中国人民大学
RENMIN UNIVERSITY OF CHINA

第8讲 递归(2)

余力

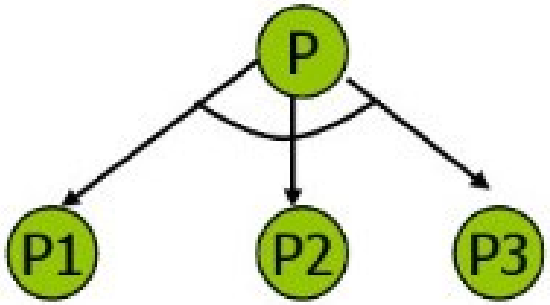
buaayuli@ruc.edu.cn

中国人民大学
RENMIN UNIVERSITY OF CHINA

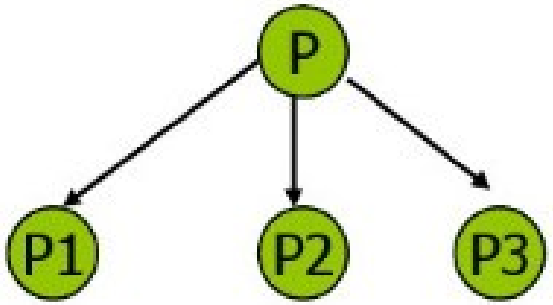
4. 递归分析工具-与或图

与或图

- **与图**：把一个原问题**分解**为若干个子问题，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”。



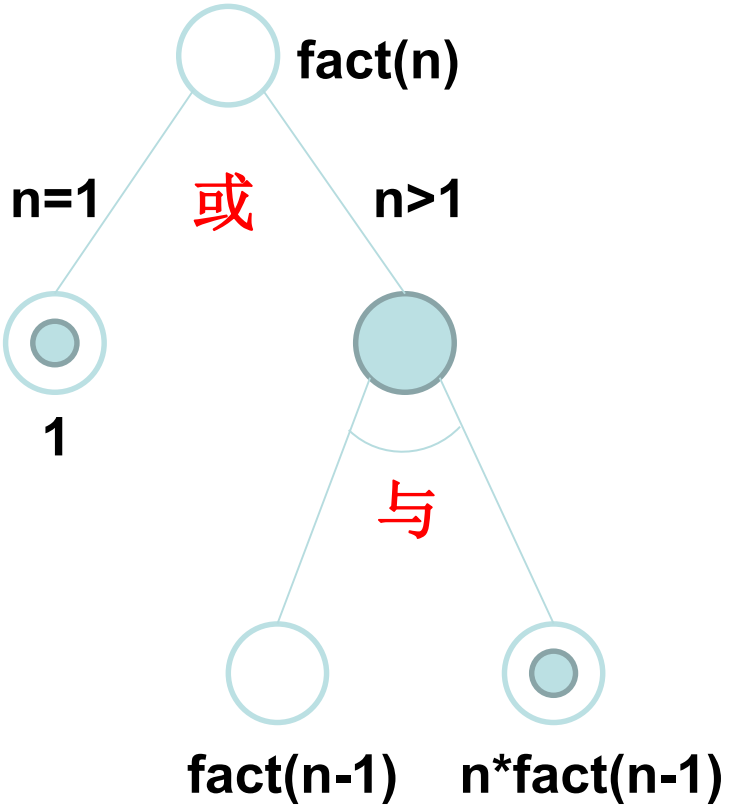
与



或

阶乘

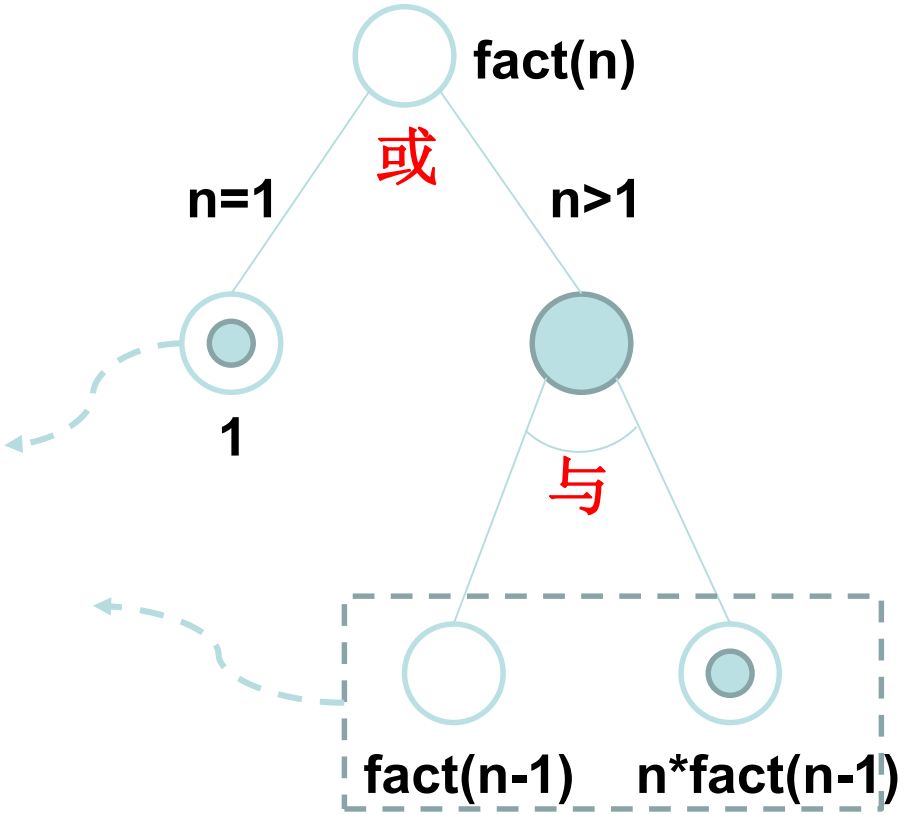
- 阶乘n!的计算
  - Base case：
    - n=1时，返回结果1
  - Inductive case:
    - n>1时，返回n!=n*(n-1)!



如何用图表形式表示?

与或图与递归程序编写

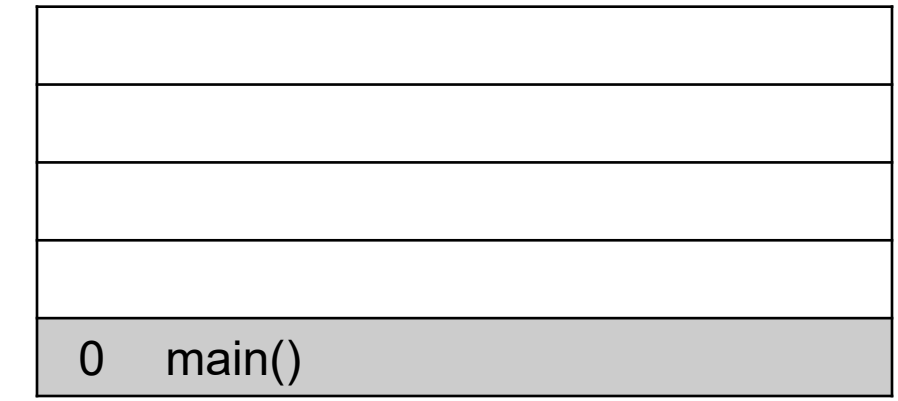
```
#include<iostream>
using namespace std;
int fact (int n) {
    if (n==1) return 1;
    else {
        int fn1 = fact(n-1);
        return n*fn1;
    }
}
int main () {
    cout << fact(3);
}
```



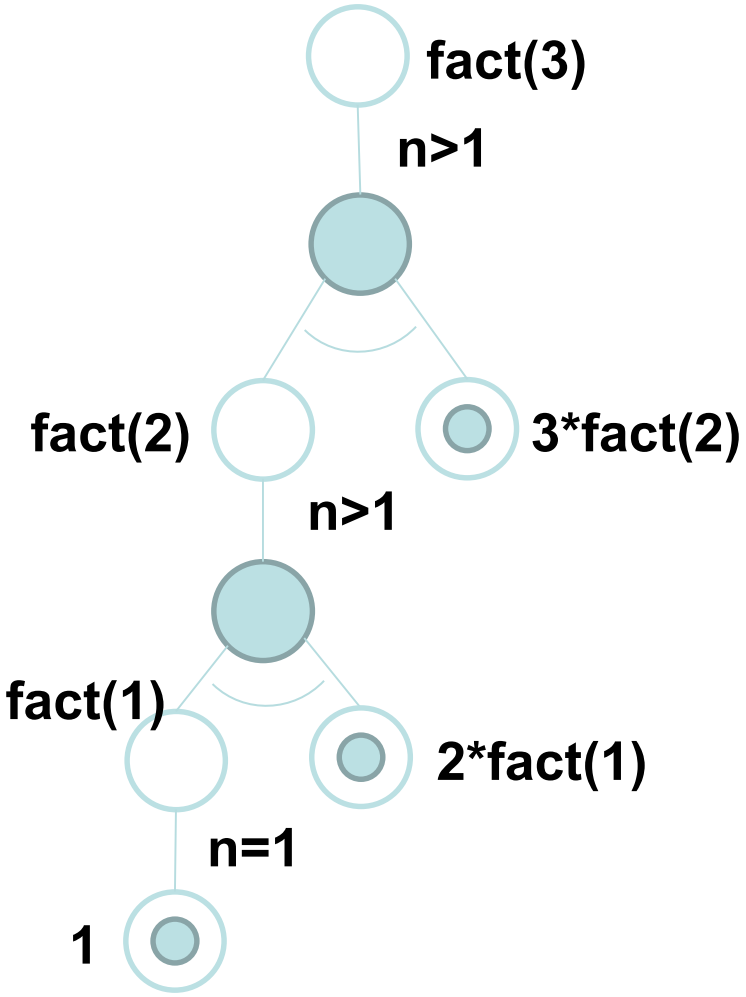
Fact阶乘.cpp

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)

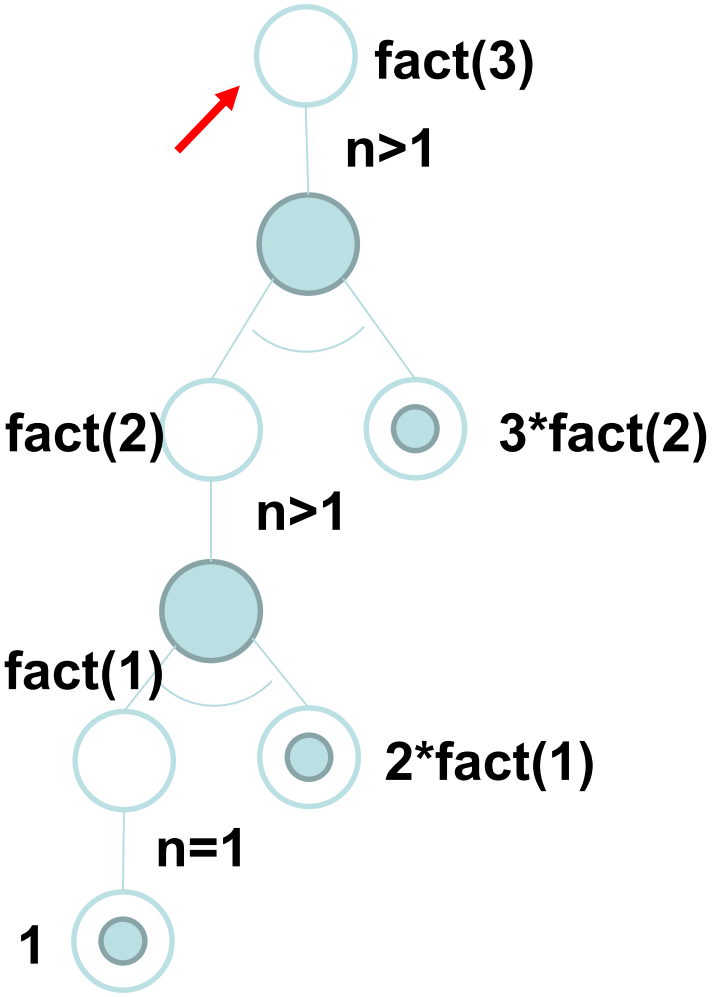


与或图

```
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)

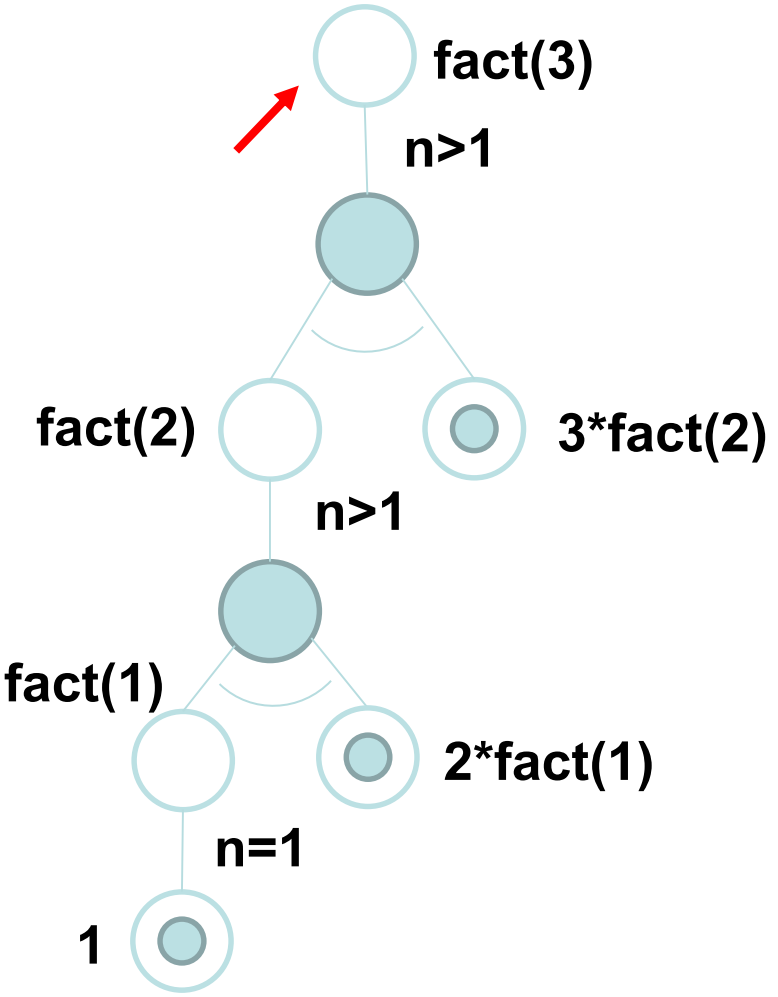


与或图

```
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)

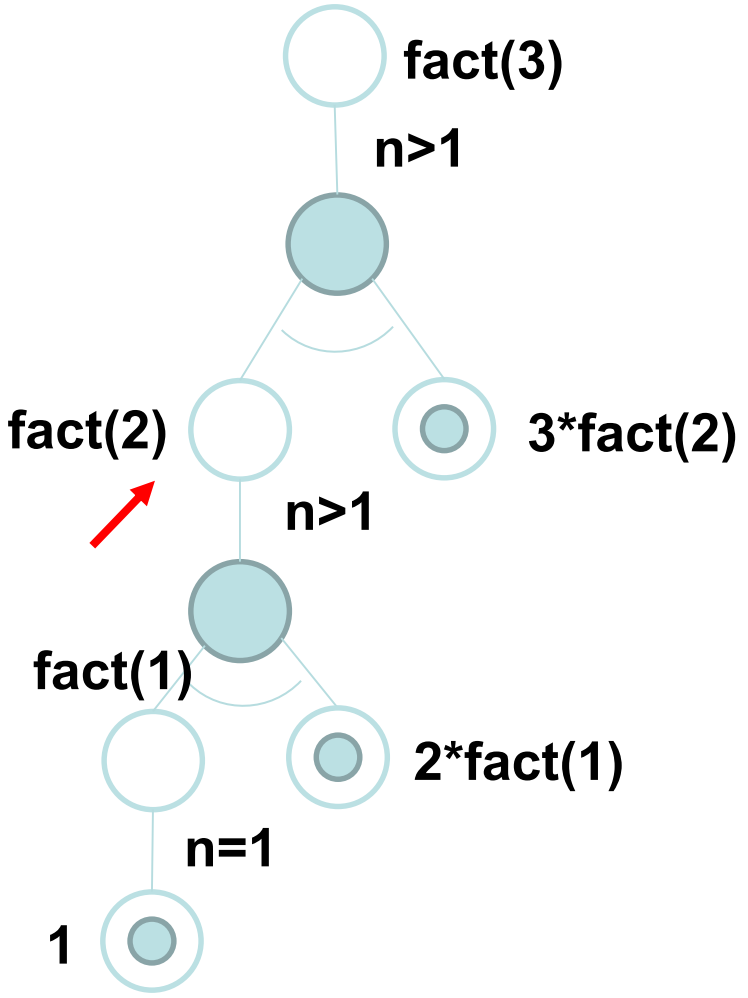


与或图

```
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)

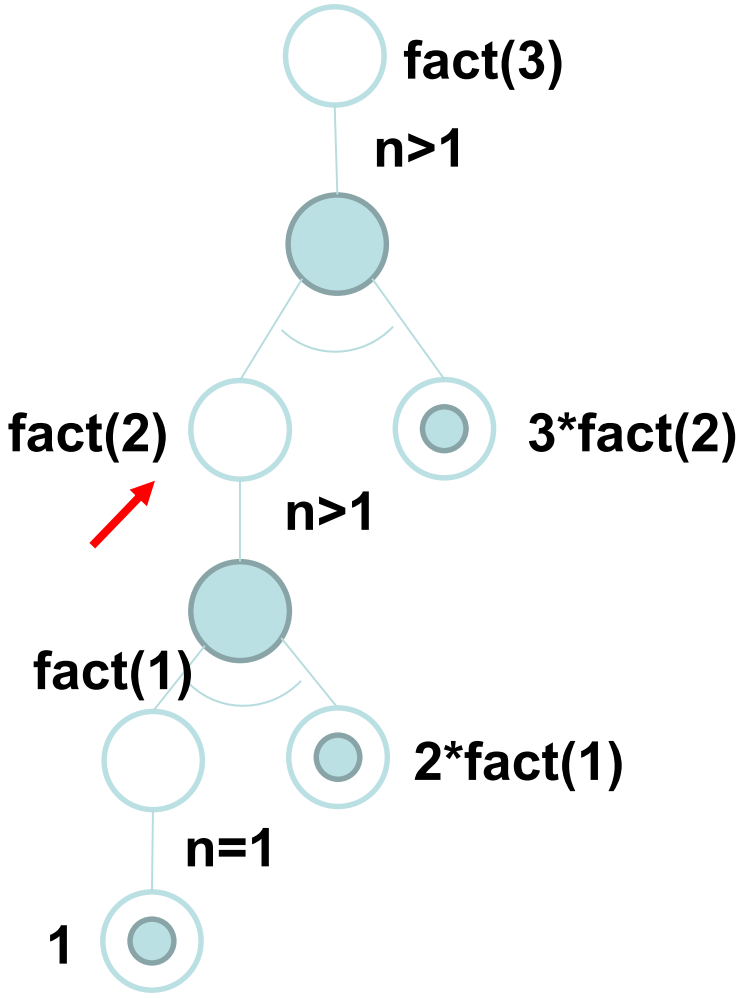


与或图

```
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)

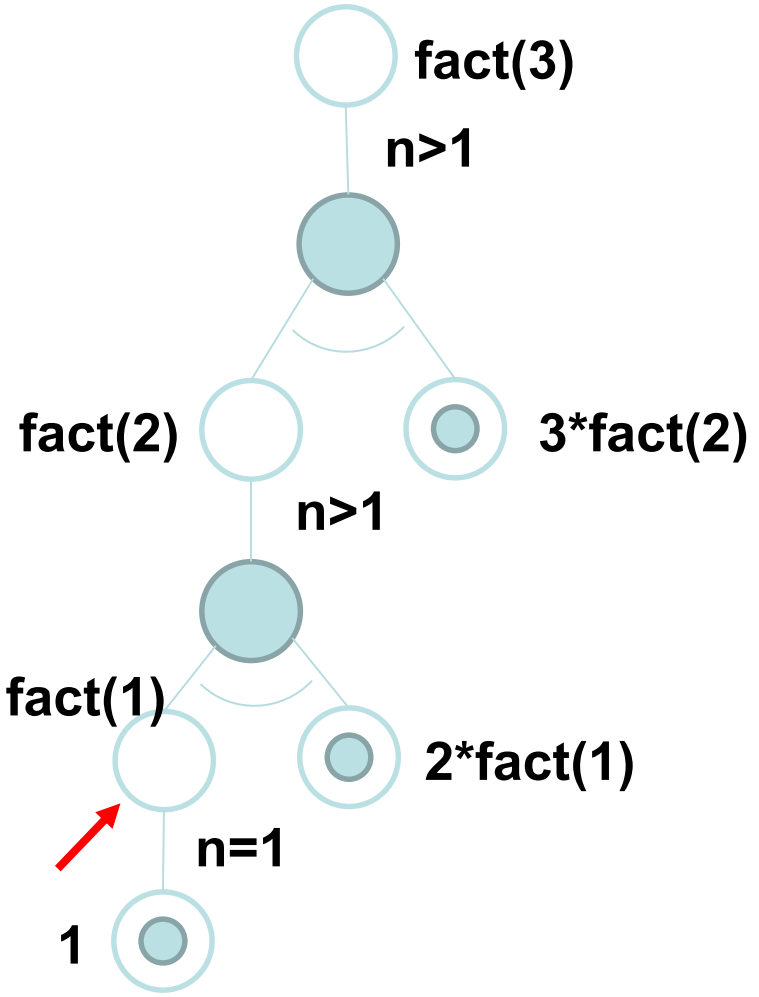


与或图

```
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)

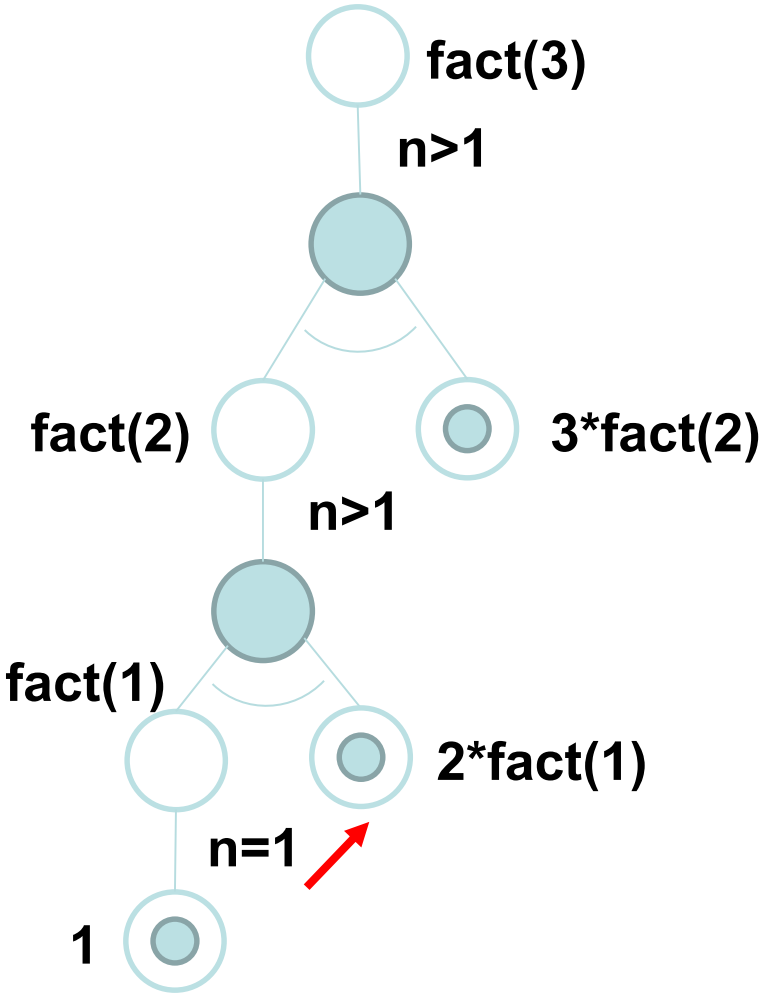


与或图

```
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)

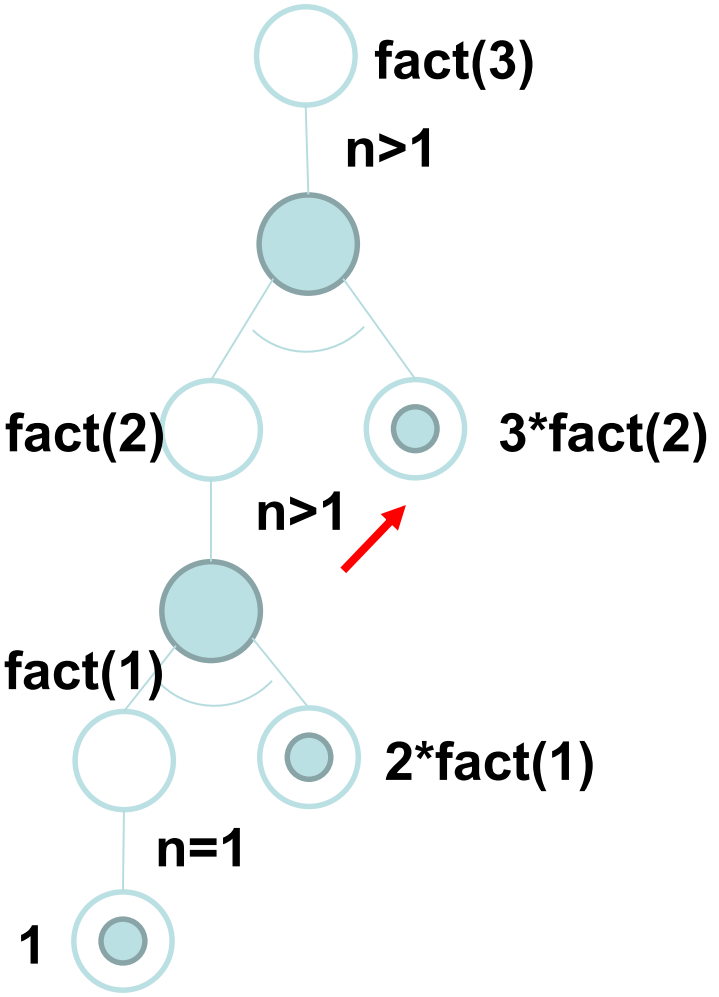


与或图

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

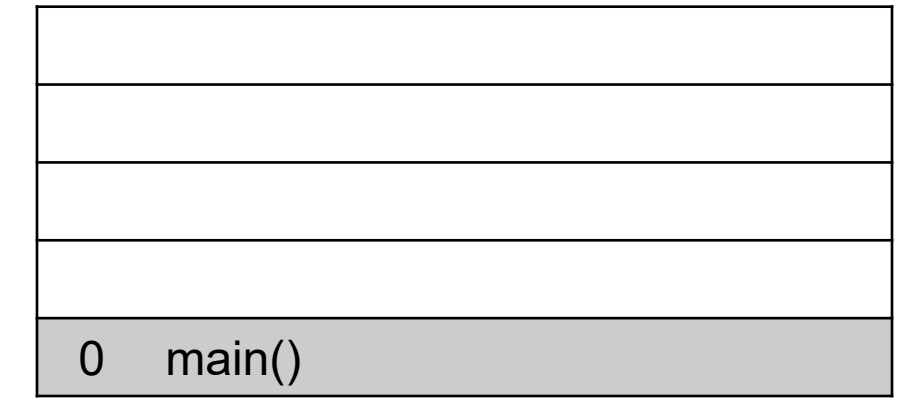
1	fact(3)
0	main()

函数调用栈(Call Stack)

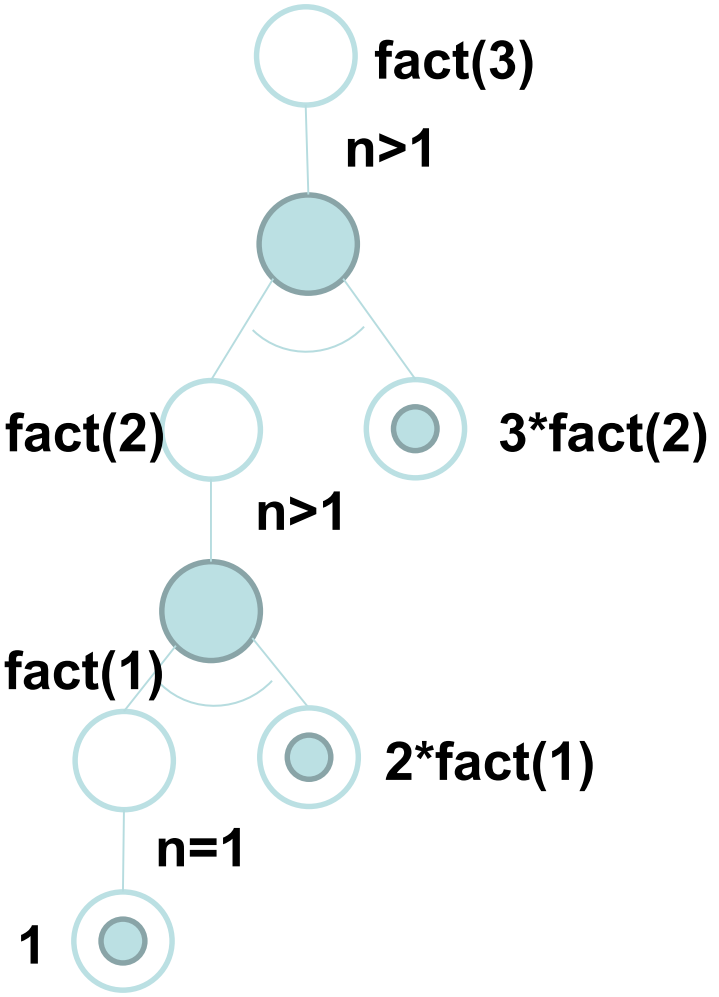


与或图

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



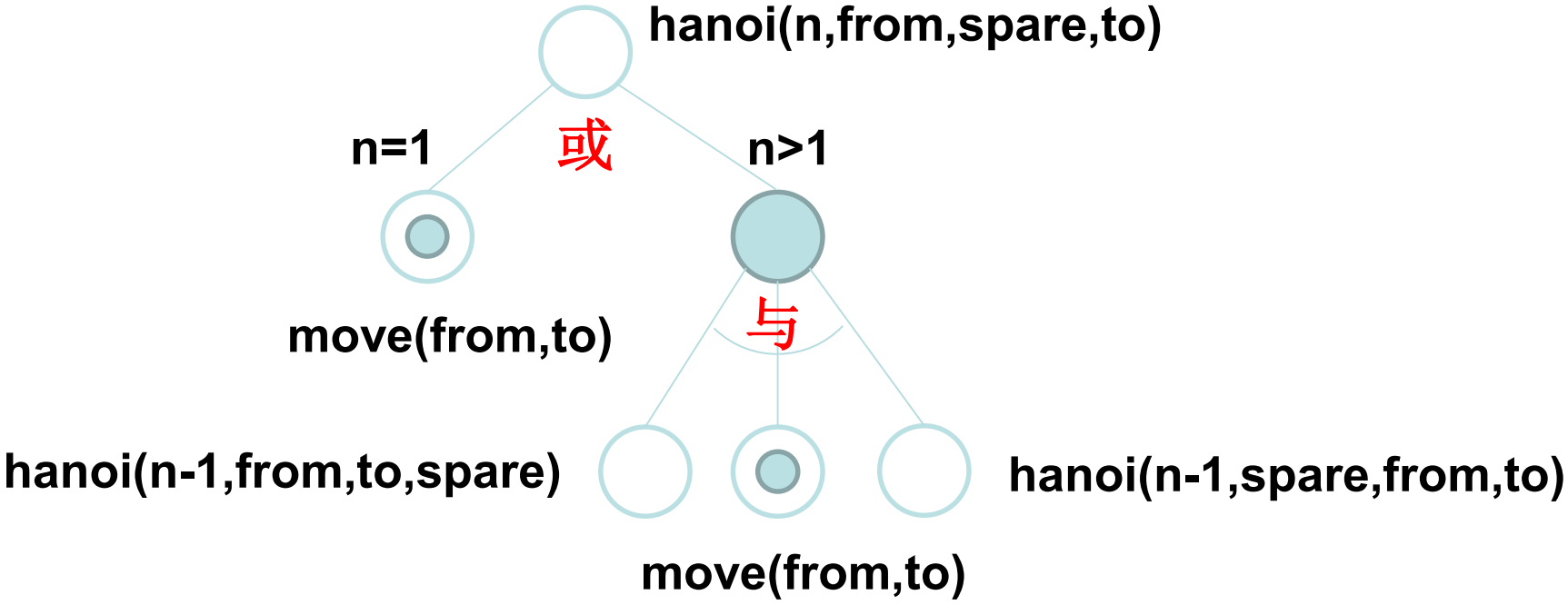
函数调用栈(Call Stack)



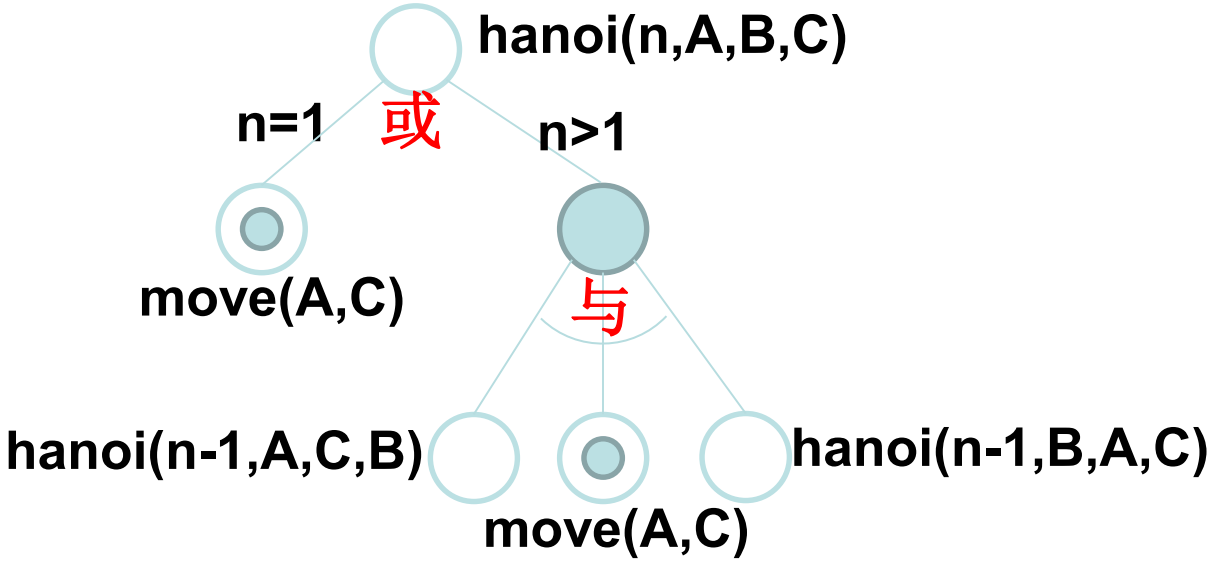
与或图

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

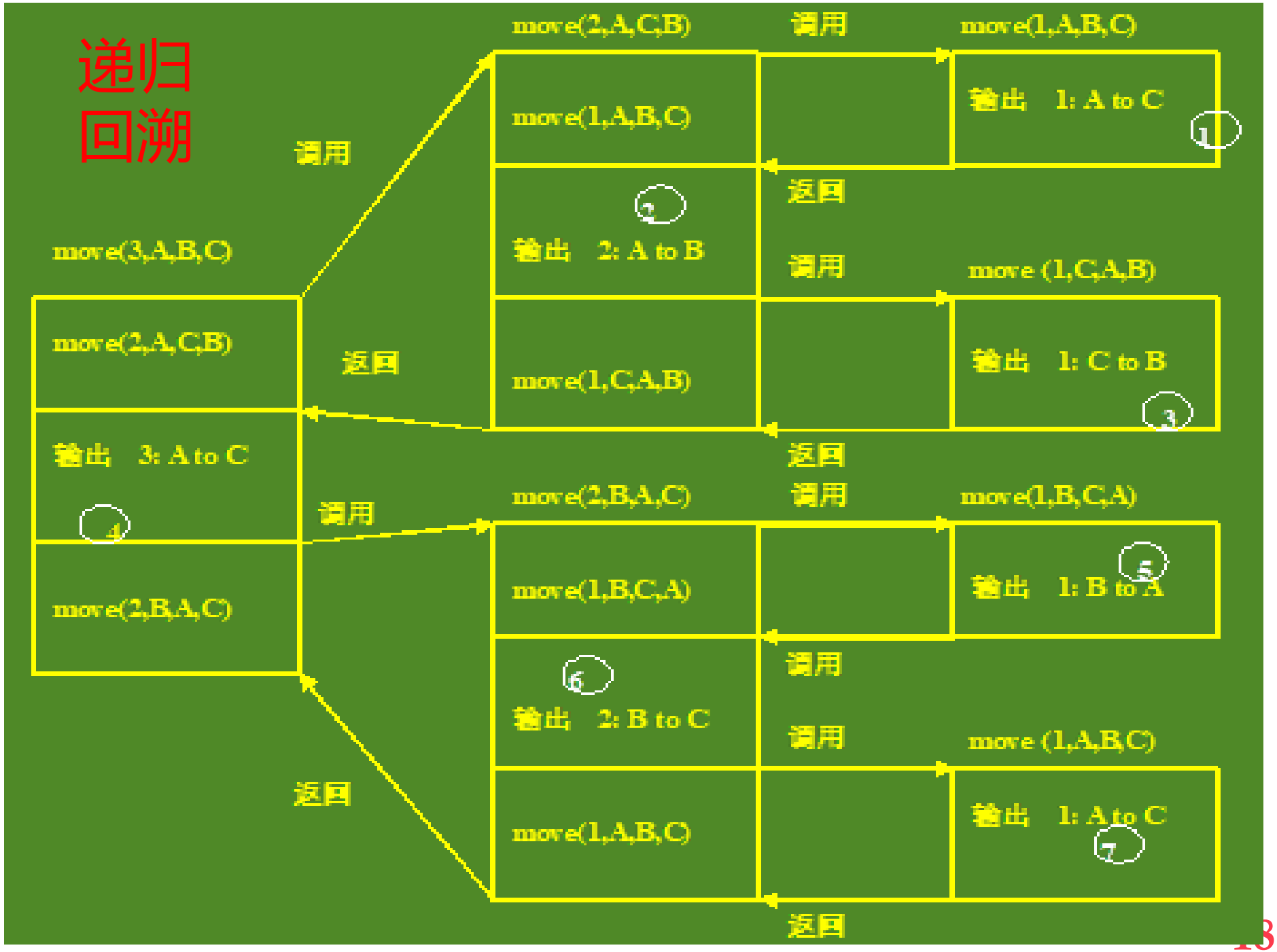



```
void move (int n, char A, char B, char C) {
    if (n == 1)
        cout << "move from " << A << " to " << C << endl;
    else {
        move (n-1, A, C, B);
        cout << "move from " << A << " to " << C << endl;
        move (n-1, B, A, C);
    }
}
int main () {
    int n = 4;
    char A = 'A',B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

Hanoi汉诺塔.cpp

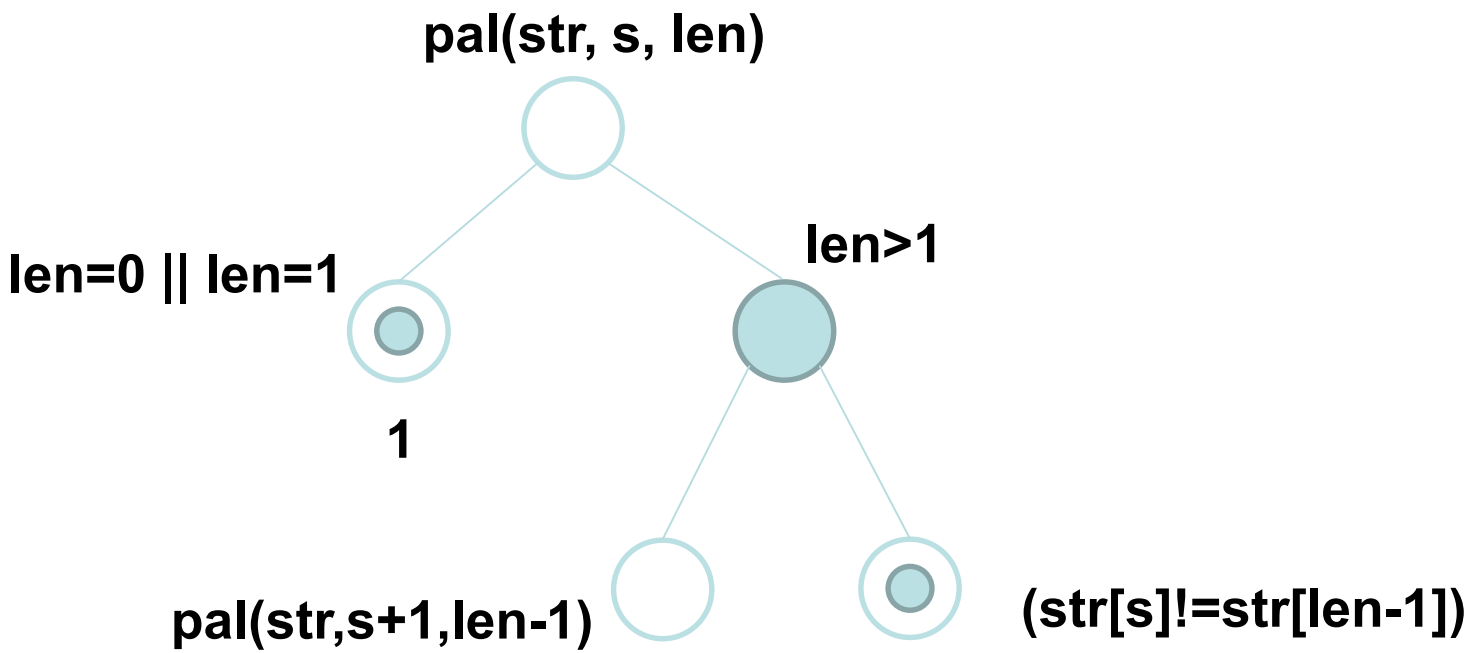
```
#include <stdio.h>
int main()
{  void hanoi(int n, char one, char two, char three);

    int m;
    printf("the number of diskes:");
    scanf("%d",&m);
    printf("move %d diskes:\n",m);
    hanoi(m,'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);
Count++;
 }
```


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

回文判断

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



判断回文

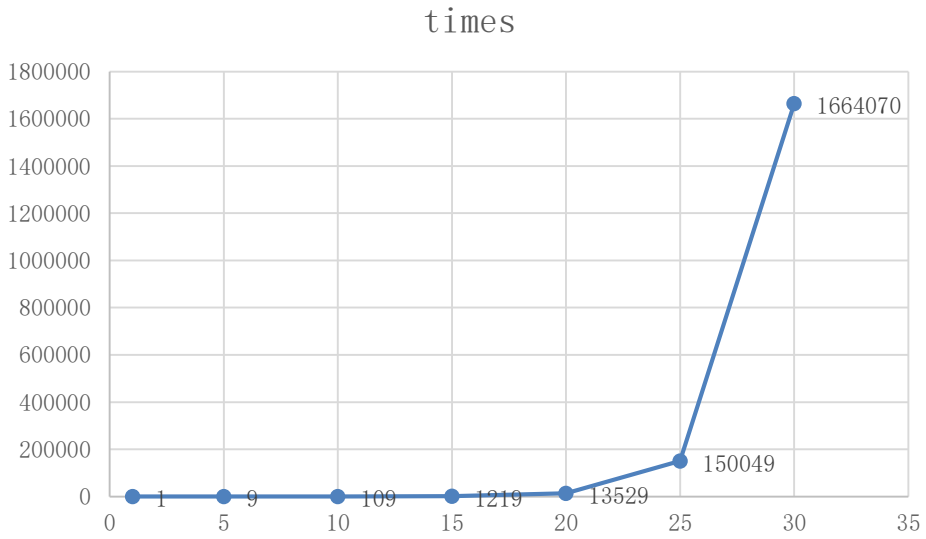
```
#include<iostream>
using namespace std;
#include<string.h>
int pal(char str[], int low, int high) {
    if(high<=low)
        return 1; // base case
    else
        if (str[low]!=str[high]) return 0;
        else return pal(str,low+1,high-1);

}
int main() {
    char str[100] = "madamimadam";
    int len = (int)strlen(str);
    cout << pal(str,0,len-1) << endl;
}
```

```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;
}
```

```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

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

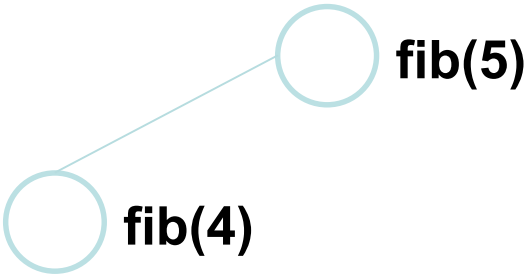


指数级增长！

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”

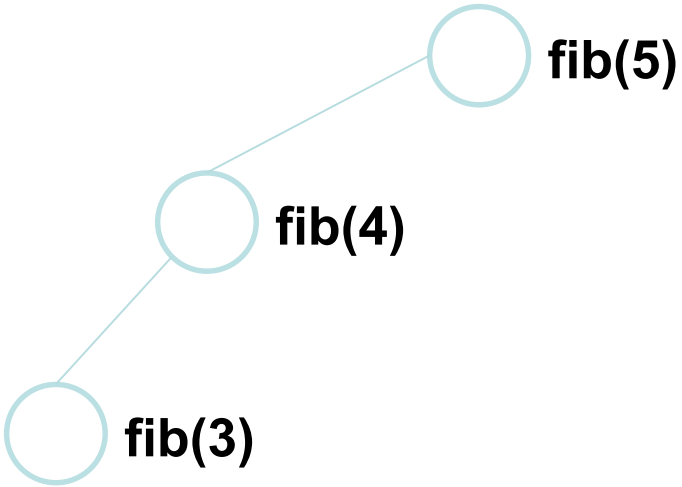


memo数组

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”

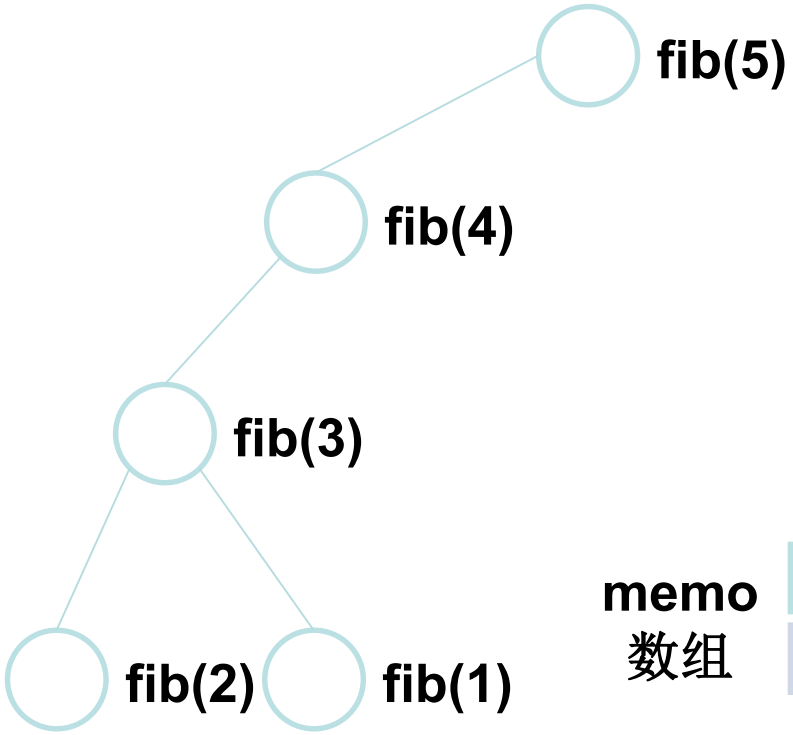


memo数组

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”

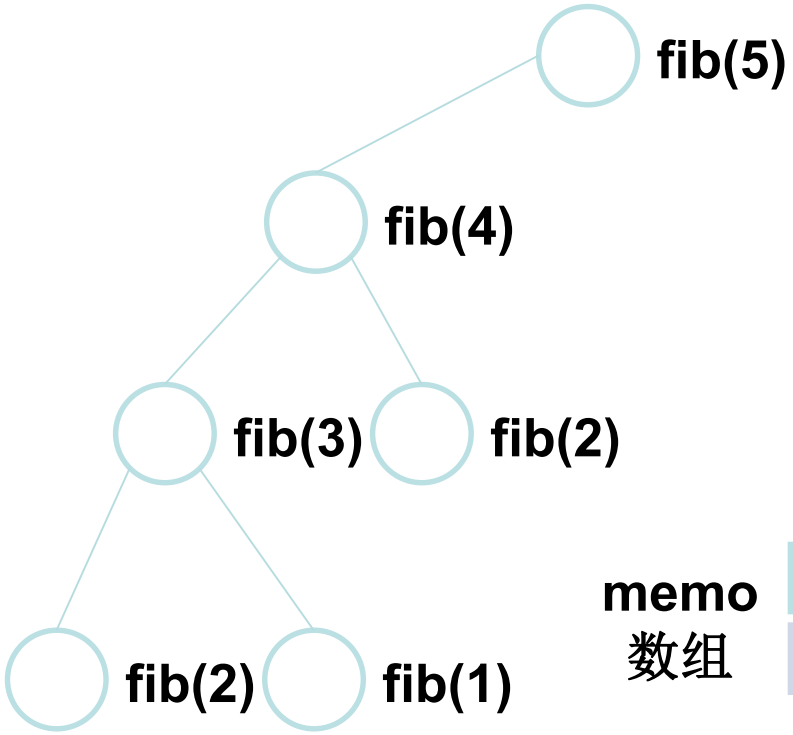


memo数组

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”

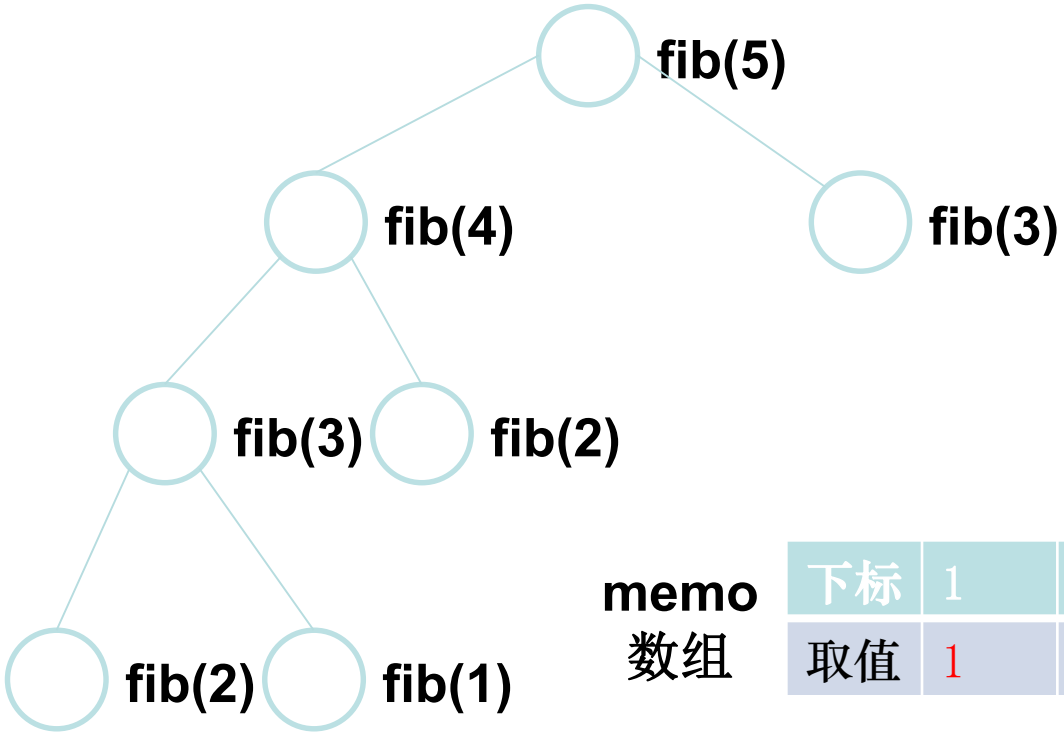


memo数组

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

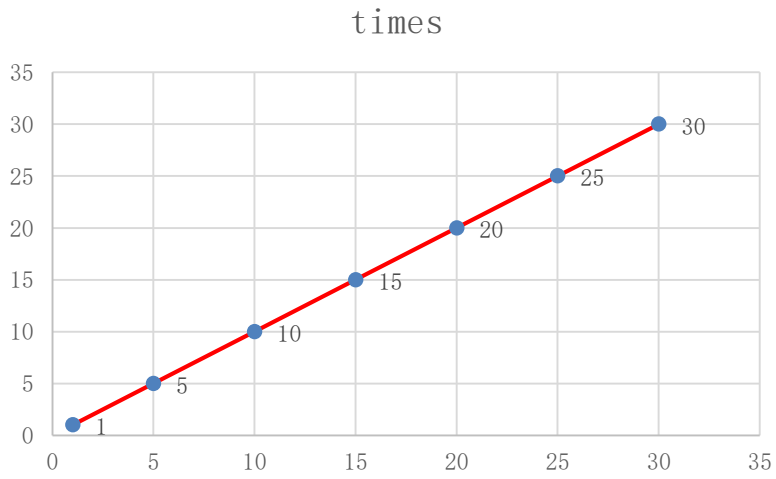
- 分析memo数组的作用
  - 将已经计算过的“中间结果”存起来
  - 直接使用存好的“中间结果”
  - 避免了重复计算

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];
    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];
    for (int i = 0; i < 100; i ++) memo[i]=-1;
    cout << fib(5,memo) << endl;
    cout << "times: " << times << endl;
}
```

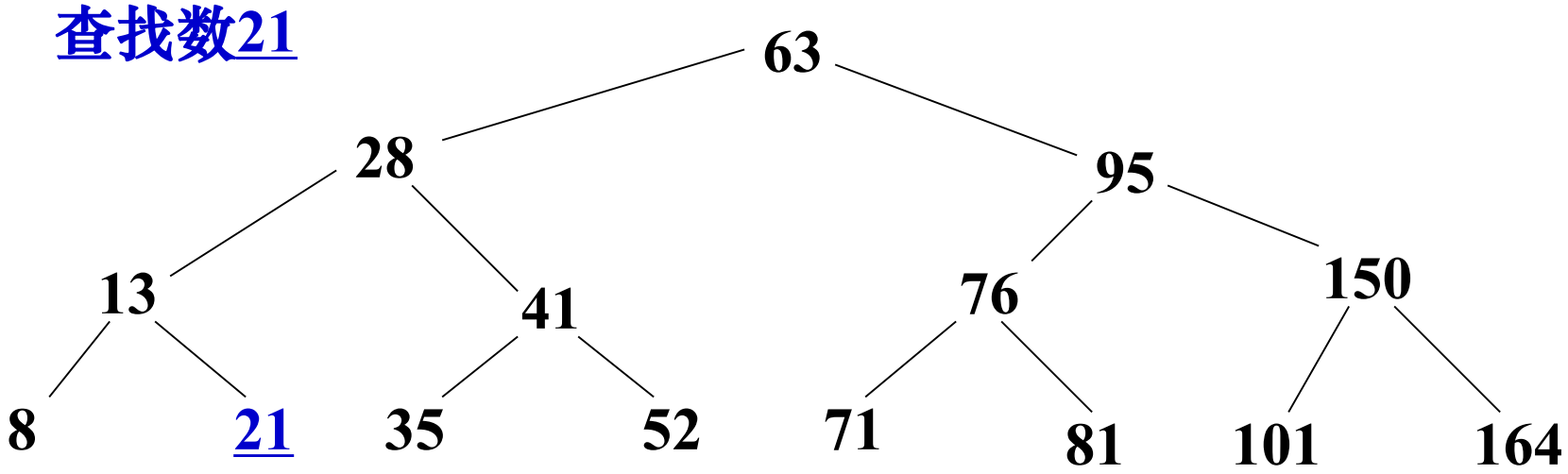


中国人民大学
RENMIN UNIVERSITY OF CHINA

5. 查找排序与递归

二分查找

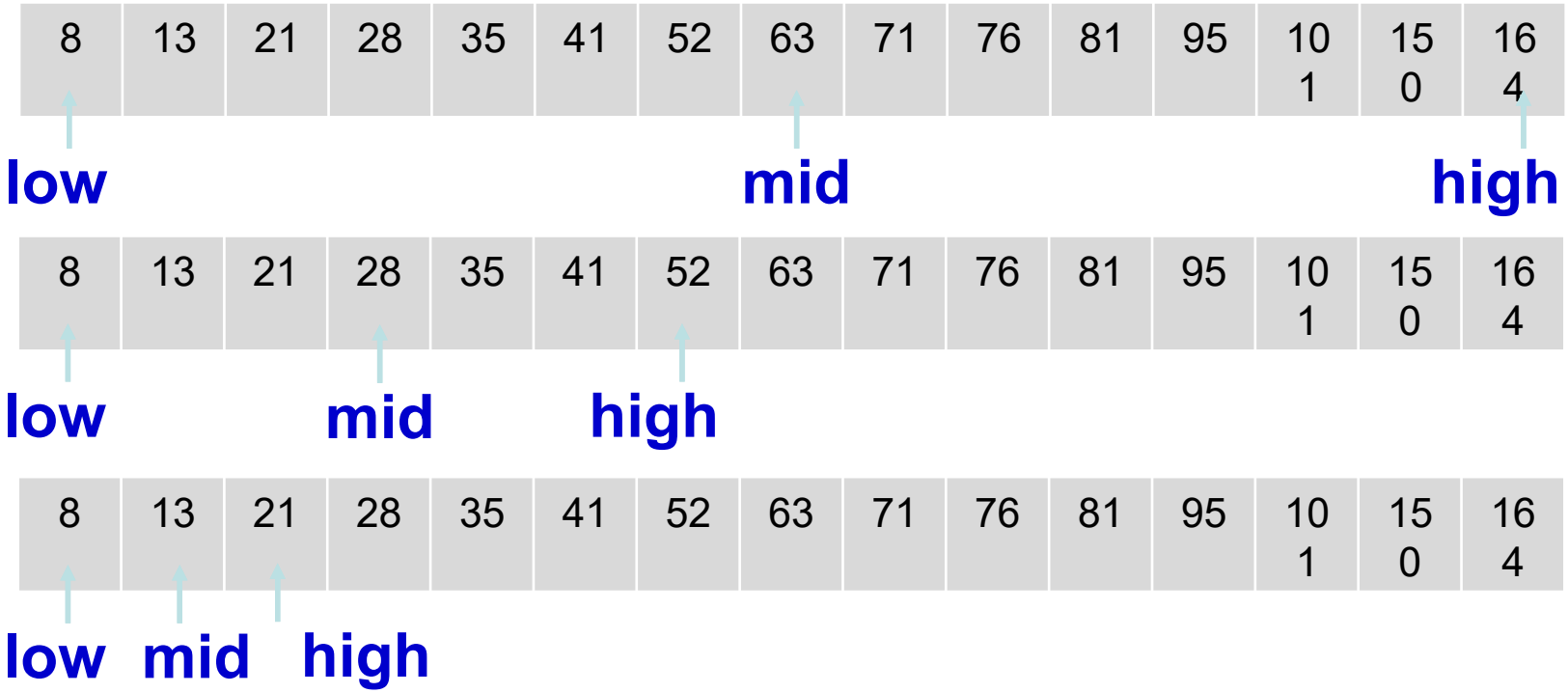
- 如果把排好序的数据看成一棵树……



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



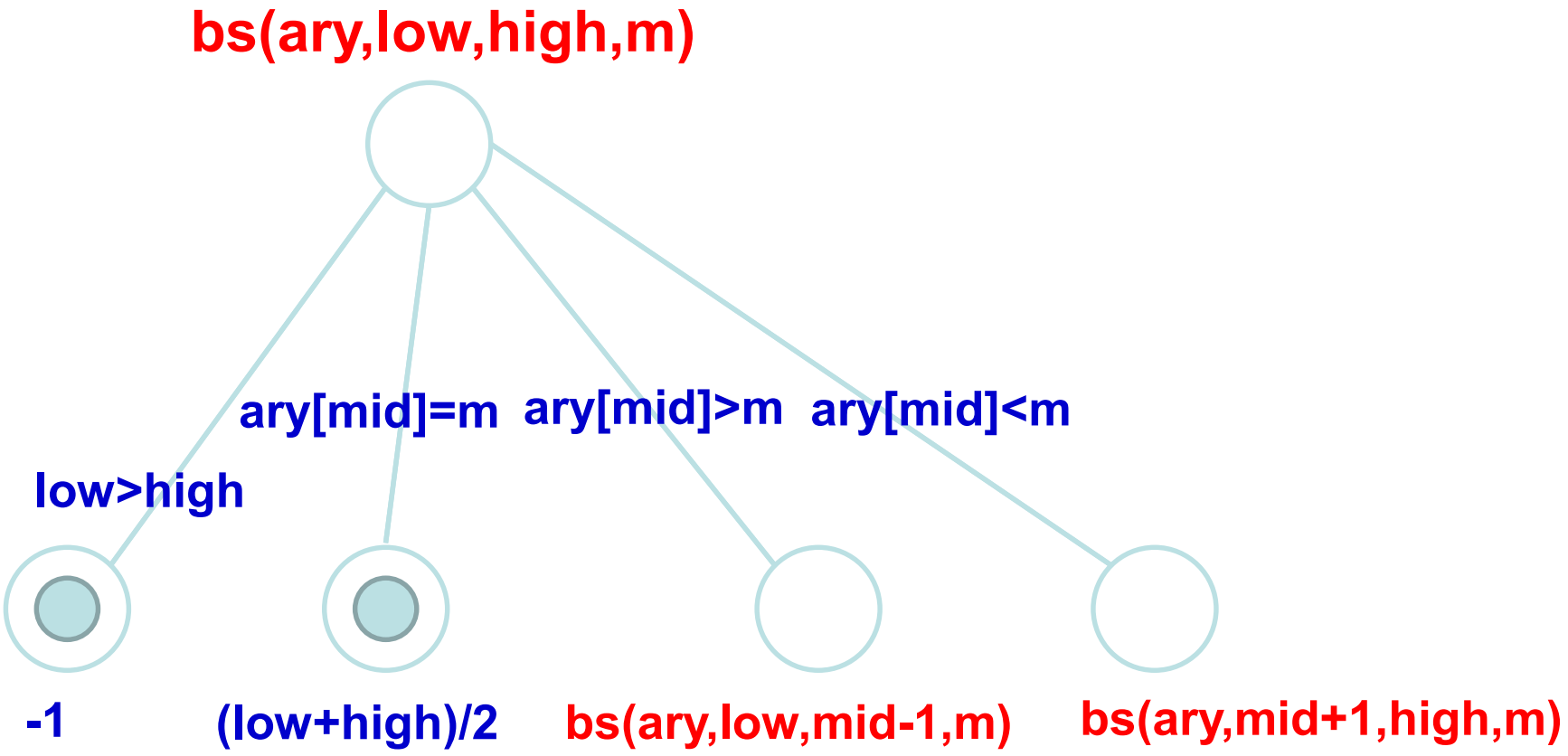
二分查找的循环实现 (2)

```
int 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
}
```

二分查找的循环实现 (3)

- 边界条件
  - 可能找到：low <= 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)


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)

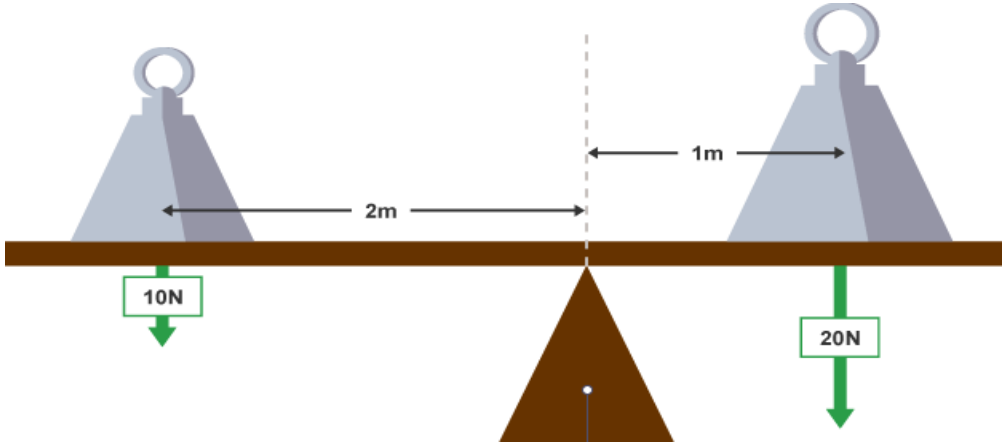
```
int bisearch(int ary[], int low, int high, int m) {
  if (low > high)          //没找到
        return -1;
  int mid = (low + high) / 2;
  if (ary[mid] == m)  //找到
        return mid;
  else if (ary[mid] > m)  //找左半边
         return bisearch(ary, low, mid - 1, m);
  else  //找右半边
        return bisearch(ary,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

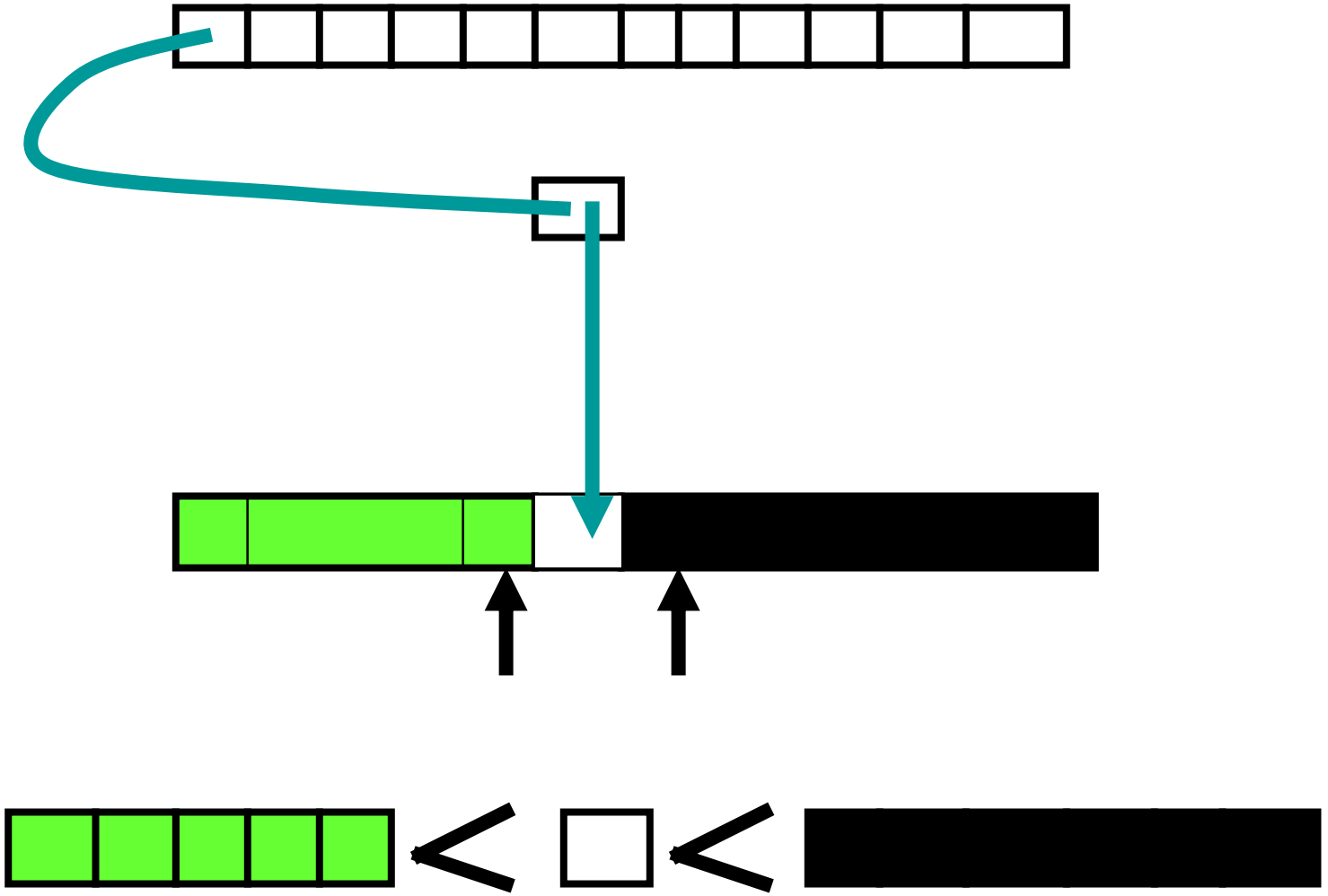
1. 选枢纽：从数组中选择出一个元素，称之为枢纽pivot元素



分区

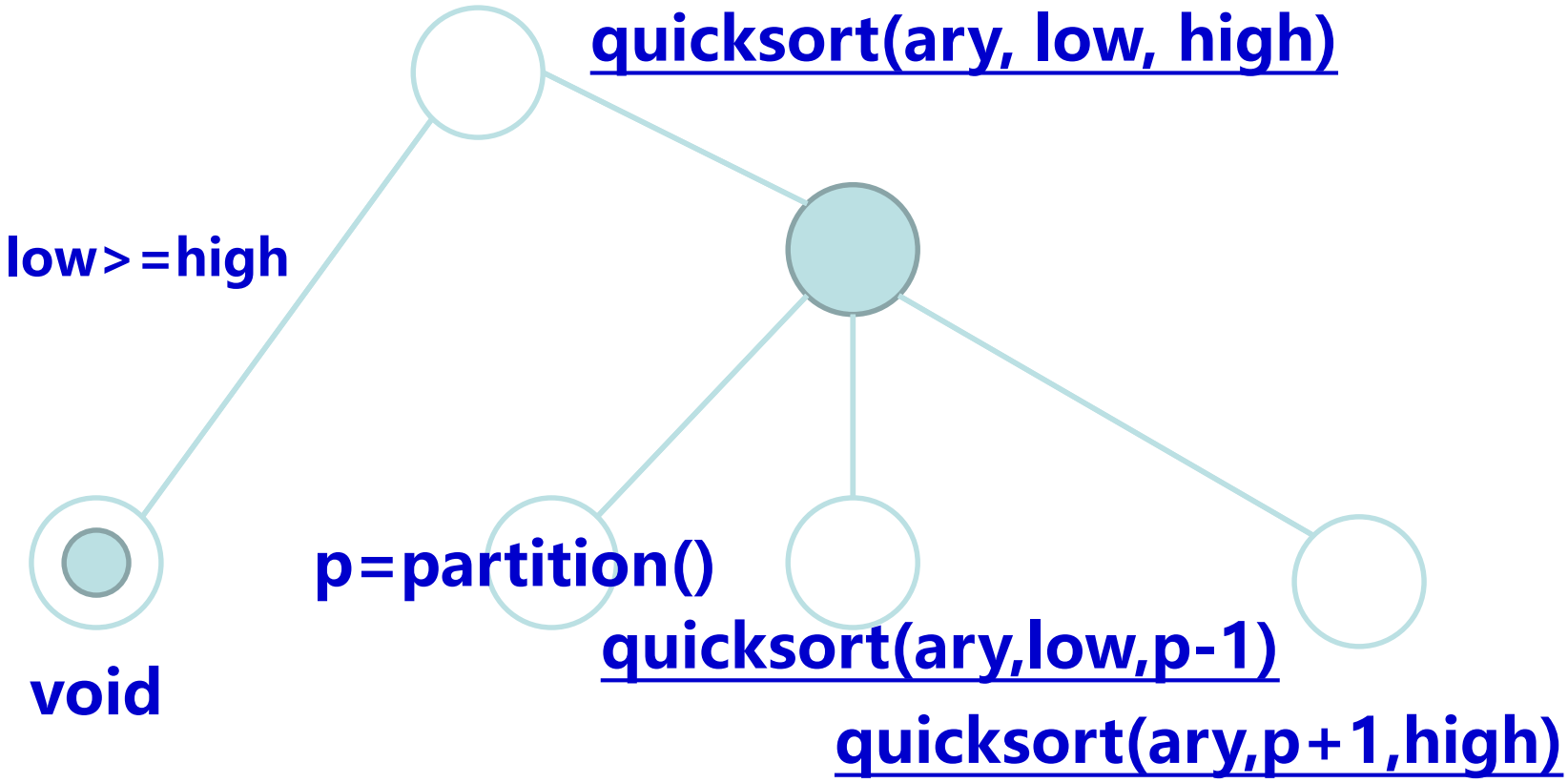
2. 重组织：重新组织数组，使其满足比pivot小的在左侧，大的在右侧，以此分区。
3. 做递归：分别对前后两部分执行上述步骤

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)

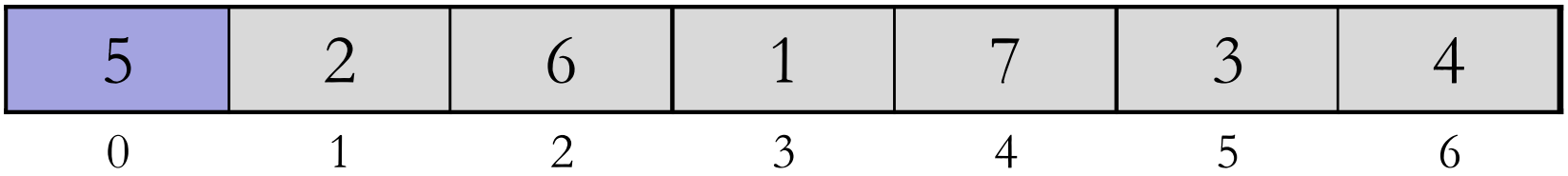


快速排序函数设计

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

快速排序partition函数设计 (1)

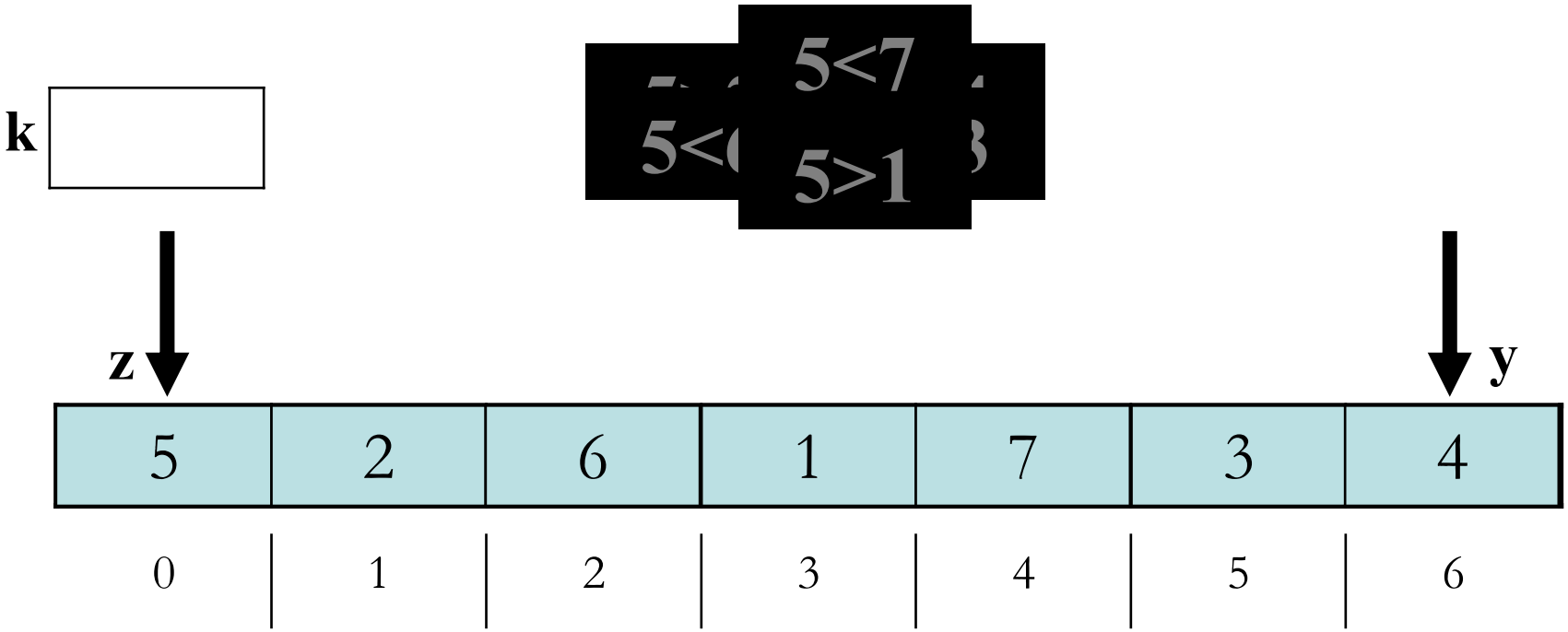
- **int partition(int ary[], int low, int high)**
- 核心要解决两个问题
  1. 选择哪个元素作为pivot？最左侧



  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?