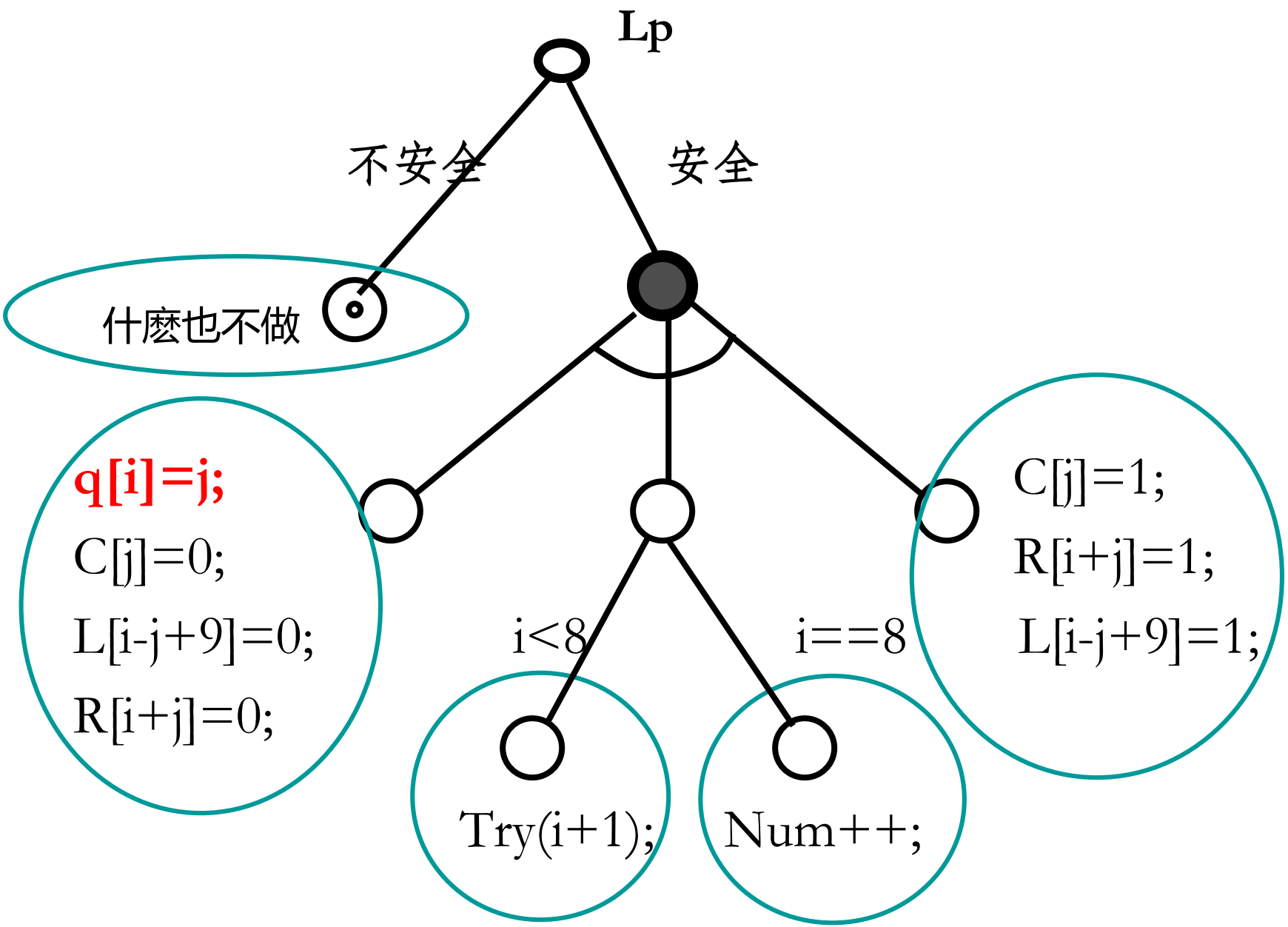
- 数据类型为整型：1----安全　0----不安全

  L[i - j +9] = 0;

  R[i + j] = 0;

- 在 **(i, j)** 位置放皇后的安全条件为:

  C[j]==1　&&　L[i-j+9]==1　&& R[i+j] ==1


Lp
不安全
安全
什麼也不做
q[i]=j;
C[j]=0;
L[i-j+9]=0;
R[i+j]=0;
i<8
i==8
C[j]=1;
R[i+j]=1;
L[i-j+9]=1;
Try(i+1);
Num++;

```
#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)
{   int j;          // 循环变量，表示列号
    int k;
    for (j=1; j<=8; j++)        // 逐个试探每一列
    {   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，放位置(i,j)
    C[j] = 0;          // 修改安全标志
    L[i-j+Normalize] = 0;
    R[i+j] = 0;
    if ( i<8 )         // 第二件事
            Try(i+1);
    else
        {
            Num++;
            printf("方案%d: " , Num);
            for ( k=1; k<=8; k++)
                    printf("%d ", q[k]);
            printf ("\n" );
        }
    C[j] = 1;   // 第三件事，修改安全标志
    L[i-j+Normalize] = 1;
    R[i+j] = 1;
}
```

第i个皇后放在第j列

+Chp08_八皇后.cpp

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

int main()
{
    int i;
    Num = 0;                    // 方案数清零
    for(i=0; i<9; i++)
        C[i] = 1;

    for(i=0; i<17; i++)
        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

...

```
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+9] == 1 ) {
                q[i] = j;       // 第一件事，占用位置(i,j)
                C[j] = 0;       // 修改安全标志
                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
                L[i - j + Normalize] = 1;
                R[i + j] = 1;
                }
}
```

```
void Try(int i)
{
    是否到站
    for (int j=1; j<=N; j++) // 逐个每一种情况
    {   if 第j位置是否可放
            标记占第j位置
            Try(i+1)
            释放第j位置
    }
}
```

总结

```
void Try(int i)
{    for (int j=1; j<=N; j++)   // 逐个试每一种情况
     {  if 第j位置是否可放
            标记占第j位置
            Y：是否到站
            N：Try(i+1)
            释放第j位置
     }
}
```

总结

#124 滑雪

小袁非常喜欢滑雪，因为滑雪很刺激。为了获得速度，滑的区域必须向下倾斜，而且当你滑到坡底，你不得不再次走上坡或者等待升降机来载你。小袁想知道在某个区域中最长的一个滑坡。区域由一个二维数组给出。数组的每个数字代表点的高度。如下：

1	2	3	4	5
16	17	18	19	6
15	24	25	20	7
14	23	22	21	8
13	12	11	10	9

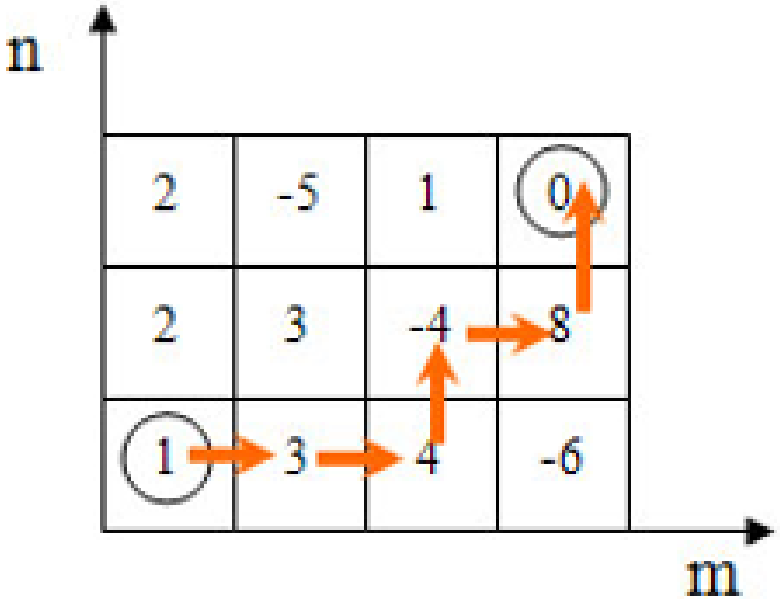
4个方向探索

```
for(i=1;i<=r;i++)
  for(j=1;j<=c;j++)
      Try(i, j, 0);
```

```
void Try( int x, int y, int length )
{
  if(超过边界) return;
  length++;
  if(length>max) max=length;
  if( *** )  Try(x-1, y,   length );
  ...
}
```

#126 Travel

有一个 n*m 的棋盘，如图所示，骑士 X 最开始站在方格**(1,1)**中，目的地是方格**(n,m)**。他的每次都只能移动到**上、左、右**相邻的任意一个方格。每个方格中都有一定数量的宝物 k（可能为负），对于任意方格，骑士 X 能且只能经过最多 1 次（因此从(1,1)点出发后就不能再回到该点了）。



你的任务是，帮助骑士 X 从(1,1)点移动到(n,m)点，且使得他获得的宝物数最多。

```
void Try(int x, int y) {
    // 第 1 步 探索
    mine += a[x][y];
    used[x][y] = 1;
    value[step] = a[x][y];
    step++;
    // 第 2 步 判断+递归
    if (x == n && y == m)
      {          }
    else
      {          }
    // 第 3 步 回溯
    mine -= a[x][y];
    used[x][y] = 0;
    step--;
}
```

```
int m, n;
int a[9][9], used[9][9] = {0};
int mine = 0, best_mine = a[1][1];
int step = 1, best_step;
int value[100], best_value[100];
```

Try(1, 1);

#278 爬楼梯

假如你正在爬一个有 n 个台阶的楼梯，一次只能爬 1 个台阶或者 2 个台阶，请问一共有多少种爬到顶端的走法？

【输入格式】

□□一个整数 n（n <= 30）

【输出格式】

□□一个整数，代表走法的数量

【输入样例】

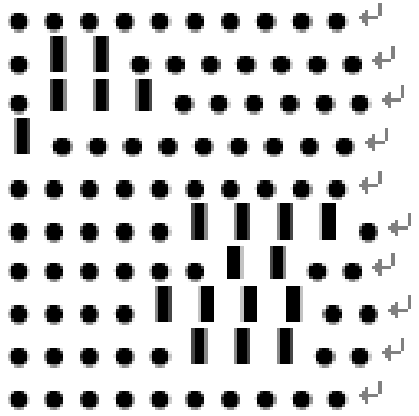
2

【输出样例】

2

#163 最大岛屿

A 国是个岛国，由若干个小岛组成。A 国地图如下所示：



其中“.”表示水，“I”表示陆地，相邻的两个“I”是同一个岛屿上的。本题中一个点与它周围的 8 个点都是相邻的。例如：上图中有 2 个岛屿，右下这个岛屿是最大的。

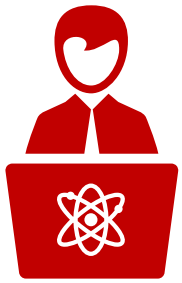
请编程找出 A 国有几个岛屿，其中最大的岛屿面积的是多少（即最大岛屿中一共包含多少个“I”）？

上下左右及四个角

8个方向探索

```
void search(int x,int y){
    if(x<0||x>=n||y<0||y>=m||map[x][y]=='.') return;
    area++;  map[x][y]='.';
    search(x+1,y);  search(x-1,y);  search(x,y+1);  search(x,y-1);
    search(x+1,y+1); search(x+1,y-1); search(x-1,y+1); search(x-1,y-1);
}
```

中国人民大学
RENMIN UNIVERSITY OF CHINA



谢谢大家！