中国人民大学 RENMIN UNIVERSITY OF CHINA

信息学院 SCHOOL OF INFORMATION

程序设计荣誉课程

# 11. 算法3——递归与分治

授课教师：游伟 副教授、孙亚辉 副教授

授课时间：周二14:00 – 15:30，周四10:00 – 11:30（教学三楼3304）

上机时间：周二18:00 – 21:00（理工配楼二层机房）

课程主页：https://www.youwei.site/course/programming

# 目录

# 引子：盗梦空间


**描述**

《盗梦空间》是一部精彩的影片，在这部电影里，Cobb等人可以进入梦境之中，梦境里的时间会比现实中的时间过得快得多，这里假设现实中的3分钟，在梦里就是1小时。

然而，Cobb他们利用强效镇静剂，可以从第一层梦境进入第二层梦境，甚至进入三层，四层梦境，每层梦境都会产生同样的时间加速效果。那么现在给你Cobb在各层梦境中经历的时间，你能算出现实世界过了多长时间吗?

比如，Cobb先在第一层梦境待了1个小时，又在第二层梦境里待了1天，之后，返回第一层梦境之后立刻返回了现实。

那么在现实世界里，其实过了396秒（6.6分钟）

**输入**

第一行输入一个整数M(3<=M<=100)
随后的M行每行的开头是一个字符串，该字符串如果是"IN" 则Cobb向更深层的梦境出发了，如果是字符串"OUT"则表示Cobb从深层的梦回到了上一层。如果是首字符串是"STAY"则表示Cobb在该层梦境中停留了一段时间，本行随后将是一个整数S表示在该层停留了S分钟（1<=S<=10000000)。数据保证在现实世界中，时间过了整数秒。

**输出**

输出现实世界过的时间（以秒为单位）。

**样例输入**

```
6
IN
STAY 60
IN
STAY 1440
OUT
OUT
```

**样例输出**

```
396
```

# 11.1 递归

- 什么是递归?
  - 递归函数：一个函数直接或间接调用自己本身，这种函数叫递归函数
  - 递归算法：把问题转化为规模缩小了的同类问题，然后递归调用函数来表示问题的解。具体到程序设计中，让函数不断引用自身，直到引用的对象已知（即到达递归边界）
- 迭代 v.s. 递归
  - 迭代：从简单的问题出发，一步步向前发展，最终求得问题解，是正向的，**从已知到未知**
  - 递归：从问题的最终目标出发，逐渐将复杂问题化为简单问题，最终求得问题解，是逆向的，**从未知到已知**
  - 二者有相同的求解能力，只是有些问题用递归思想更容易表达
  - 效率方面，递归的效率比迭代低

## 11.1.1 阶乘函数

- 非递归定义： $n! = 1\times 2\times 3\times \ldots \times (n-1)\times n$
- 递归定义：

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

- 其中：
  - n=0时，n!=1为**边界条件**（**基础步**）
  - n>0时，n!=n(n-1)!为**递推方程**（**归纳步**）

**说明**：**边界条件**与**递推方程**是递归函数的两个要素，递归函数只有具备了这两个要素，才能在有限次计算后得出结果。

# 11.1.1 阶乘函数

### 迭代解法

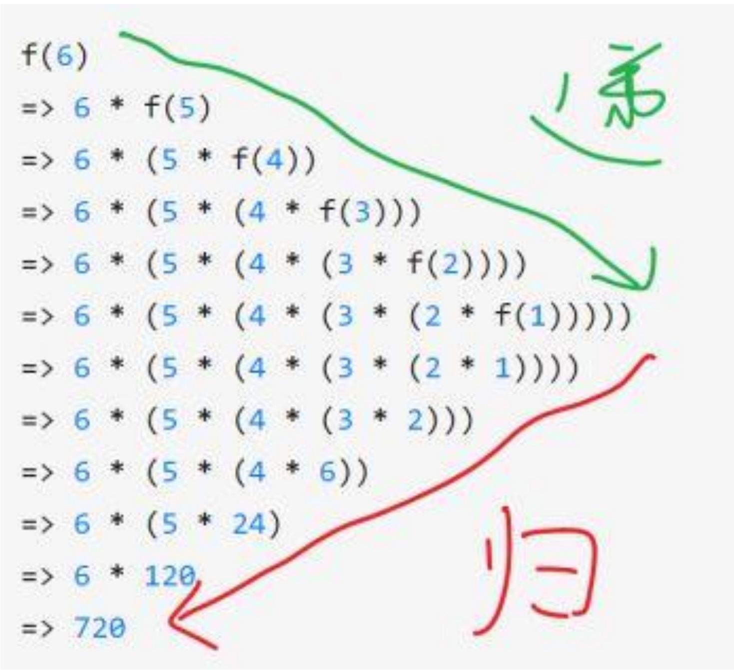
```c
#include <stdio.h>

int fact(int n) {
  int res = 1;
  int i;

  for (i = 1; i <= n; i++) res *= i;
  return res;
}

void main() {
  int n, res;
  scanf("%d", &n);
  res = fact(n);
  printf("%d", res);
}
```

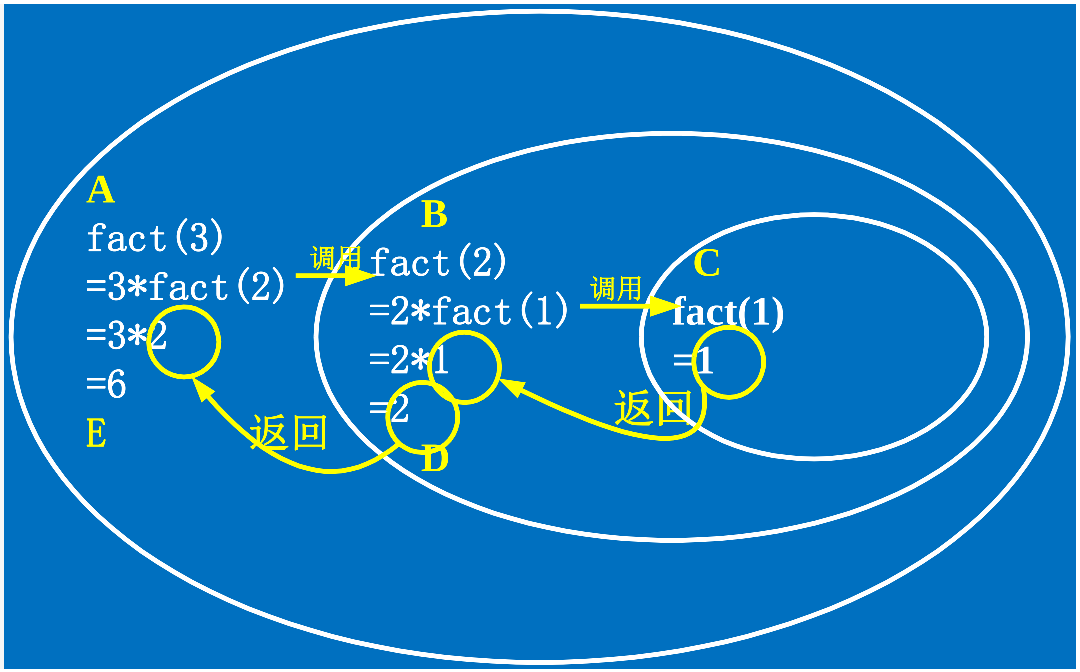
### 递归解法

```c
#include <stdio.h>

int fact(int n)
{
  if (n == 0 || n == 1) return 1;
  else return n * fact(n-1);
 }

void main()
{
  int n, res;
  scanf("%d", &n);
  res = fact(n);
  printf("%d", res);
}
```

# 11.1.1 阶乘函数

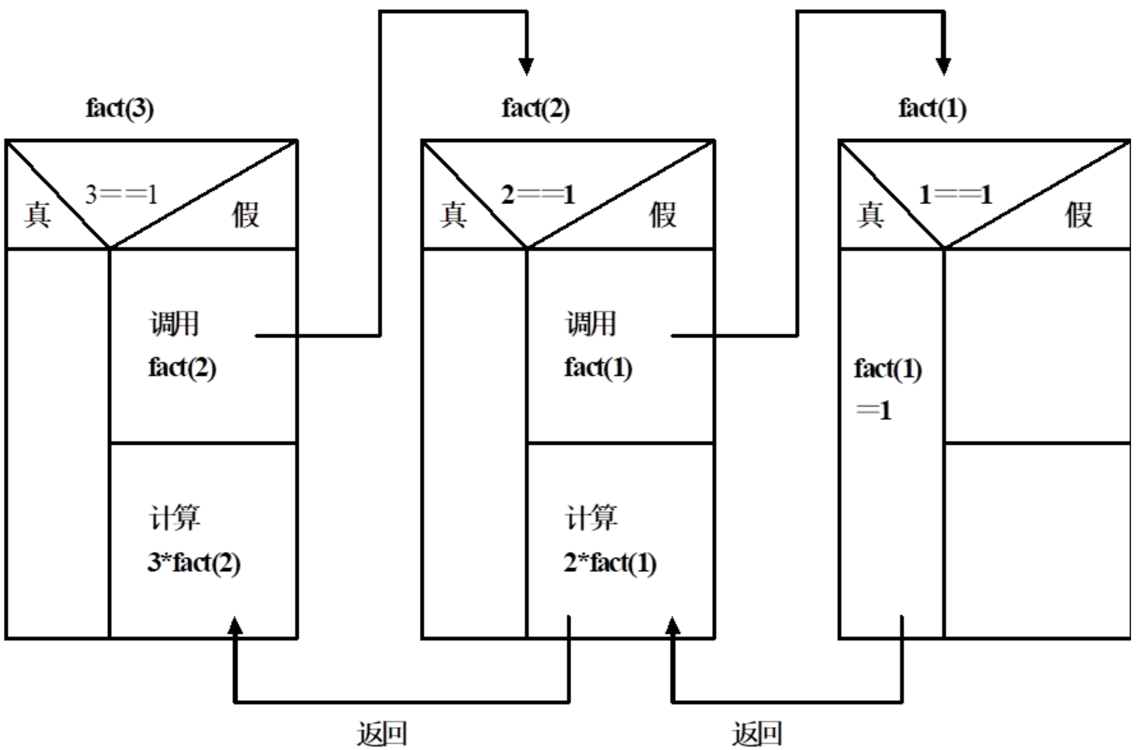
- 递归过程的形象示意图



# 11.1.1 阶乘函数

求解过程：欲求 fact( 3 )，先要求 fact( 2 )；要求 fact( 2 ) 先求 fact( 1 )。就像剥一颗圆白菜，从外向里，一层层剥下来，到了菜心，遇到 1 的阶乘，其值为1，到达了递归边界。然后再用 fact( n )=n*fact( n-1 ) 这个普遍公式，从里向外倒推回去得到 fact( n ) 的值。

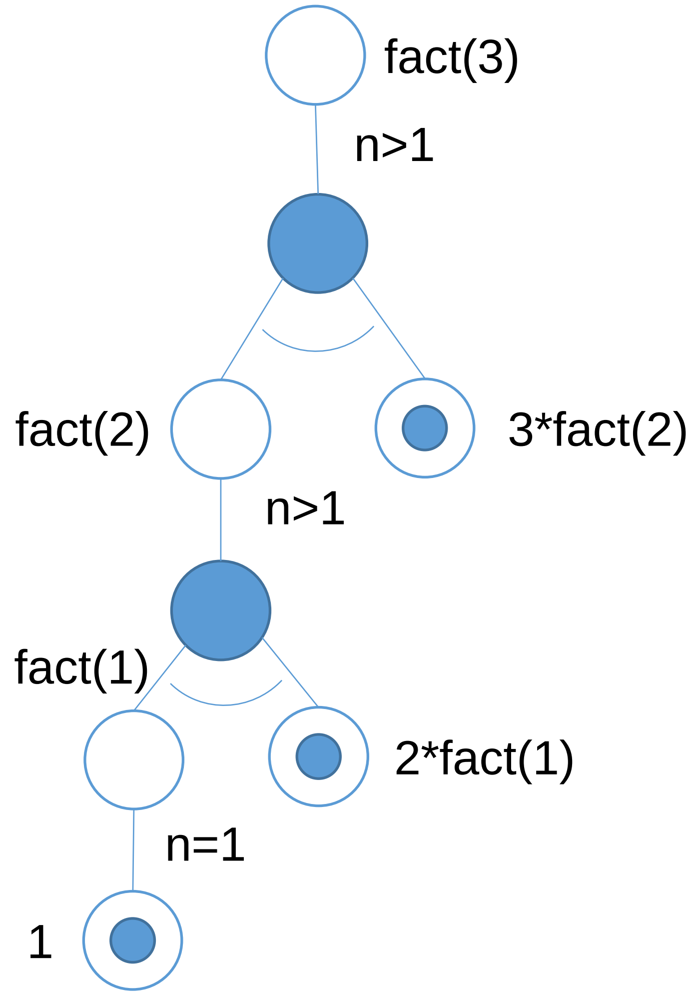


# 函数调用追踪

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

| | |
|---|---|
| | |
| | |
| | |
| | |
| 0 | main() |

函数调用栈(Call Stack)



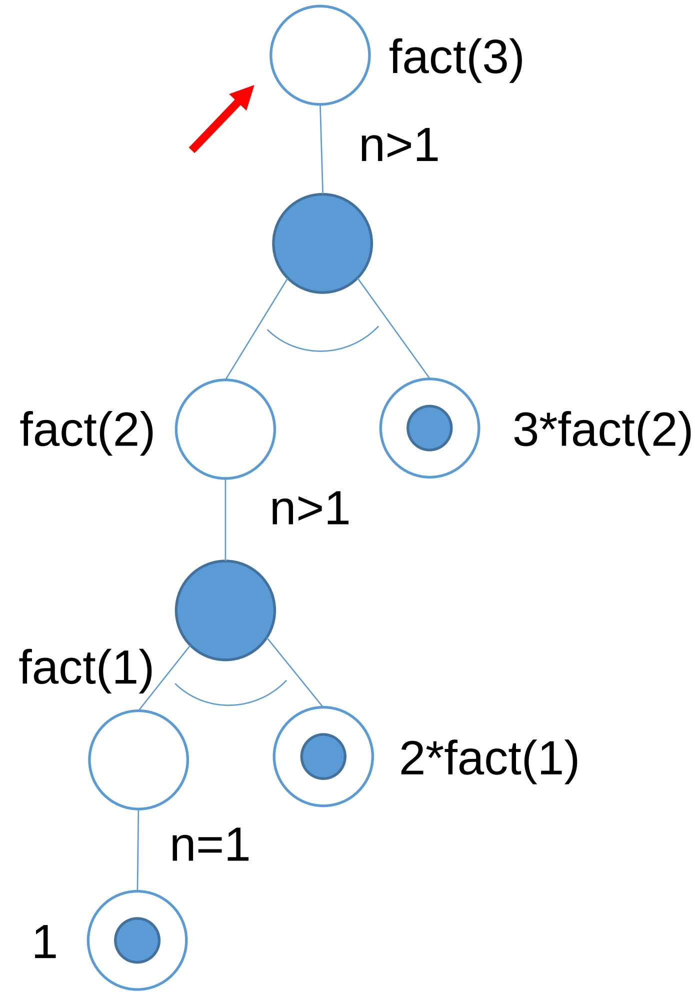

与或图

# 函数调用追踪

```
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)



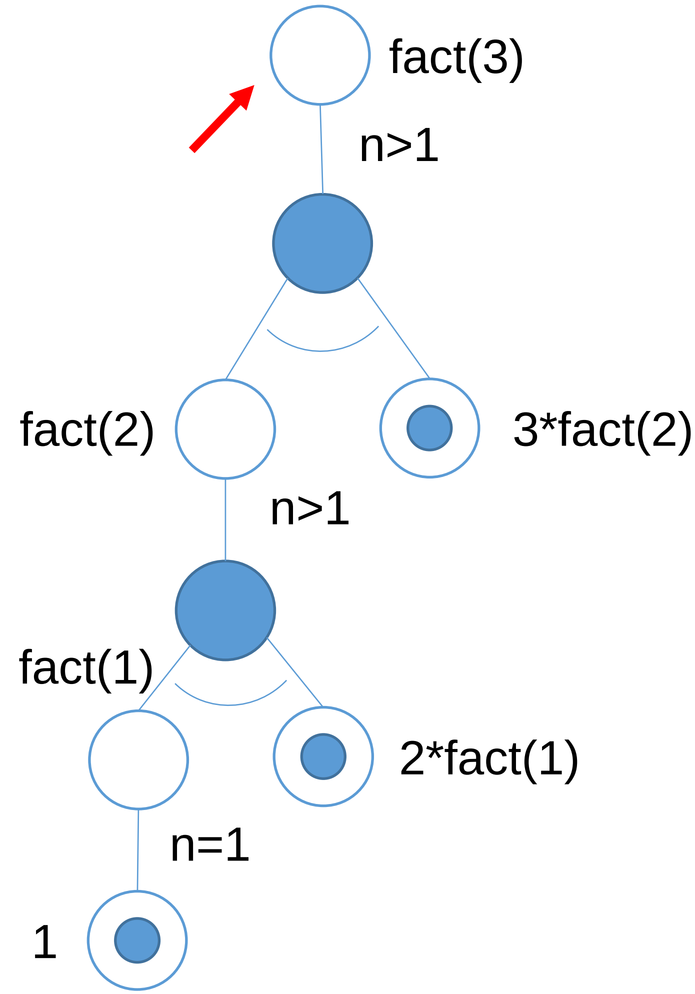

与或图

# 函数调用追踪

```
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)



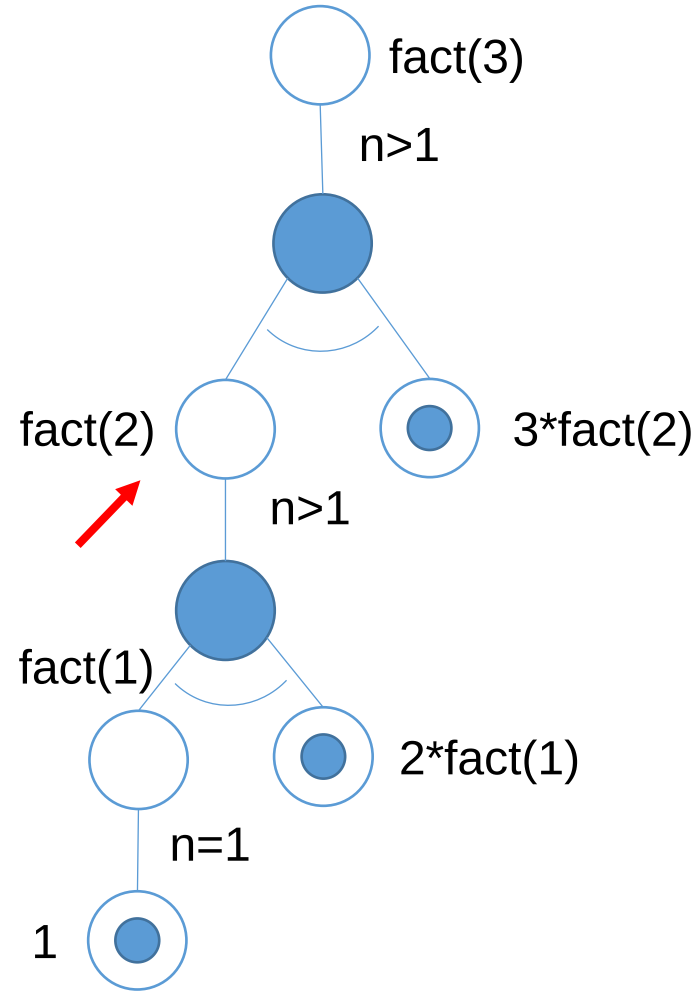

与或图

# 函数调用追踪

```
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)



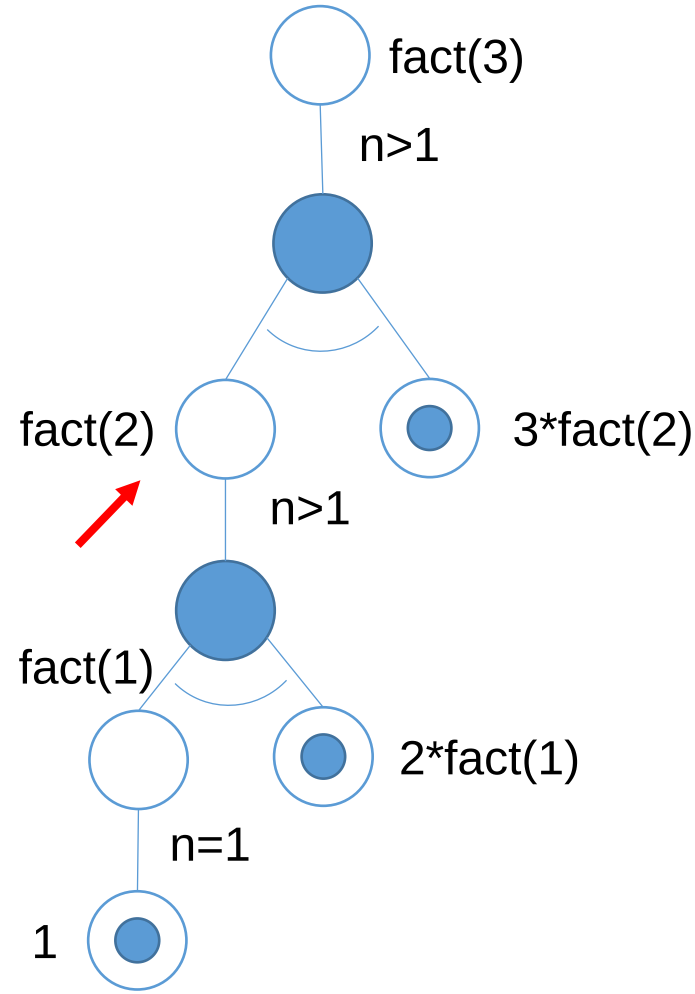

与或图

# 函数调用追踪

```
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)



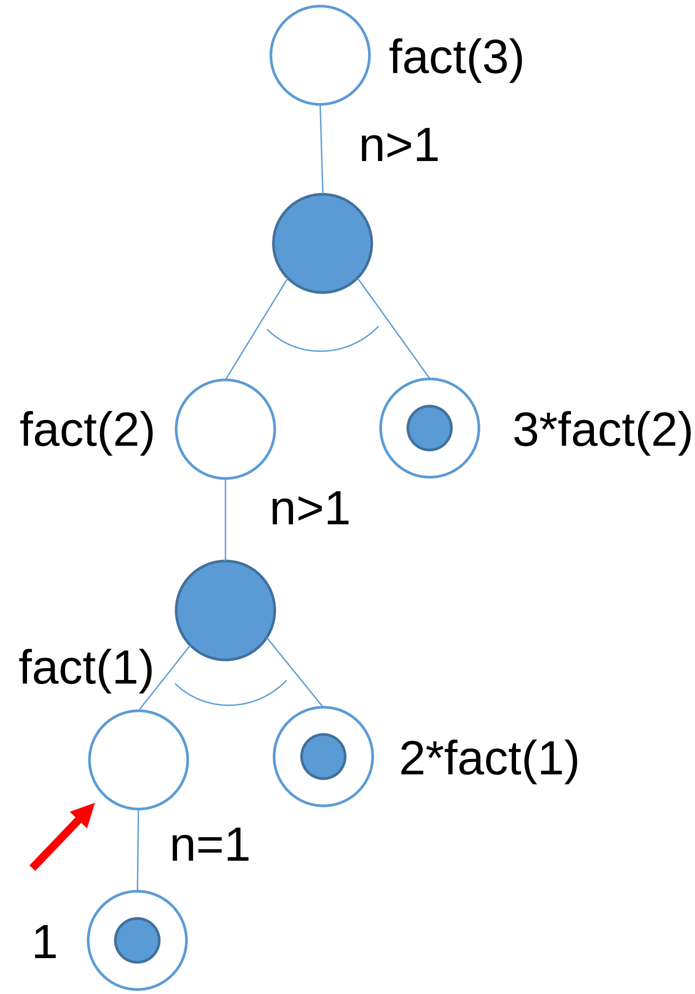

与或图

# 函数调用追踪

```
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)



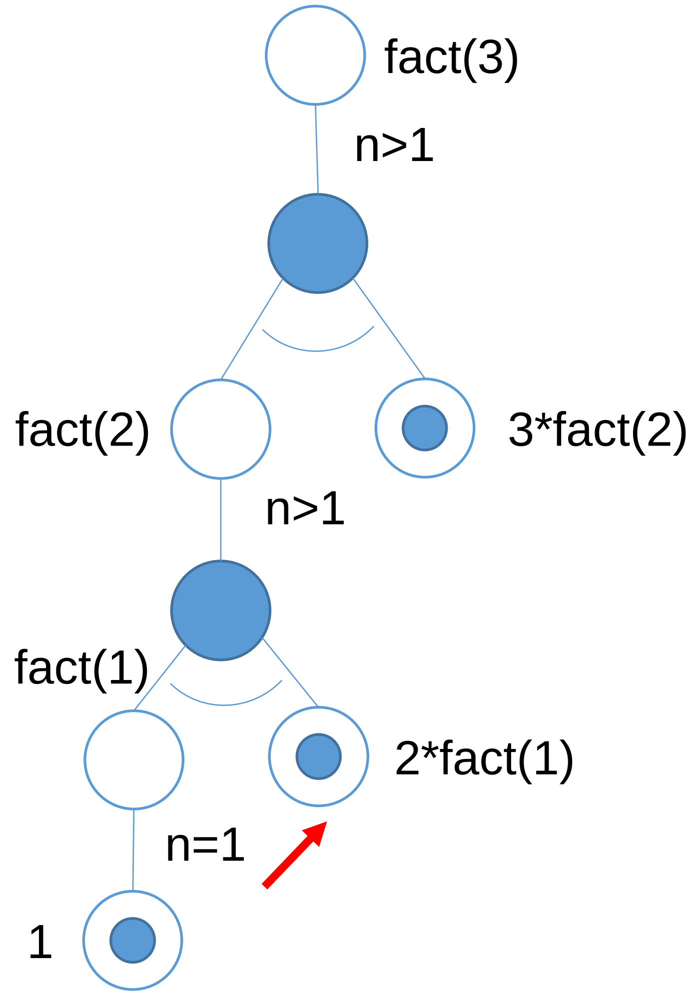

与或图

# 函数调用追踪

```
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)



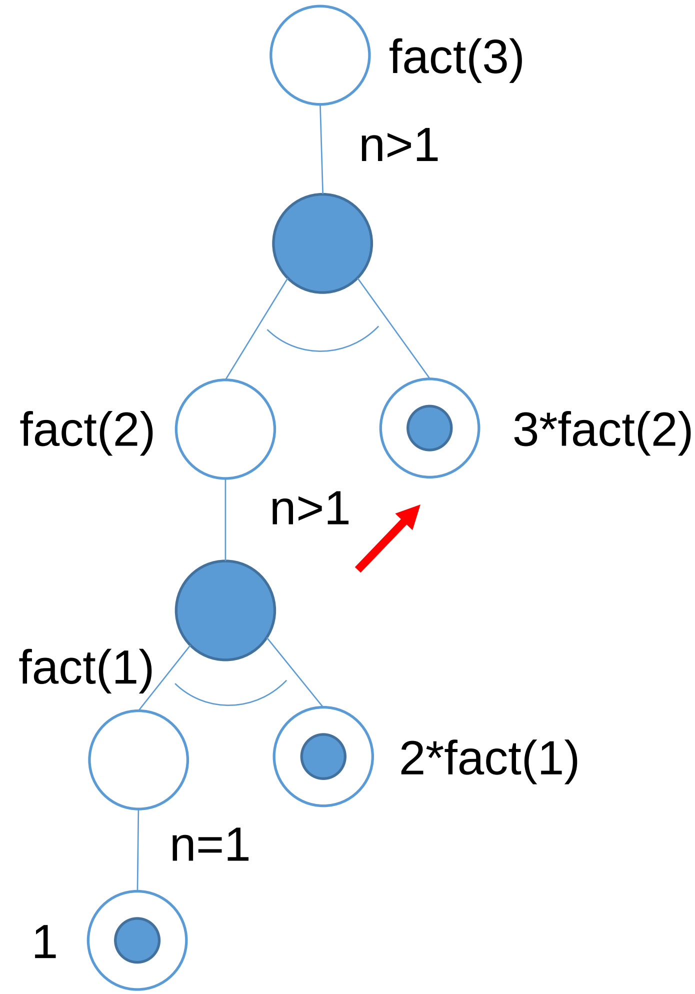

与或图

# 函数调用追踪

```
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)



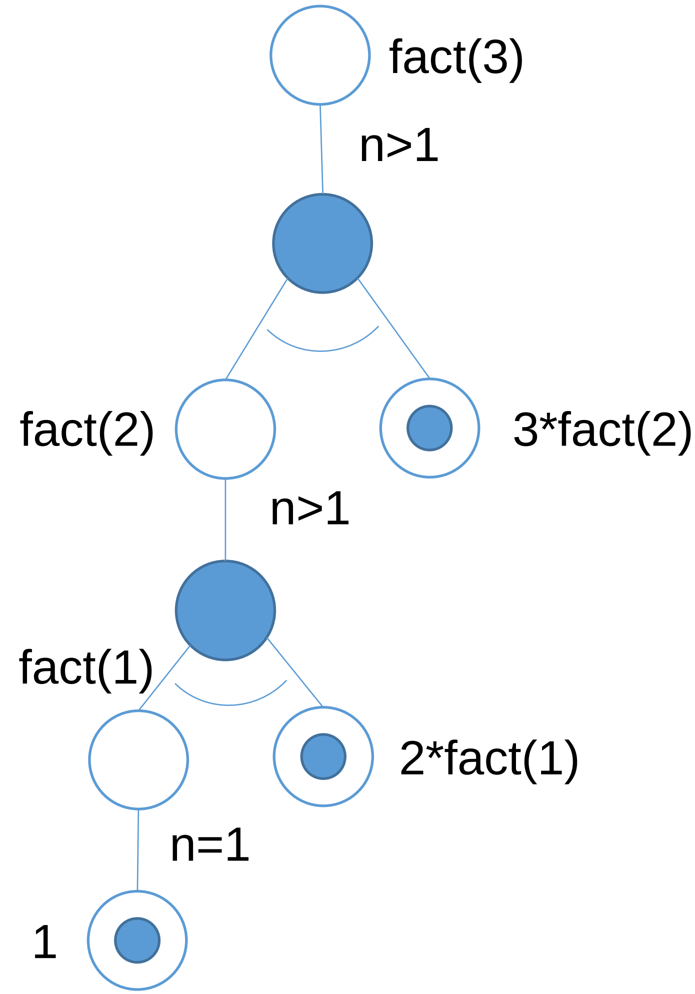

与或图

# 函数调用追踪

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

| | |
|---|---|
| | |
| | |
| | |
| | |
| 0 | main() |

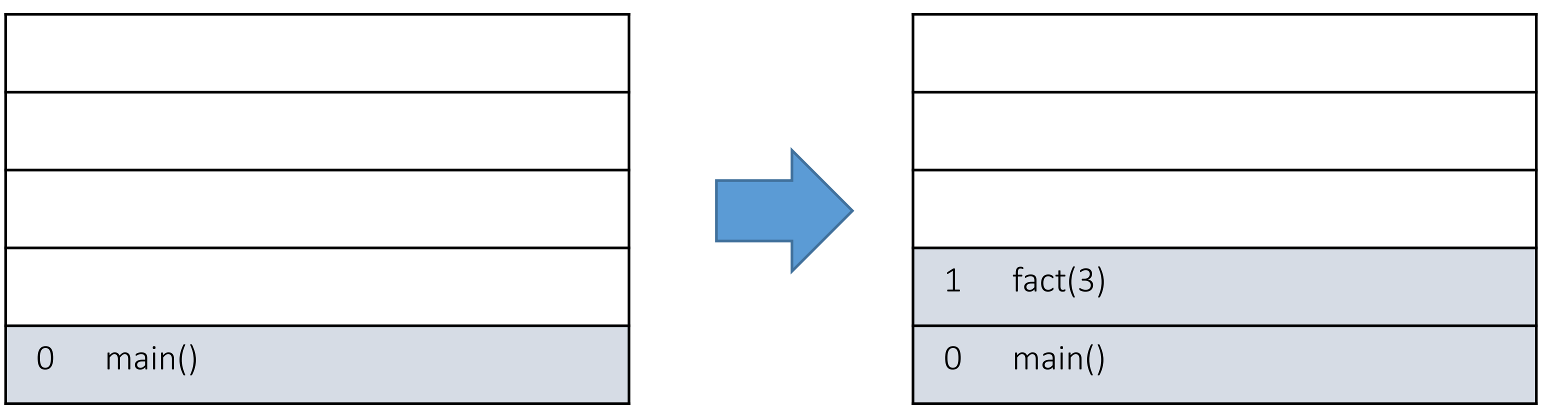
函数调用栈(Call Stack)




与或图

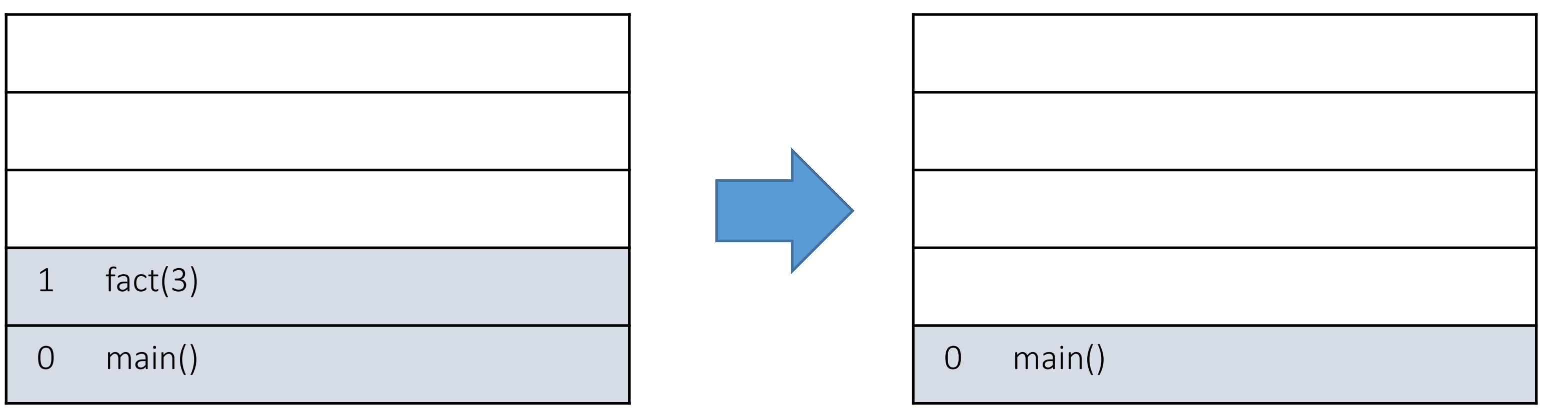
# 函数调用再思考

- 当在一个函数A运行时调用函数B时
  - 将所有的实在参数、返回地址等信息传递给被调用函数B保存；
  - 为被调用函数B的局部变量分配存储区；
  - 将控制转移到被调用函数的入口



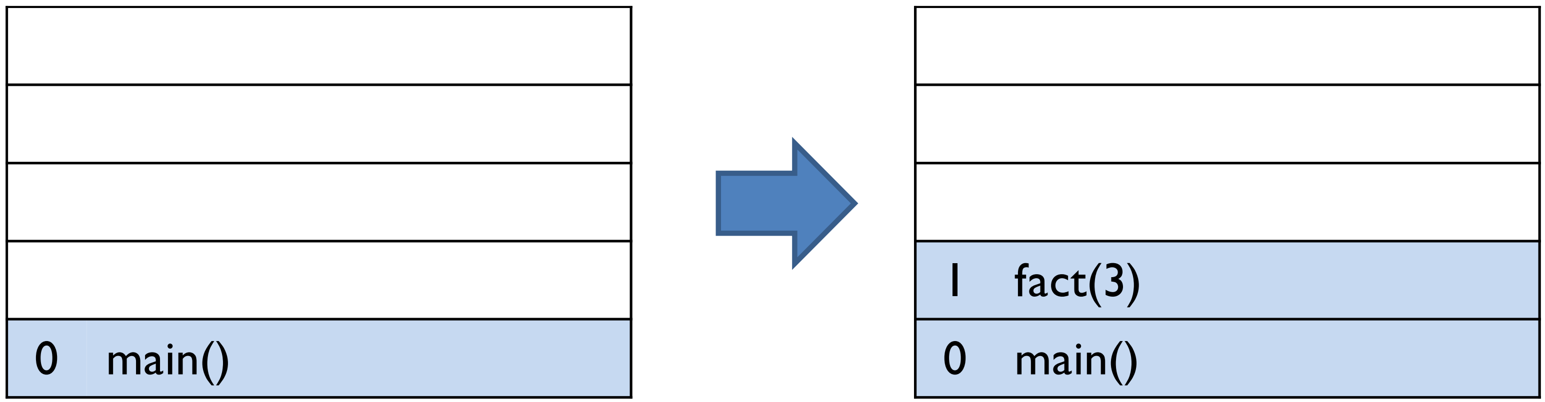
# 函数调用再思考

- 从被调用函数B结束返回函数A时
  - 保存被调用函数的计算结果；
  - 释放被调函数的数据区；
  - 依照被调函数保存的返回地址将控制转移到调用函数。

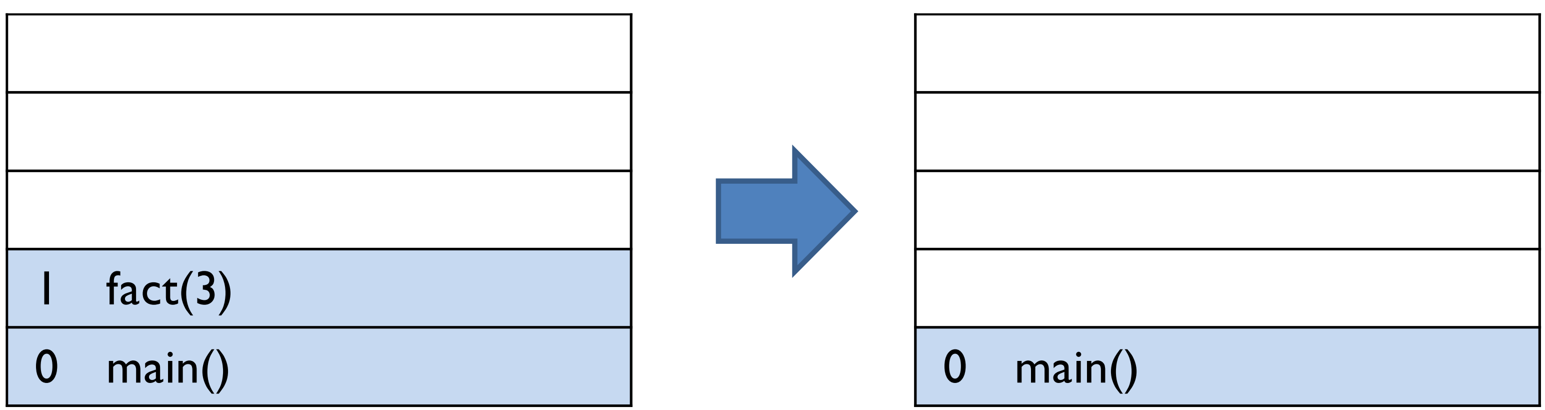


# 函数调用再思考

- 递：当在一个函数A运行时调用函数B时




- 归：从被调用函数B结束返回函数A时



# 函数调用再思考

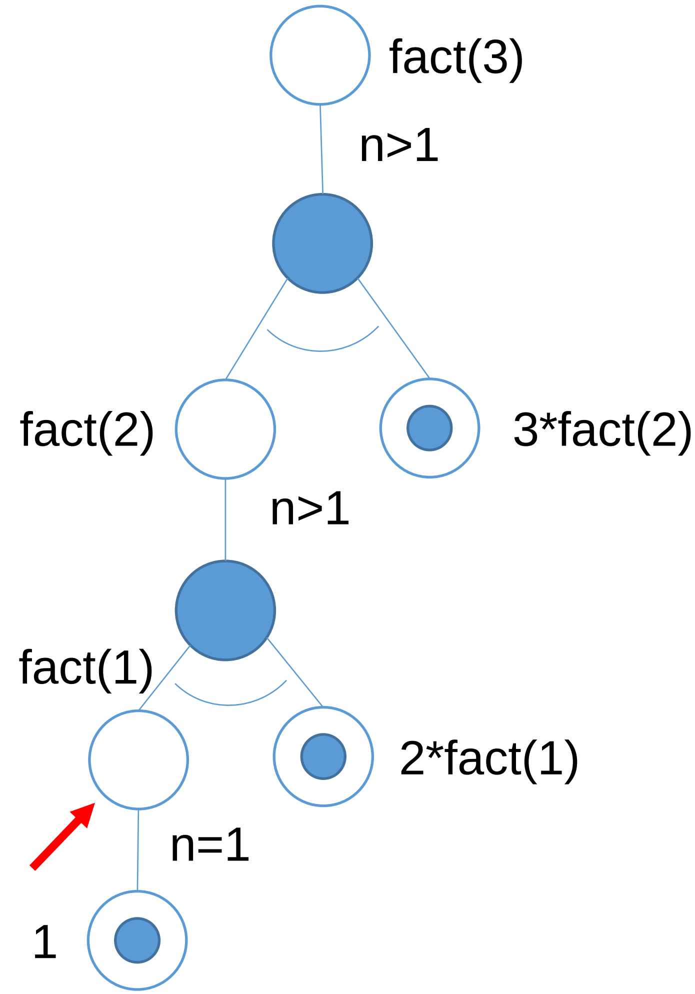
- 函数调用栈（Call Stack）
  - 将整个程序运行时所需的数据安排在一个栈中
  - 当前正运行的函数的数据区必在栈顶
  - 每当调用一个函数时，就为它在栈顶分配一个存储区
  - 每当从一个函数退出时，就释放它的存储区
- 什么是栈
  - 一种重要的数据结构
  - 后入先出Last In, First Out (LIFO)
  - 现实中例子：摞盘子

# 课堂思考

- 此时fact(2)和fact(3)的值是否计算出来了？
- 没有！ fact(1)的结果要首先返回，才能继续计算fact(2)和fact(3)

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

函数调用栈(Call Stack)




与或图

# 课堂思考

- 如果递归程序中没有对递归边界的处理？

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


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

| | |
|---|---|
| | .... |
| | .... |
| 8 | fact(-4) |
| 7 | fact(-3) |
| 6 | fact(-2) |
| 5 | fact(-1) |
| 4 | fact(0) |
| 3 | fact(1) |
| 2 | fact(2) |
| 1 | fact(3) |
| 0 | main() |

Stack

Overflow

## 11.1.2 斐波拉契数列

- 无穷数列1, 1, 2, 3, 5, 8, 13, 21, 34, 55,…， 被称为**Fibonacci数列**。
- 递归定义：

**边界条件**

$$F(n)=\begin{cases} 0 & n=0 \\ 1 & n=1 \\ F(n-1)+F(n-2) & n>1 \end{cases}$$

**递推方程**

# 11.1.2 斐波拉契数列

- **通项公式的推导**

设常数 $s,t$，使得 $F(n)-s\cdot F(n-1)=t[F(n-1)-s\cdot F(n-2)]$.

则 $s+t=1, st=-1$

n≥3 时，有

$$F(n)-s\cdot F(n-1)=t[F(n-1)-s\cdot F(n-2)]$$
$$F(n-1)-s\cdot F(n-2)=t[F(n-2)-s\cdot F(n-3)]$$
$$F(n-2)-s\cdot F(n-3)=t[F(n-3)-s\cdot F(n-4)]$$
$$\cdots\cdots$$
$$F(3)-s\cdot F(2)=t[F(2)-s\cdot F(1)]$$

将以上 n-2 个式子相乘，得：

$$F(n)-s\cdot F(n-1)=t^{n-2}[F(2)-s\cdot F(1)]$$

$$\because t=1-s, F(1)=F(2)=1$$

上式可化简为：$F(n)=t^{n-1}+s\cdot F(n-1)$

$$\begin{aligned}\therefore F(n)&=t^{n-1}+s\cdot F(n-1)\\&=t^{n-1}+st^{n-2}+s^2\cdot F(n-2)\\&=t^{n-1}+st^{n-2}+s^2t^{n-3}+s^3\cdot F(n-3)\\&=\cdots\cdots\\&=t^{n-1}+st^{n-2}+s^2t^{n-3}+\cdots\cdots+s^{n-2}t+s^{n-1}\cdot F(1)\\&=t^{n-1}+st^{n-2}+s^2t^{n-3}+\cdots\cdots+s^{n-2}t+s^{n-1}\\&=\frac{t^{n-1}\left[1-\left(\frac{s}{t}\right)^n\right]}{1-\frac{s}{t}}\\&=\frac{t^n-s^n}{t-s}\end{aligned}$$

$s+t=1, st=-1$ 的一解为 $s=\frac{1-\sqrt{5}}{2}, t=\frac{1+\sqrt{5}}{2}$

$$\therefore F(n)=\frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

# 11.1.2 斐波拉契数列

## 迭代解法

```c
#include <stdio.h>
#define N 1000

int compute;

int F(int n) {
  int ans[N] = {0, 1}, i;
  for (i = 2; i <= n; i++) {
    ans[i] = ans[i-1] + ans[i-2];
    compute++;
  }
  compute += 2;   //res[0]和res[1]的计算量
  return ans[n];
}

void main() {
  int res = F(4);
  printf("res: %d compute: %d\n",
          res, compute);
}
```

res: 3 compute: 5

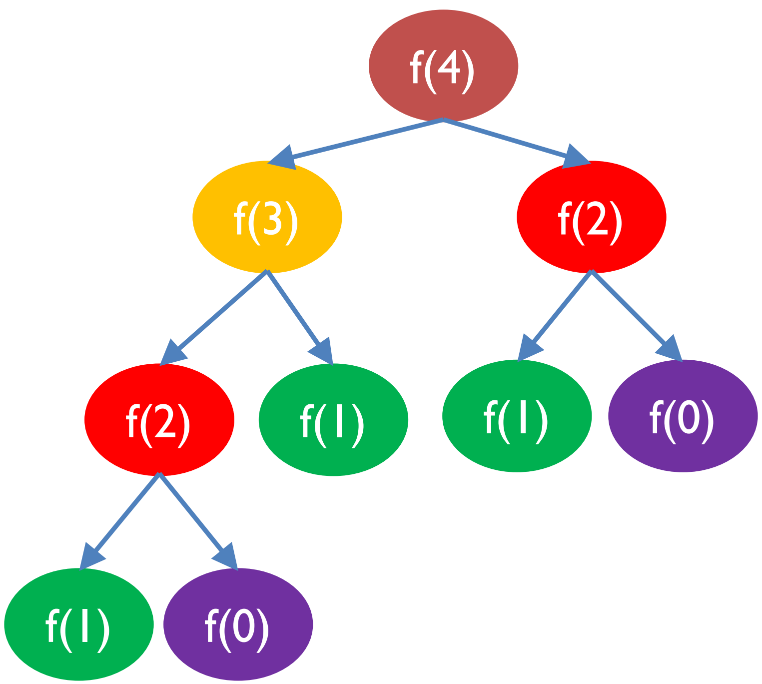
## 递归解法

```c
#include <stdio.h>

int compute;

int F(int n)
{
  compute++;
  if (n == 0) return 0;
  else if (n == 1) return 1;
  else return F(n-1) + F(n-2);
 }

void main()
{
  int res = F(4);
  printf("res: %d compute: %d\n",
          res, compute);
}
```

res: 3 compute: 9

# 11.1.2 斐波拉契数列

- 递归的优化：记忆法




Q：如何避免重复计算?
A：把已经计算过的值保存下来

递归解法（记忆优化）

```c
#include <stdio.h>
#define N 1000

int compute;
int ans[N];

int F(int n)
{
  compute++;
  if (n == 0) return 0;
  else if (n == 1) return 1;
  else {
    if (ans[n-1] == -1) ans[n-1] = F(n-1);
    if (ans[n-2] == -1) ans[n-2] = F(n-2);
    return ans[n-1] + ans[n-2];
  }
 }

void main()
{
  int i, res;
  for (i = 0; i < N; i++) ans[i] = -1;
  res = F(4);
  printf("res: %d compute: %d\n",
          res, compute);
}
```

res: 3 compute: 5

## 11.1.3 组合数$C_n^m$

- $C_n^m$表示从$n$个元素中取$m$个的组合数，递归定义如下：

$$C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$$

$$C_n^m = \begin{cases} 1, & m = n \\ n, & m = 1 \end{cases}$$

限制n ≥ m

限制m ≥ 1

- 非递归定义：

$$C_n^m = \frac{n!}{n!(n-m)!}$$

# 11.1.3 组合数$C_n^m$

### 迭代解法

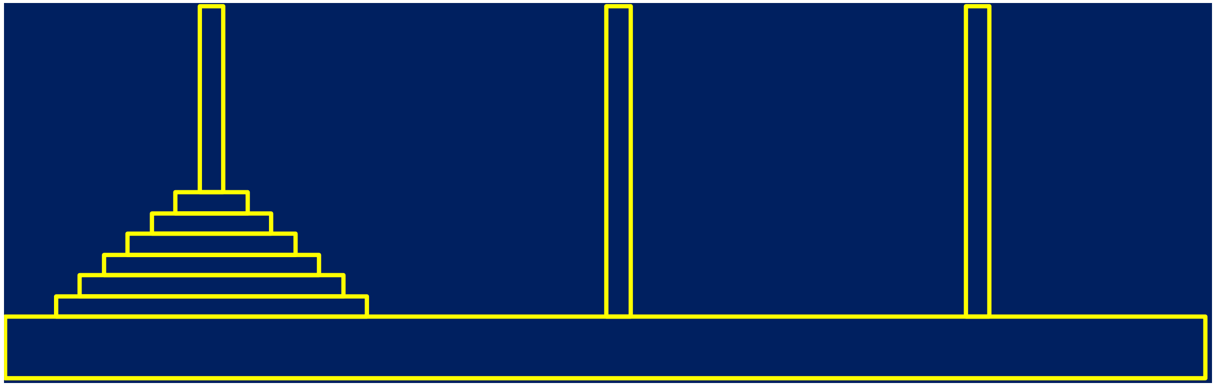
```c
#include <stdio.h>
#define MAX 100

long long ans[MAX][MAX];

void C(int n, int m) {
  int i, j;
  for (i = 1; i <= n; i++) {
    ans[i][1] = i;
    ans[i][i] = 1;
    for (j = 2; j < i; j++)
      ans[i][j] = ans[i-1][j] + ans[i-1][j-1];
  }
}

void main() {
  int n, m;
  scanf("%d %d", &n, &m);
  C(n, m);
  printf("ans = %lld\n", ans[n][m]);
}
```

### 递归解法（记忆优化）

```c
#include <stdio.h>
#define MAX 100

long long ans[MAX][MAX];

long long C(int n, int m) {
  if (n == m) return 1;
  if (m == 1) return n;
  if (ans[n-1][m] == 0)
    ans[n-1][m] = C(n-1, m);
  if (ans[n-1][m-1] == 0)
    ans[n-1][m-1] = C(n-1, m-1);
  return ans[n-1][m] + ans[n-1][m-1];
}

void main() {
  int n, m;
  scanf("%d %d", &n, &m);
  printf(“ans = %lld\n", C(n, m));
}
```

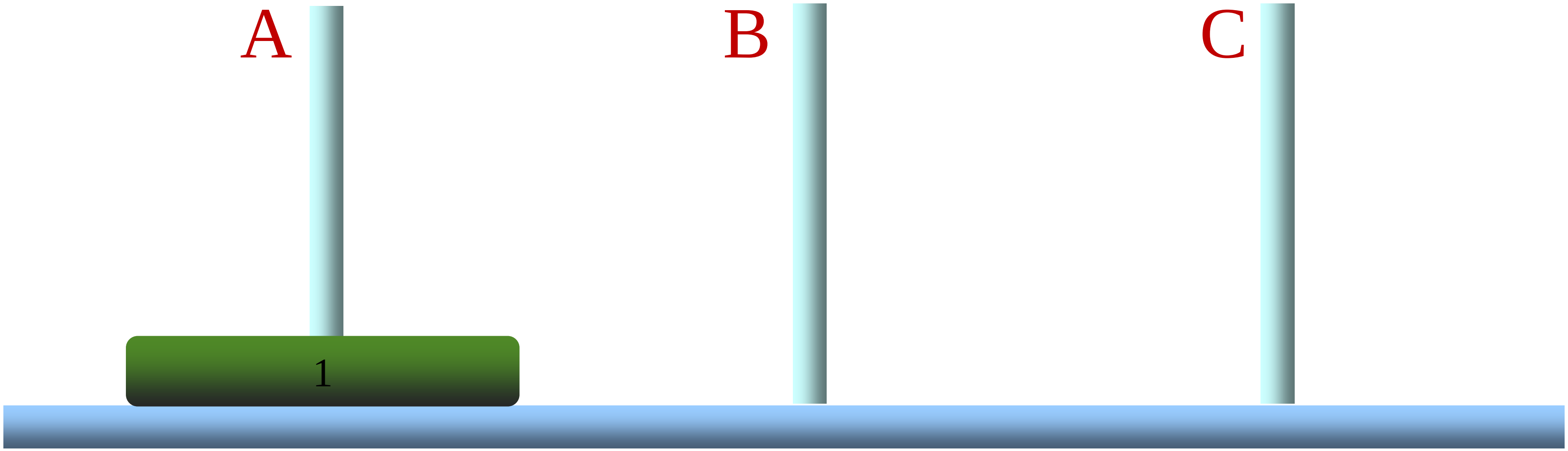
# 11.1.4 汉诺塔

相传在古代印度的 Bramah 庙中，有位僧人整天把三根柱子上的金盘倒来倒去，原来他是想把64个一个比一个小的金盘从一根柱子上移到另一根柱子上去。移动过程中恪守下述规则：每次只允许移动一只盘，且大盘不得落在小盘上面。有人会觉得这很简单，真的动手移盘就会发现，如以每秒移动一只盘子的话，按照上述规则将64只盘子从一个柱子移至另一个柱子上，所需时间约为5800亿年。



# 11.1.4 汉诺塔

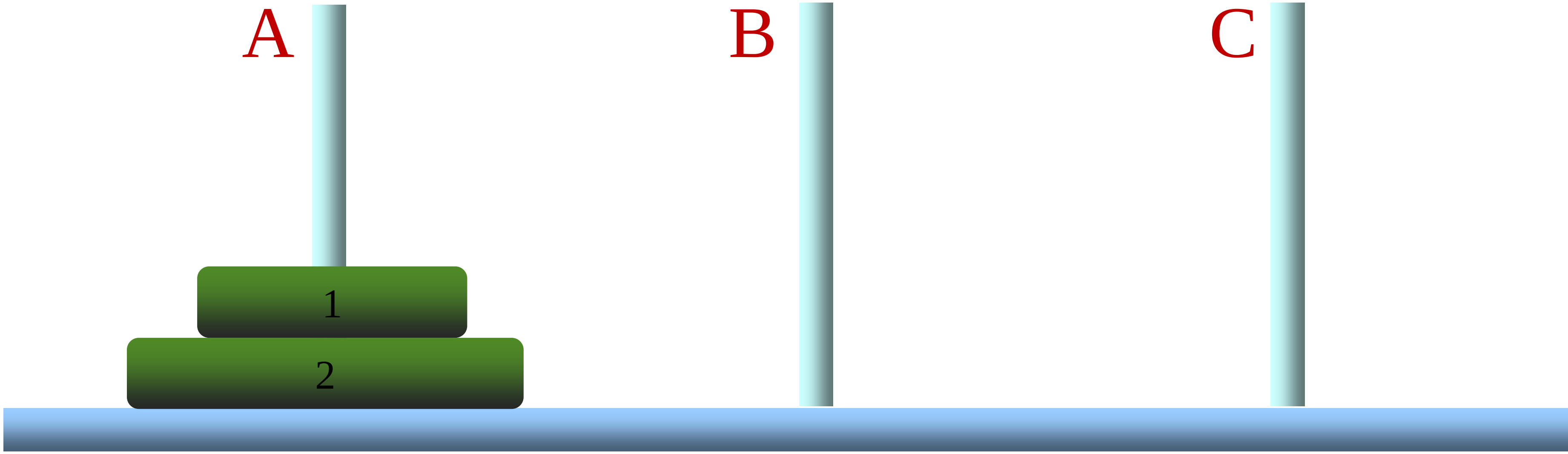
1. 在A柱上只有一只盘子，假定盘号为 1，这时只需将该盘从A搬至 C，一次完成，记为move 1 from A to C



# 11.1.4 汉诺塔

## 2. 在 A 柱上有二只盘子，1 为小盘，2 为大盘

- 第1步：将1号盘从A移至B，这是为了让 2号盘能移动，记为move 1 from A to B
- 第2步：将 2 号盘从A 移至 C，记为move 2 from A to C
- 第3步：再将 1 号盘从 B 移至 C，记为move 1 form B to C

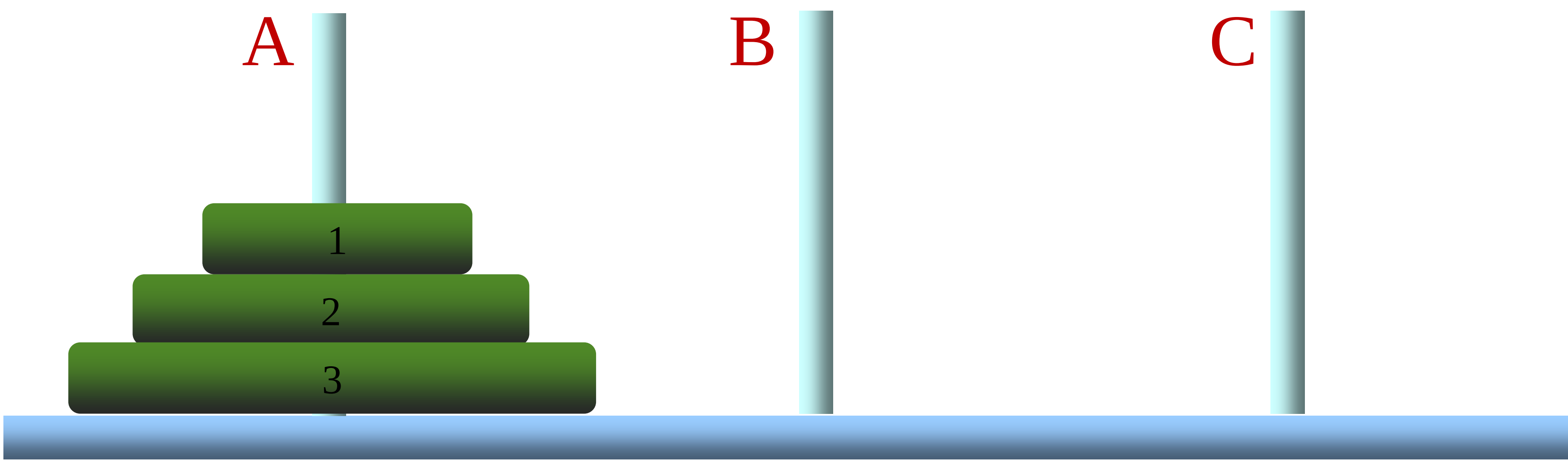


move (3, A, B, C) → move (2, A, C, B)
move (3, A, B, C) → move (1, A, B, C)
move (3, A, B, C) → move (2, B, A, C)

# 11.1.4 汉诺塔

## 3. 在A柱上有3只盘子，从小到大分别为1号，2号，3号

- 第1步：将1号盘和2号盘视为一个整体；先将二者作为整体从A移至B，给3号盘创造能够一次移至C的机会。这一步记为move(2, A, C, B)，意思是将上面的2只盘子作为整体从A借助C移至B
- 第2步：将3号盘从A移至C，一次到位，记为move 3 from A to C
- 第3步：处于B上的作为一个整体的2只盘子，再移至C。这一步记move(2, B, A, C)，意思是将2只盘子作为整体从B借助A移至C



# 11.1.4 汉诺塔

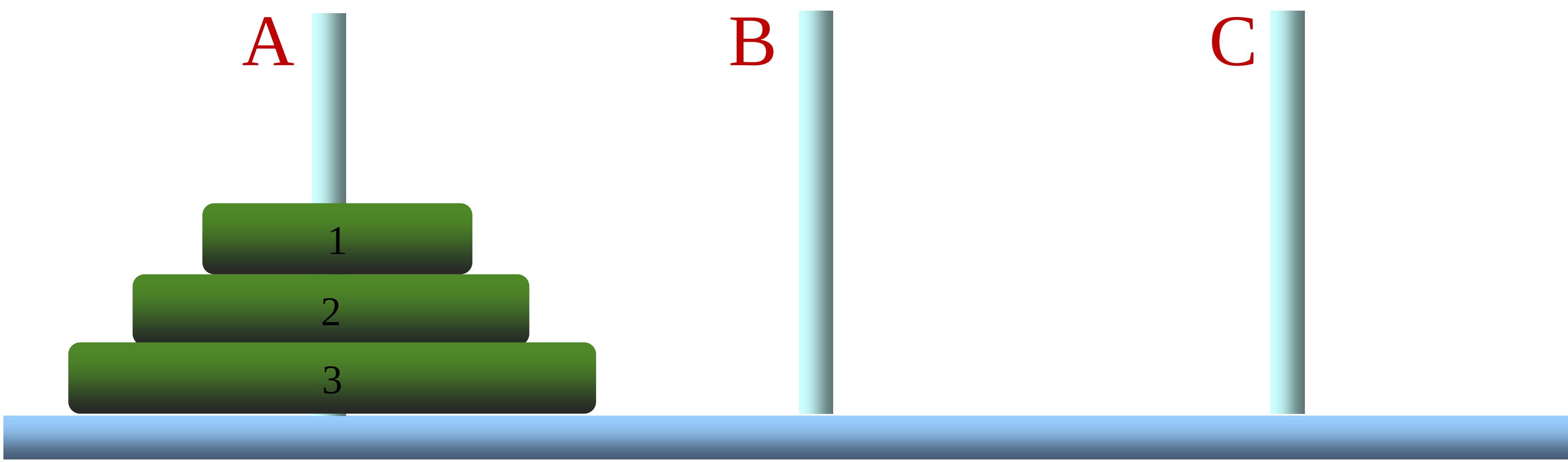
4. 从题目的约束条件看，大盘上可以随便摞小盘，相反则不允许。在将1号和2号盘当整体，从A移至B的过程中move(2, A, C, B) 实际上是分解为以下三步
   - 第1步：move 1 form A to C
   - 第2步：move 2 form A to B
   - 第3步：move 1 form C to B
5. 经过以上步骤，将 1 号和 2 号盘作为整体从 A 移至 B，为 3 号盘从 A 移至 C 创造了条件。同样，3号盘一旦到了 C，就要考虑如何实现将 1 号和 2 号盘当整体从 B 移至 C 的过程了。实际上 move(2, B, A, C) 也要分解为三步
   - 第1步：move 1 form B to A
   - 第2步：move 2 form B to C
   - 第3步：move 1 form A to C

## 11.1.4 汉诺塔

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 (3, A, B, C) → move (1, A, B, C)

move (3, A, B, C) → move (2, B, A, C) → move (1, B, C, A), move (1, B, A, C), move (1, A, B, C)

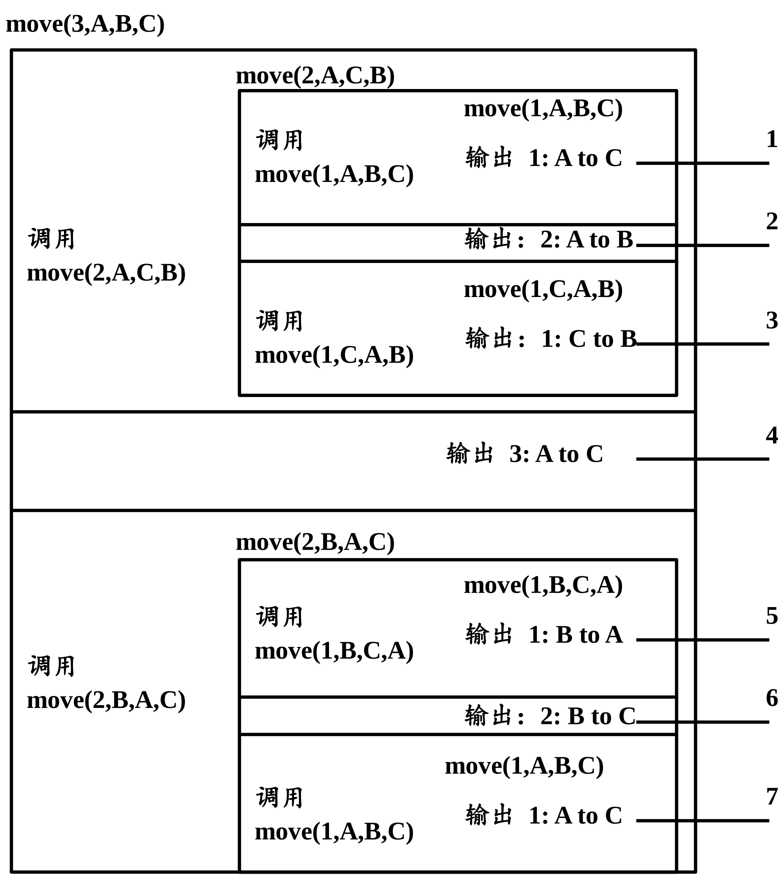


# 11.1.4 汉诺塔

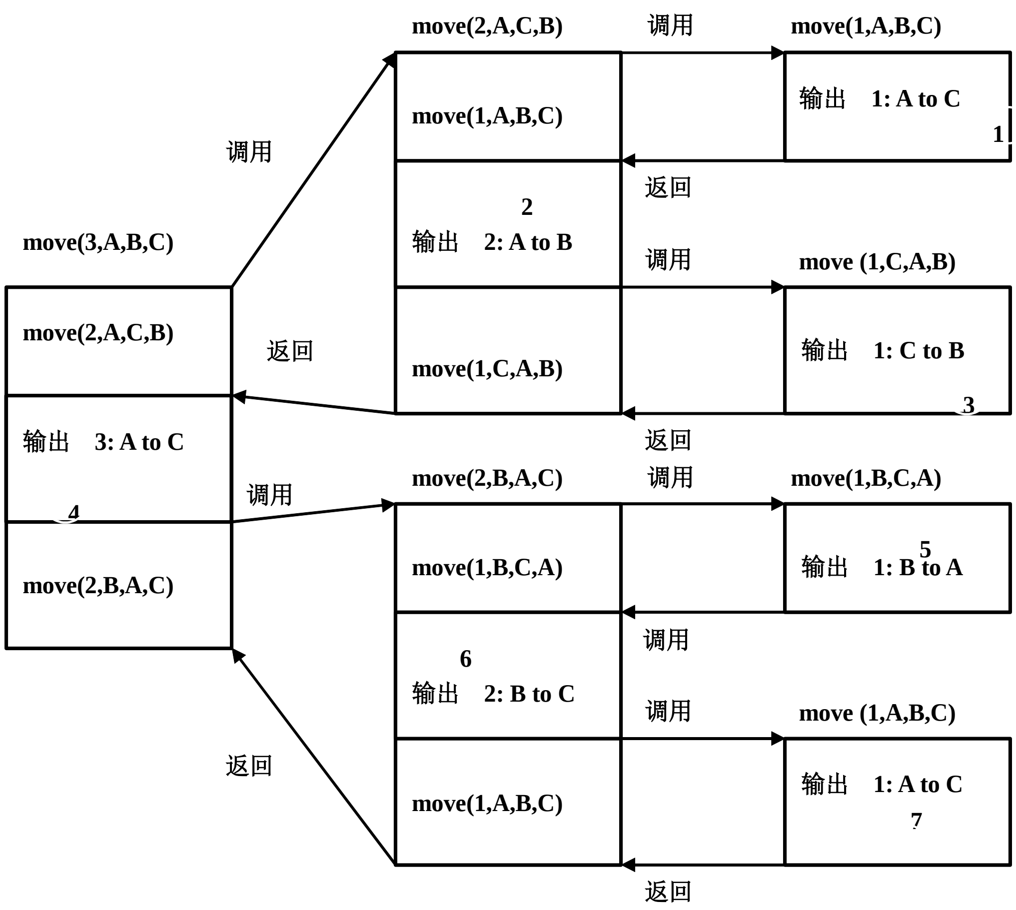
6. 定义搬移函数move(n, A, B, C)，物理意义是将 n 只盘子从 A 经 B 搬到 C。move(n, A, B, C) 分解为3步

- move(n-1, A, C, B)：理解为将上面的n-1只盘子作为一个整体从A经C移至B
- move n from A to C：理解为将n号盘从A移至C，是直接可解结点
- move(n-1, B, A, C)：理解为将上面的n-1只盘子作为一个整体从B经A移至C

# 11.1.4 汉诺塔



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

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

# 11.1.4 汉诺塔

```c
#include <stdio.h>

void move(int,char,char,char); //move函数声明
int steps;

int main()
{
   int n;

   scanf("%d", &n);
   move(n, 'a', 'b', 'c');
   printf("steps: %d\n", steps);

   return 0;
}
void move(int m, char p, char q, char r) {
   if (m==1) {  //如果m为1,则为直接可解结点
      //直接可解结点,输出移盘信息
     printf("[%d] move %d# from %d to %d\n",
             steps, m, p, r);
     steps++;    //步数加1
   } else {      //如果不为1,则要调用move(m-1)
     move(m-1,p,r,q);    //递归调用move(m-1)
     //直接可解结点,输出移盘信息
     printf("[%d] move %d# from %d to %d\n",
             steps, m, p, r);
     steps++;            // 步数加1
     move(m-1,q,p,r);    // 递归调用move(m-1)
   }
}
```

# 盗梦空间的解

## 迭代解法

```c
#include <stdio.h>

int main() {
  int m;
  int scale[6] = {0,20,400,8000,160000,3200000};
  char operation[10];
  int duration = 0, level = 0, stay;

  scanf("%d", &m);

  while (m > 0) {
    scanf("%s", operation);
    m--;
    if (strcmp(operation, "IN") == 0) {
      level++;
    } else if (strcmp(operation, "STAY") == 0) {
      scanf("%d", &stay);
      duration += (stay*60)/scale[level];
    } else if (strcmp(operation, "OUT") == 0) {
      level--;
    }
  }

  printf("%d", duration);
}
```

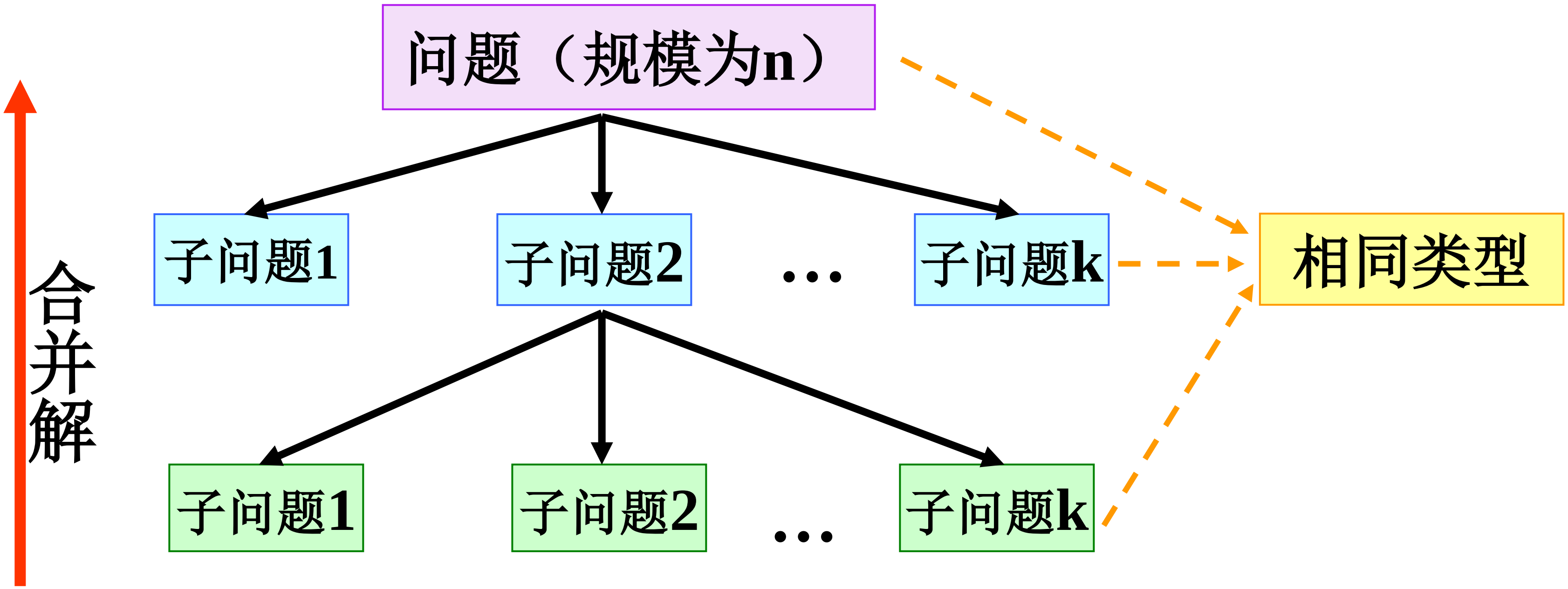
## 递归解法

```c
#include <stdio.h>
int m;

int dream(int level) {
  int duration = 0, stay;
  char operation[10];
  if (level > 6) return 0;
  while (m > 0) {
    scanf("%s", operation);
    m--;
    if (strcmp(operation, "IN") == 0) {
      duration += dream(level+1) / 20;
    } else if (strcmp(operation, "STAY") == 0) {
      scanf("%d", &stay);
      duration += stay*60;
    } else if (strcmp(operation, "OUT") == 0) {
      return duration;
    }
  }
  return duration;
}

int main() {
  int duration;
  scanf("%d", &m);
  duration = dream(0);
  printf("%d", duration);
}
```

# 11.2 分治

- 基本思想：将一个规模为n的问题分解为k个规模较小的子问题，这些子问题互相独立且与原问题相同
- 对这k个子问题分别求解。如果子问题的规模仍然不够小，则再划分为k个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止
- 将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解

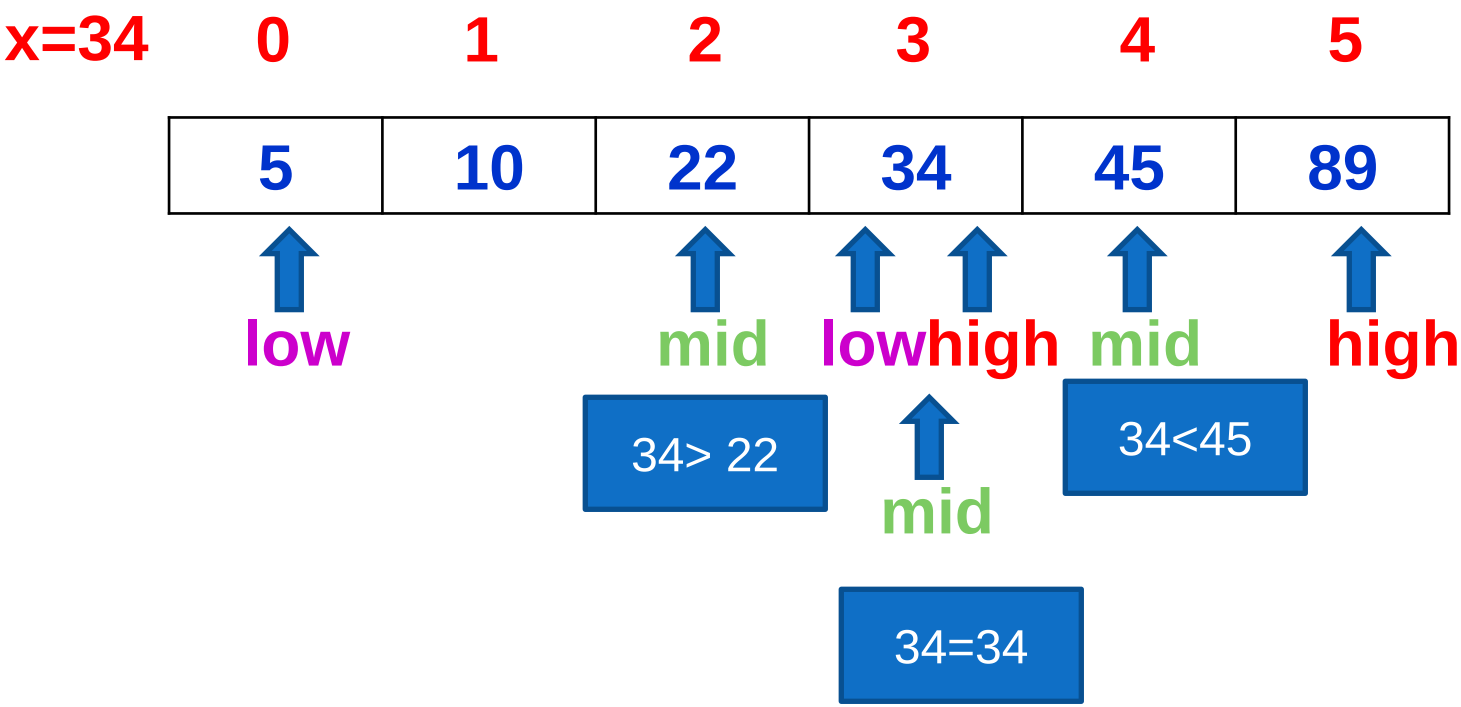
# 11.2 分治




- 由于分治法要求分解成同类子问题，并允许不断分解，使问题规模逐步减小，最终可用已知的方法求解足够小的问题
- 因此，分治法求解显然是一个递归算法

# 11.2.1 折半查找

- 基本思想
  - 适用范围：有序表
  - 方法：
    - 设置变量low和high分别指向查找数据段的起止位置，用mid指向中间位置，
    - 将被查找数与mid位置指向的数进行比较
    - 若被查找数大于mid位置指向的数，则将low与mid之间的数折掉， low定位于mid+1位置；
    - 若被查找数小于mid位置指向的数，则将mid 与high之间的数折掉， high定位于mid-1位置；
    - 继续求mid位置，并比较被查找数与mid位置指向的数，直到找到或确定不存在为止

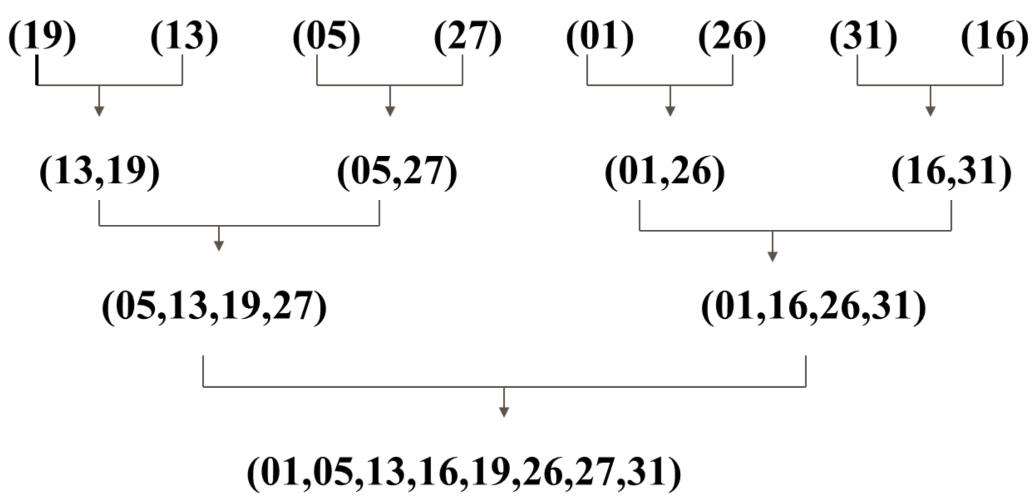



```
int BinSearch(long a[], long x, int n)
{
  int low, high, mid;
  low = 0;
  high = n - 1;
  while (low <= high)
  {
    mid = (high + low) / 2;
    if (x > a[mid]) low = mid + 1;
    else if (x < a[mid]) high = mid - 1;
    else return (mid);
  }
  return(-1);
}
```

```
int BinSearch(long a[], long x, int low, int high)
{
  int mid;

  if (low > high) return -1;

  mid = (high + low) / 2;

  if (x > a[mid])
    return BinSearch(a, x, mid+1, high);
  else if (x < a[mid])
    return BinSearch(a, x, low, mid-1);
  else return mid;
}
```

# 11.2.2 归并排序

- 基本思想：将两个或两个以上有序表合并成一个新的有序表
  - 假设初始序列含有n个记录，首先将这n个记录看成n个有序的子序列，每个子序列的长度为1
  - 两两归并，得到⌈n/2⌉个长度为2（n为奇数时，最后一个序列长度为1）的有序子序列
  - 在此基础上再进行两两归并，如此重复，直至得到一个长度为n的有序序列为止。这种方法被称作2-路归并排序



## 11.2.2 归并排序

a: 19 13 05 27 01 26 31 16 len=1

b: 13 19 05 27 01 26 16 31 len=2

a: 05 13 19 27 01 16 26 31 len=4

b: 01 05 13 16 19 26 27 31 len=8

# 11.2.2 归并排序

```c
#include <stdio.h>
#define N 1000

void merge(int[], int, int, int);
void sort(int[], int, int);

int a[N], b[N];

int main()
{
    int n, i;

    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);

    sort(a, 0, n - 1);

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

```c
void sort(int a[], int low, int high)
{
    int mid;
    if (low < high)
    {
        mid = (low + high) / 2;
        sort(a, low, mid);
        sort(a, mid + 1, high);
        merge(a, low, mid, high);
    }
}

void merge(int a[], int low, int mid, int high) {
  int i = low, j = mid + 1, k = 0;

  while ((i <= mid) && (j <= high))
  {
    if (a[i] <= a[j]) b[k++] = a[i++];
    else b[k++] = a[j++];
  }

  while (i <= mid) b[k++] = a[i++];
  while (j <= high) b[k++] = a[j++];

  for (i = low, k = 0; i <= high; i++) a[i] = b[k++];
}
```

# 11.2.2 归并排序

- 算法分析
  - 假设：前后相邻的有序段长度为h，进行两两归并后，得到长度为2h的有序段，那么一趟归并排序将调用$\lceil n/2h \rceil$次算法merge将a[0..n-1]存放在b[0..n-1]中，因此时间复杂度为O(n) 。
  - 整个归并排序需进行m（$m=\log_2 n$）趟2-路归并，所以归并排序的时间复杂度为O（$n\log_2 n$）
  - 在实现归并排序时，需要和待排记录等数量的辅助空间，空间复杂度为O(n)

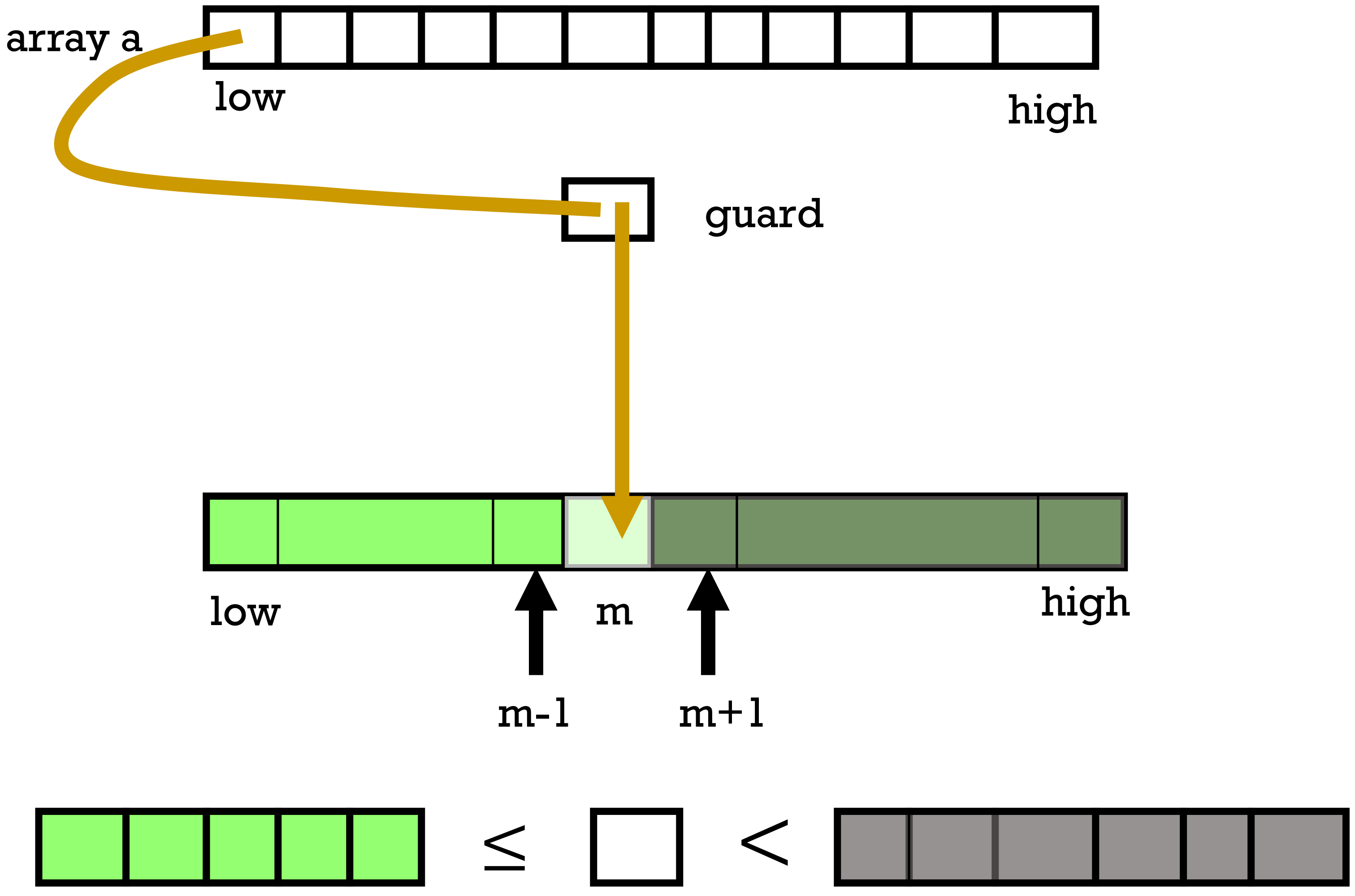
# 11.2.2 归并排序

- 多路归并排序与外部排序
  - 类似2-路归并排序，可设计多路归并排序法，归并的思想主要用于外部排序
  - 外部排序可分为两步：①待排序记录分批读入内存，用某种方法在内存排序，组成有序子文件，再按某种策略存入外存。②子文件多路归并，成为较长有序子文件，再存入外存，如此反复直到整个待排序文件有序
  - 外部排序可使用外存、磁带、磁盘，最初形成有序子文件长取决于内存所能提供排序区大小和最初排序策略，归并路数取决于所能提供排序的外部设备数

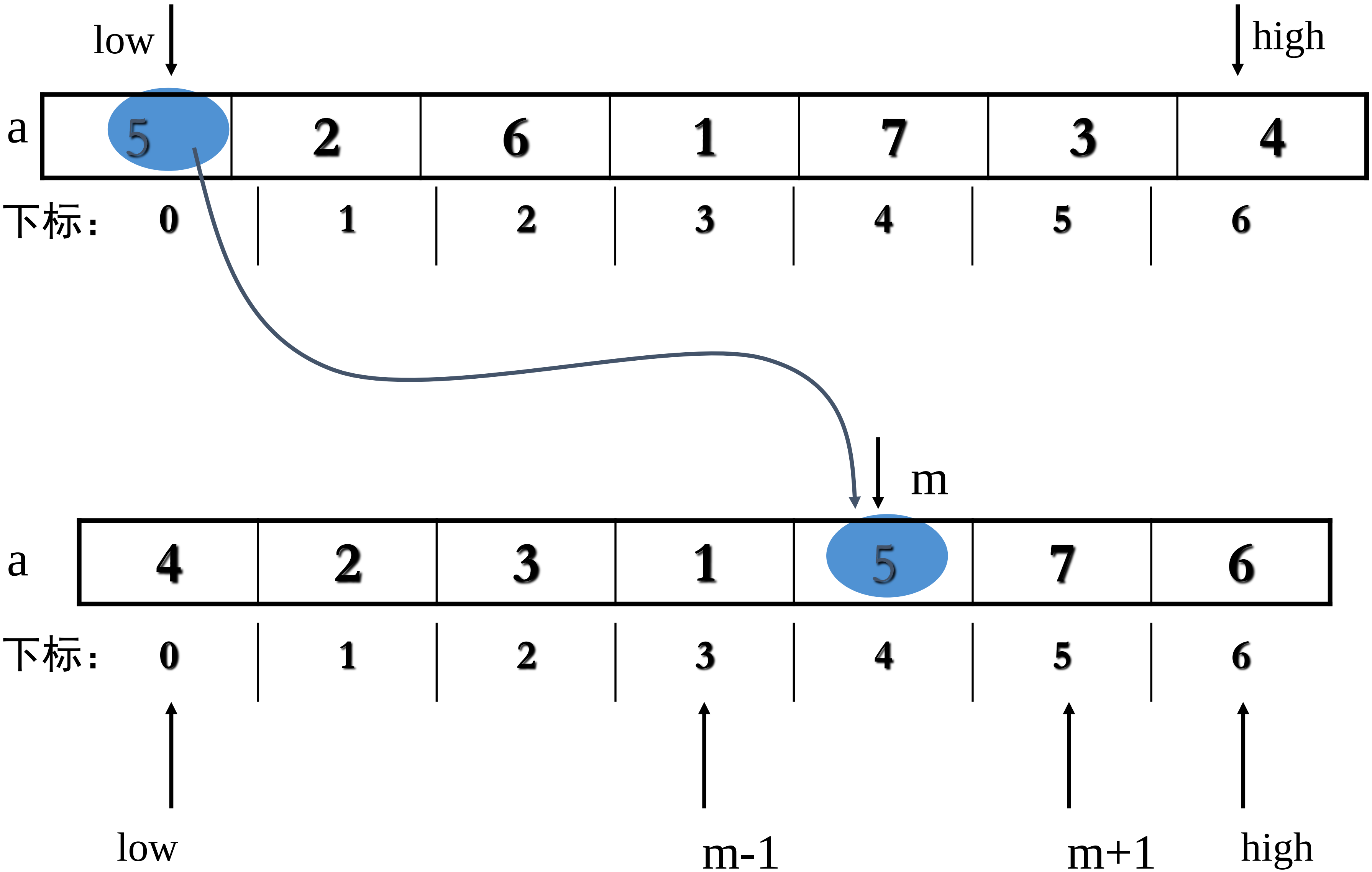
# 11.2.3 快速排序

- 基本思想:
  - 将待排序的数据放入数组a中，下标从low到high
  - 取a[low]放于变量guard中，通过分区处理，为guard选择排序后应该在的位置（>guard的数放右边，≤guard的数放左边）。当guard到达最终位置后，由guard划分左右两个集合。再用同样的思路处理左集合与右集合
  - 定义函数sort(a, low, high)：将数组a从下标为low的元素到下标为high的元素按照从小到大排序

# 11.2.3 快速排序

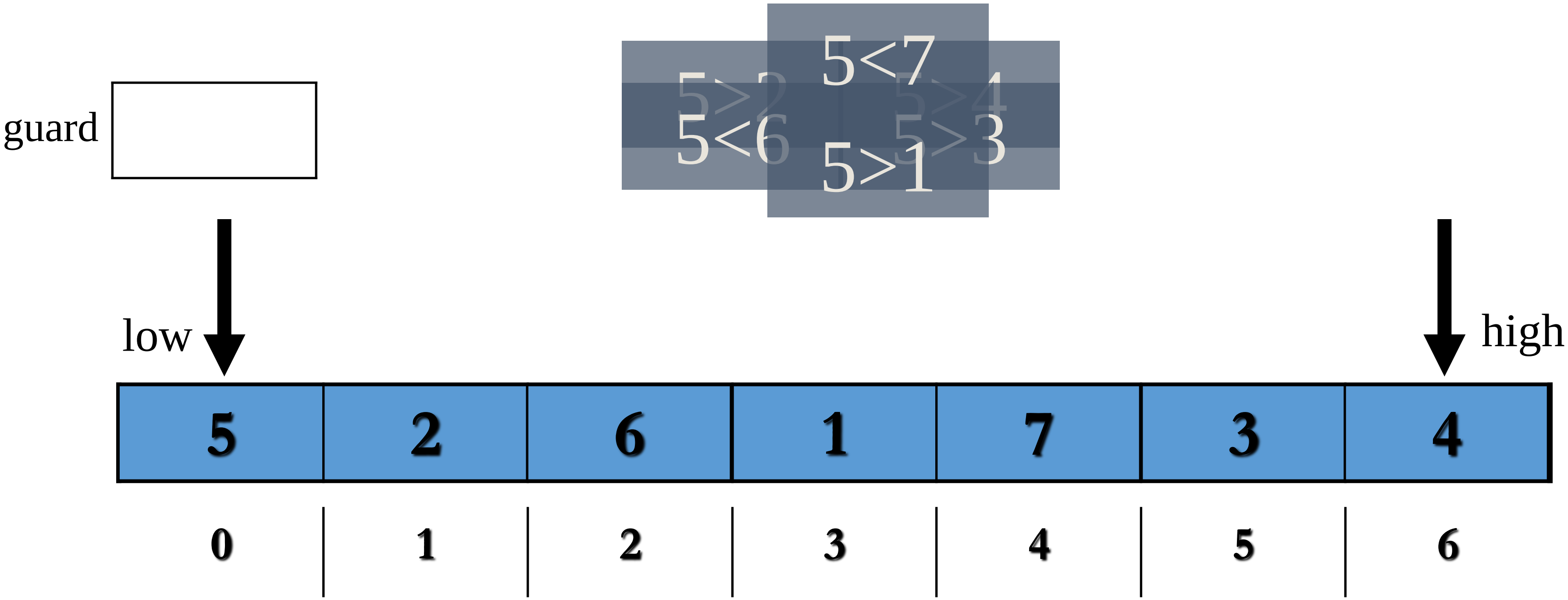


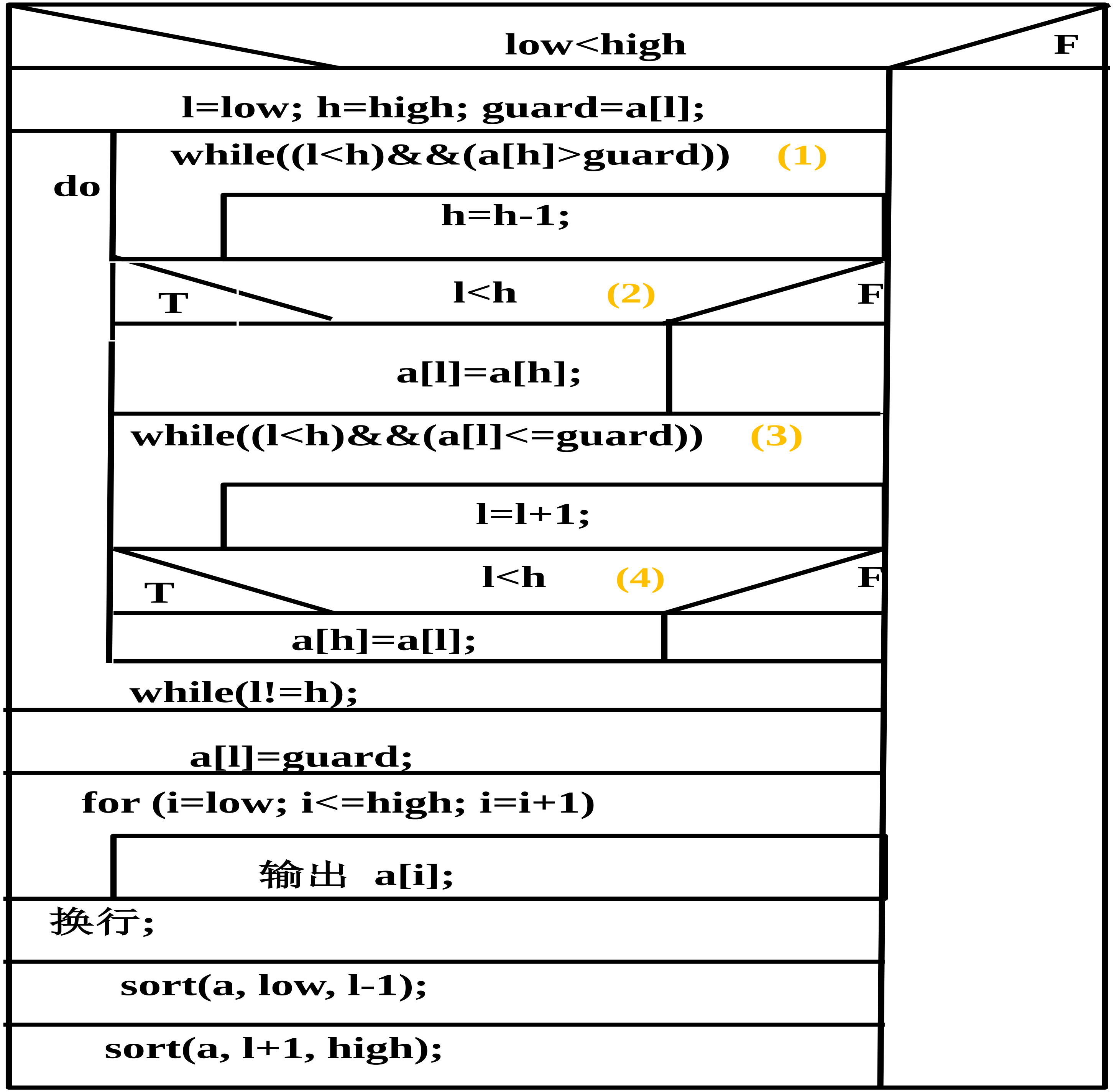
## 11.2.3 快速排序



low
high
a
5
2
6
1
7
3
4
下标：
0
1
2
3
4
5
6
m
a
4
2
3
1
5
7
6
下标：
0
1
2
3
4
5
6
low
m-1
m+1
high

# 分区过程




low<high
F
l=low; h=high; guard=a[l];
do
while((l<h)&&(a[h]>guard)) (1)
h=h-1;
T
l<h (2)
F
a[l]=a[h];
while((l<h)&&(a[l]<=guard)) (3)
l=l+1;
T
l<h (4)
F
a[h]=a[l];
while(l!=h);
a[l]=guard;
for (i=low; i<=high; i=i+1)
输出 a[i];
换行;
sort(a, low, l-1);
sort(a, l+1, high);

## 下面举例说明排序过程

数组a中有7个元素待排序

1. 让 guard = a[$\ell$] = a[0] = 5

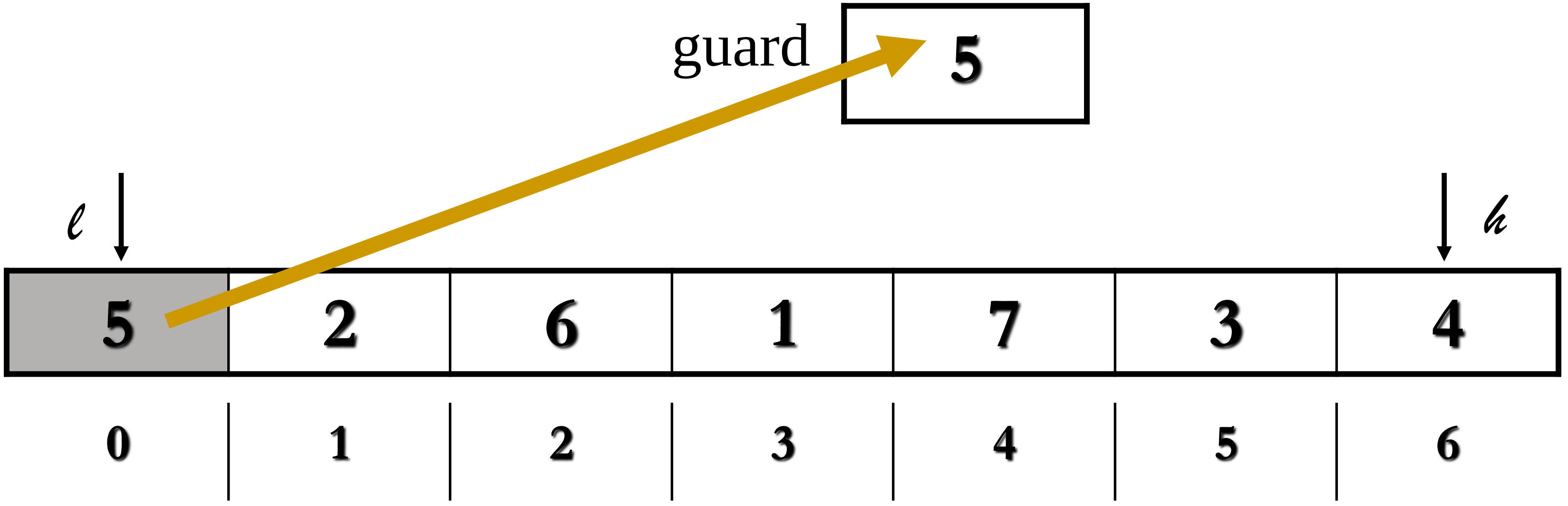



图 1

## 2、进入直到型循环

- 执行(1)：a[$h$]=a[6]=4 不满足循环条件，h不动。
- 执行(2)：$\ell < h$，做两件事：
  - ✓ a[$\ell$] = a[$h$]，即a[0] = a[6] = 4，
  - ✓ $\ell = \ell + 1 = 0+1 = 1$

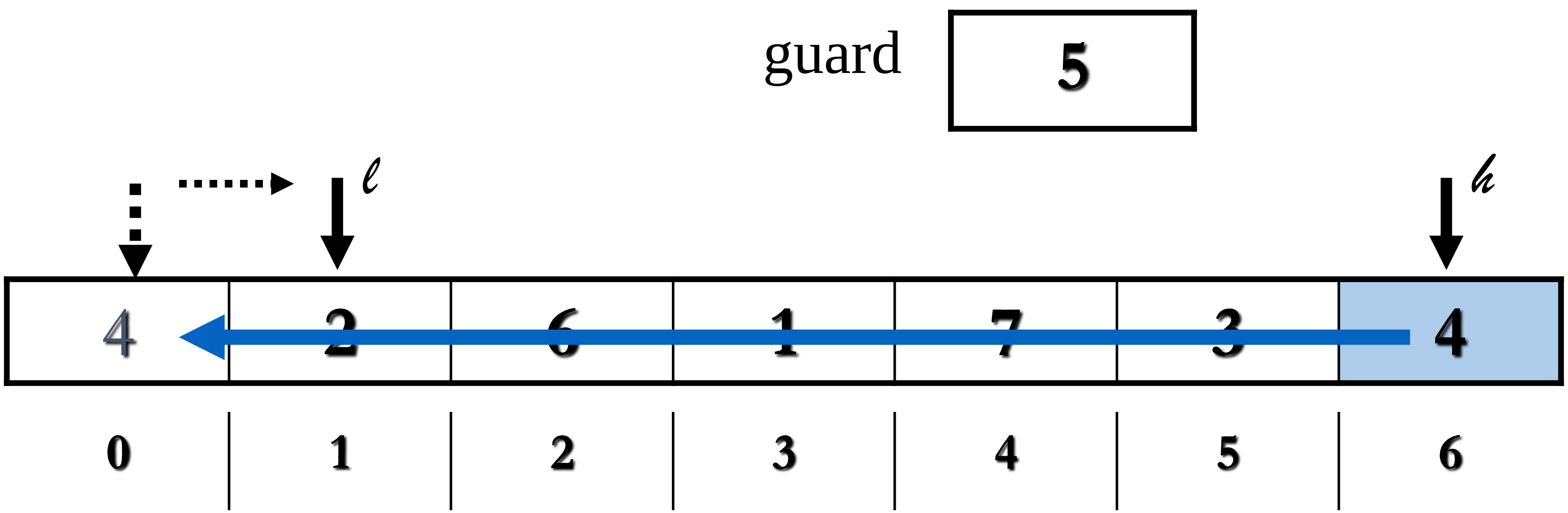



图 2

## 2、进入直到型循环

- 执行(3)：图2中的$a[\ell]<guard$，满足当循环条件，$\ell=\ell+1=2$，$\ell$增1后的情况如图3。
- 图3的情况，$a[\ell]=6>guard$，不再满足循环条件。

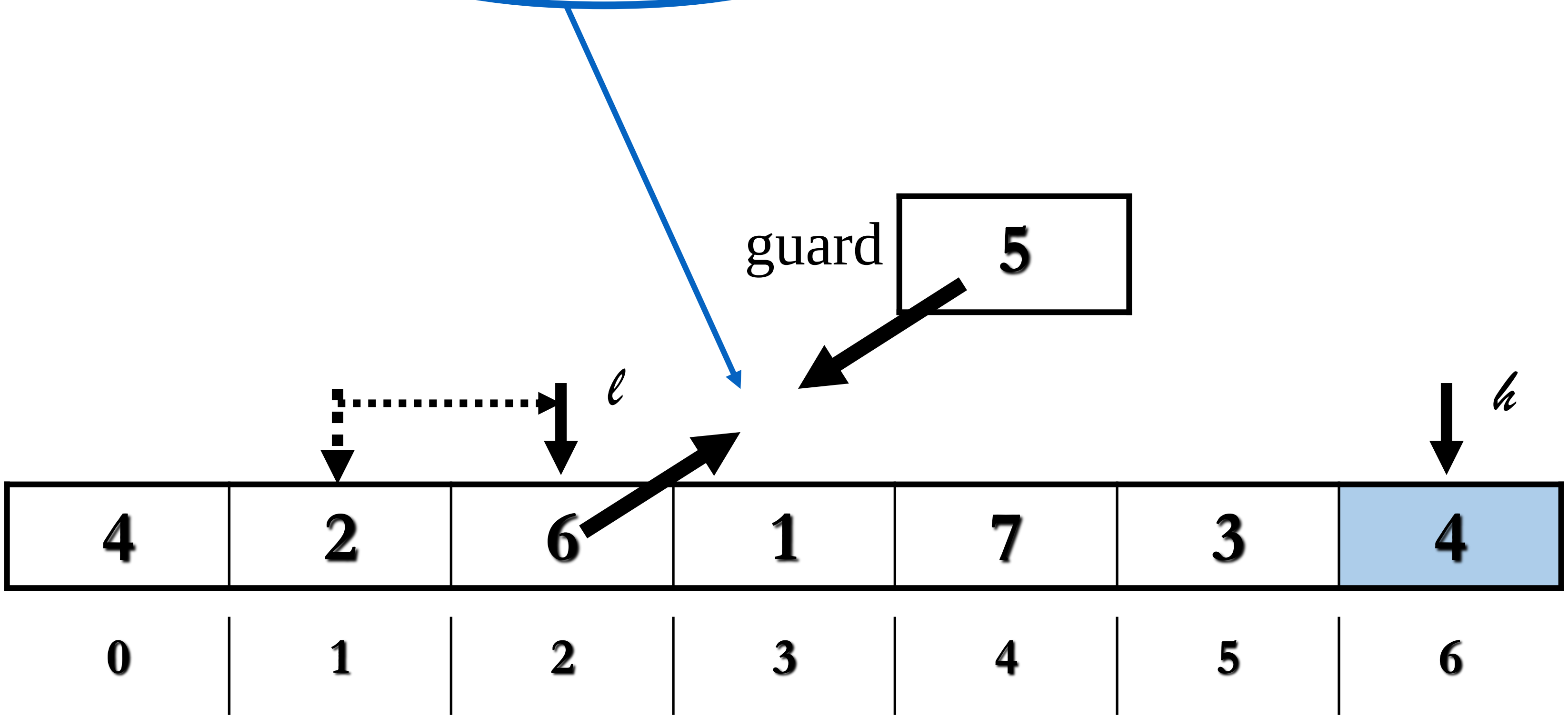



图 3

## 2、进入直到型循环

- 执行(4)：a[$h$]=a[$\ell$]，即a[6]=a[2]=6，见图4。这时$\ell$ != $h$，还得执行直到型循环的循环体

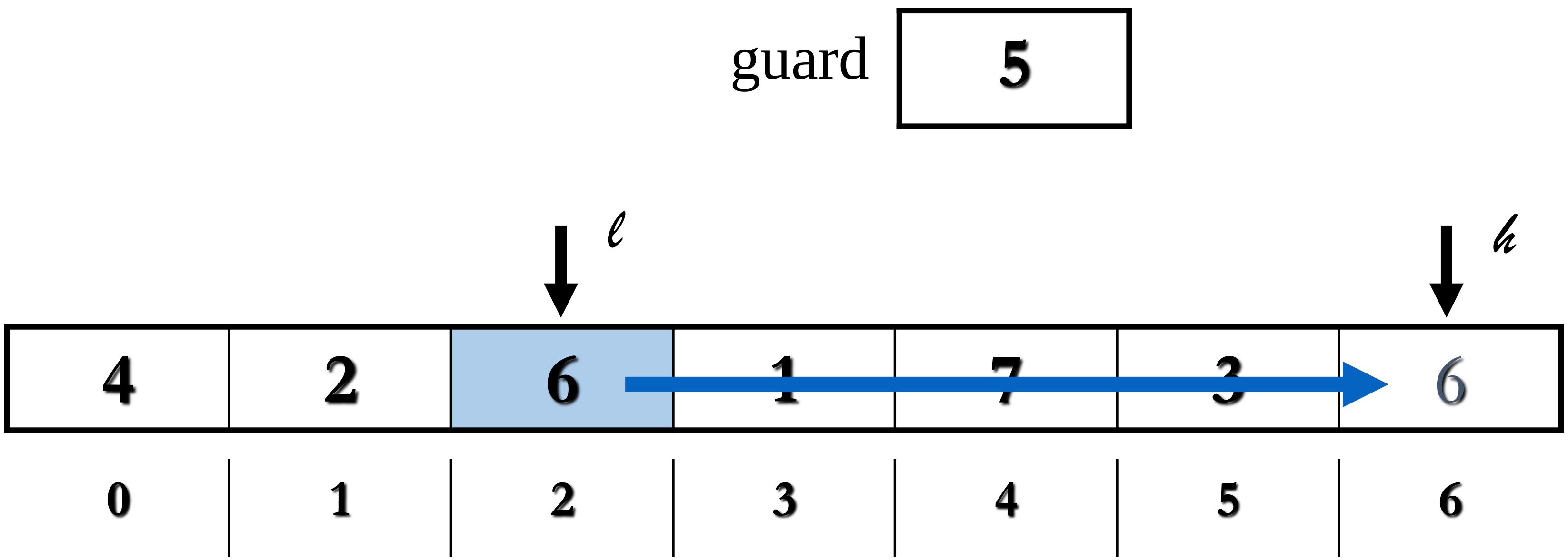



图 4

## 2、进入直到型循环

- 执行(1)：a[$h$]=a[6]=6，6>guard满足当循环的条件，$h$ = $h$-1 = 6-1 = 5，见图5
- 之后就退出当循环，因为a[$h$] = 3<guard

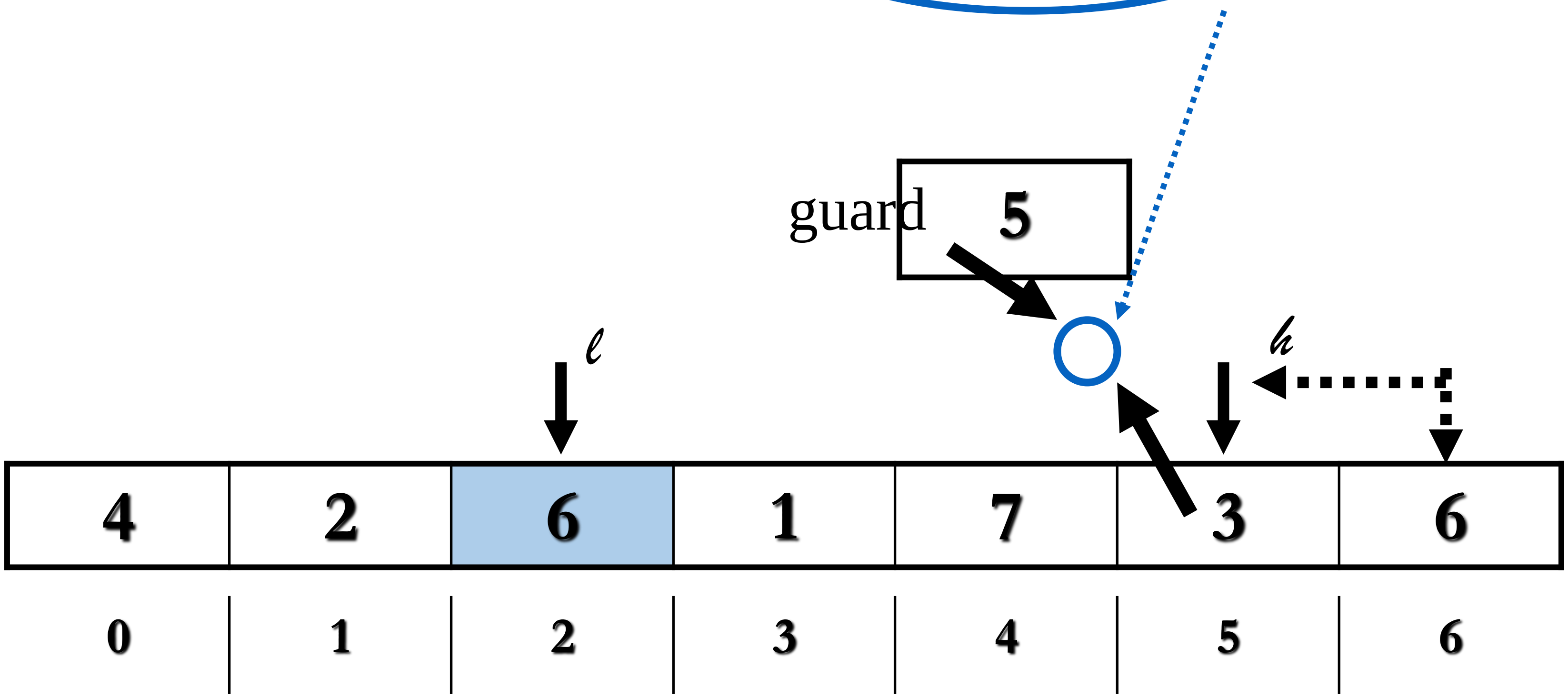



图 5

## 2、进入直到型循环

- 执行(2)：$a[\ell]=a[h]$，并让 $\ell = \ell+1=3$，见图6

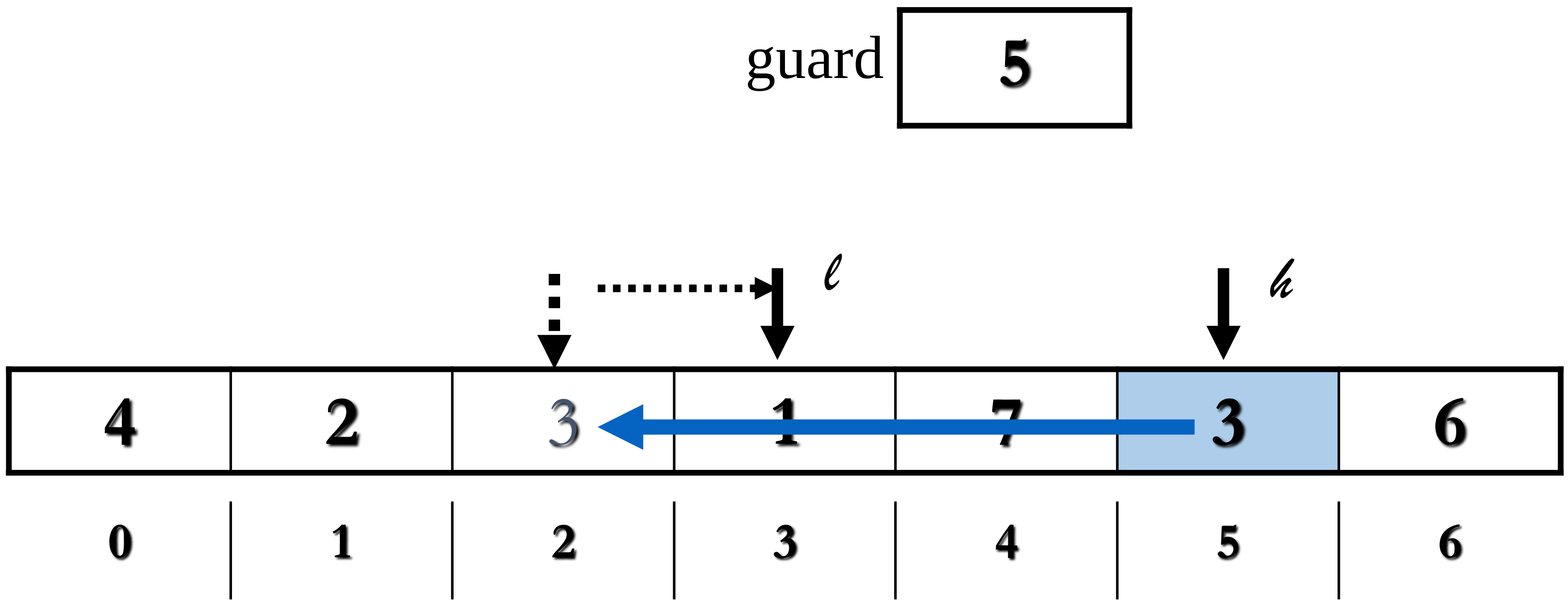
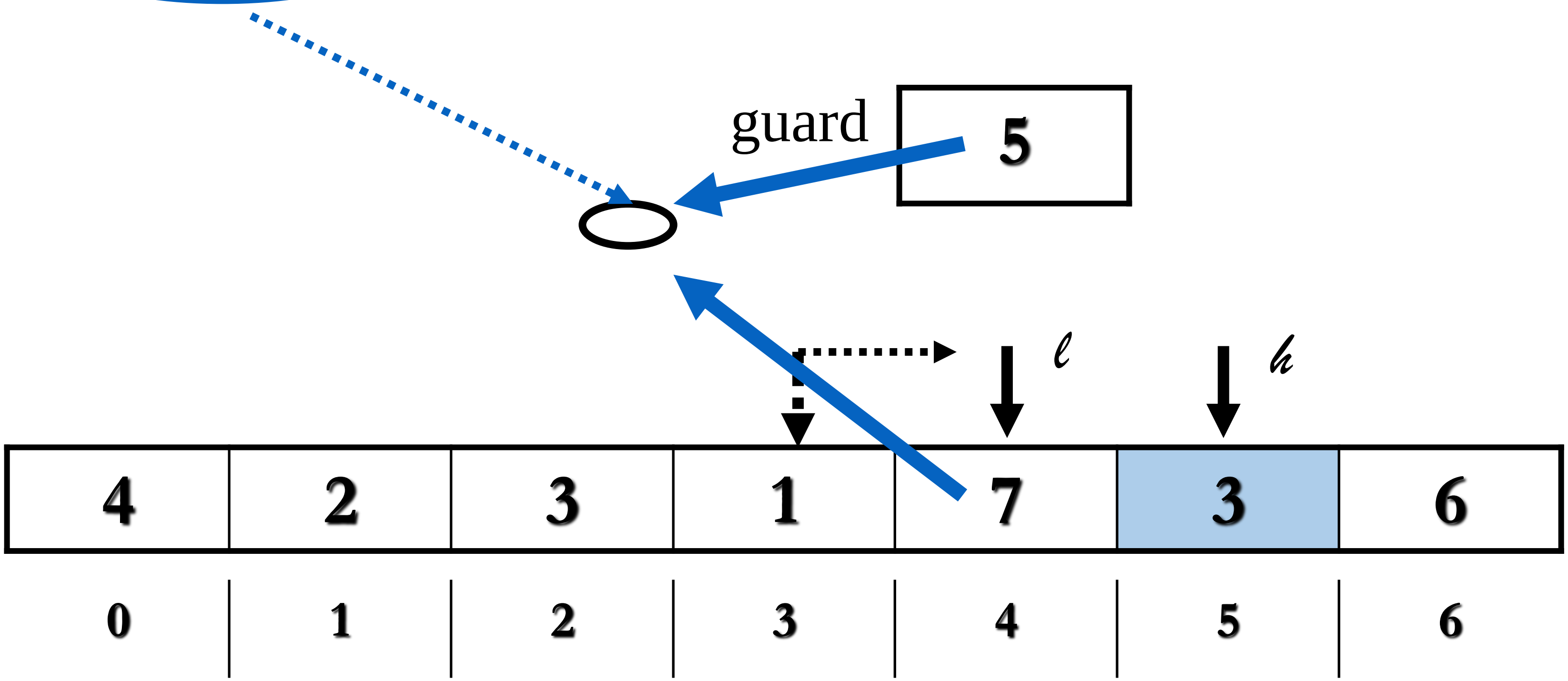



图 6

## 2、进入直到型循环

- 执行(3)：由于$a[\ell]=1<guard$，满足当循环条件，让 $\ell = \ell+1=4$。
- $a[\ell]=7>guard$，退出循环。见图7



## 2、进入直到型循环

- 执行(4)：a[$h$]=a[$\ell$]，a[5]=7。见图8
  这时仍然 $\ell<h$，应继续执行直到型循环的循环体

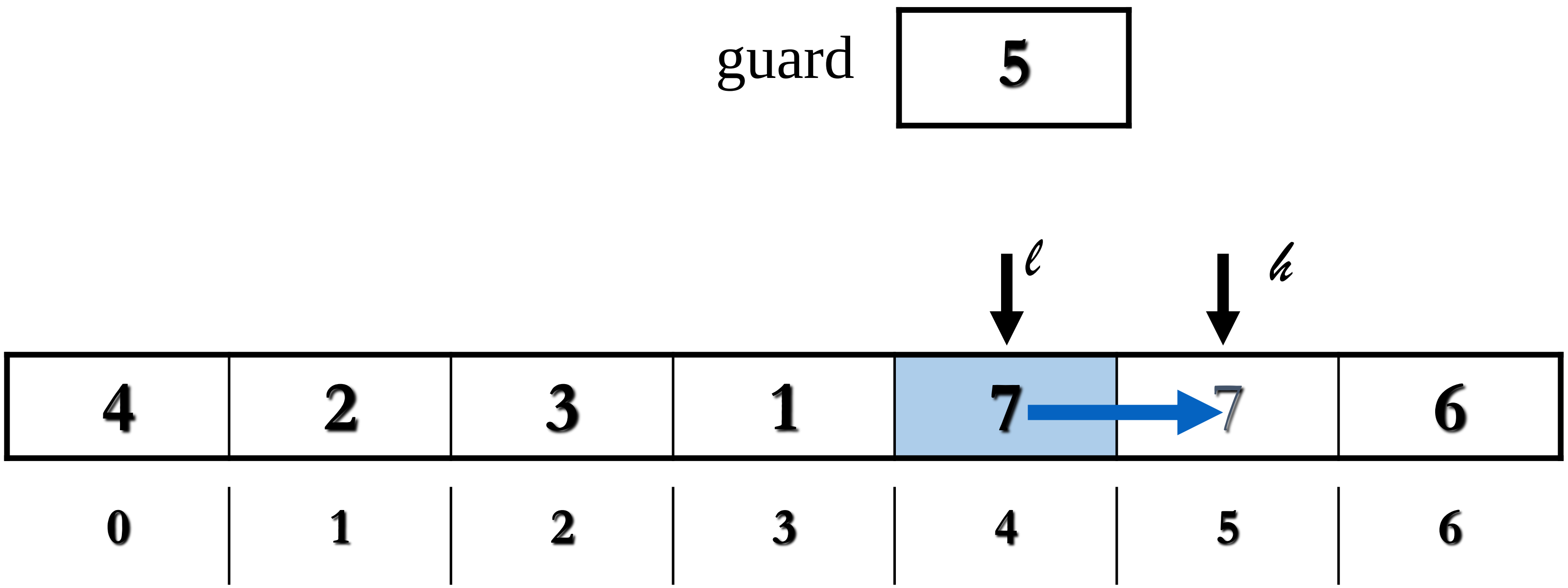



图 8

## 2、进入直到型循环

- 执行(1)，a[$h$] = 7>guard，让 $h$ = $h$-1 = 4。见图9

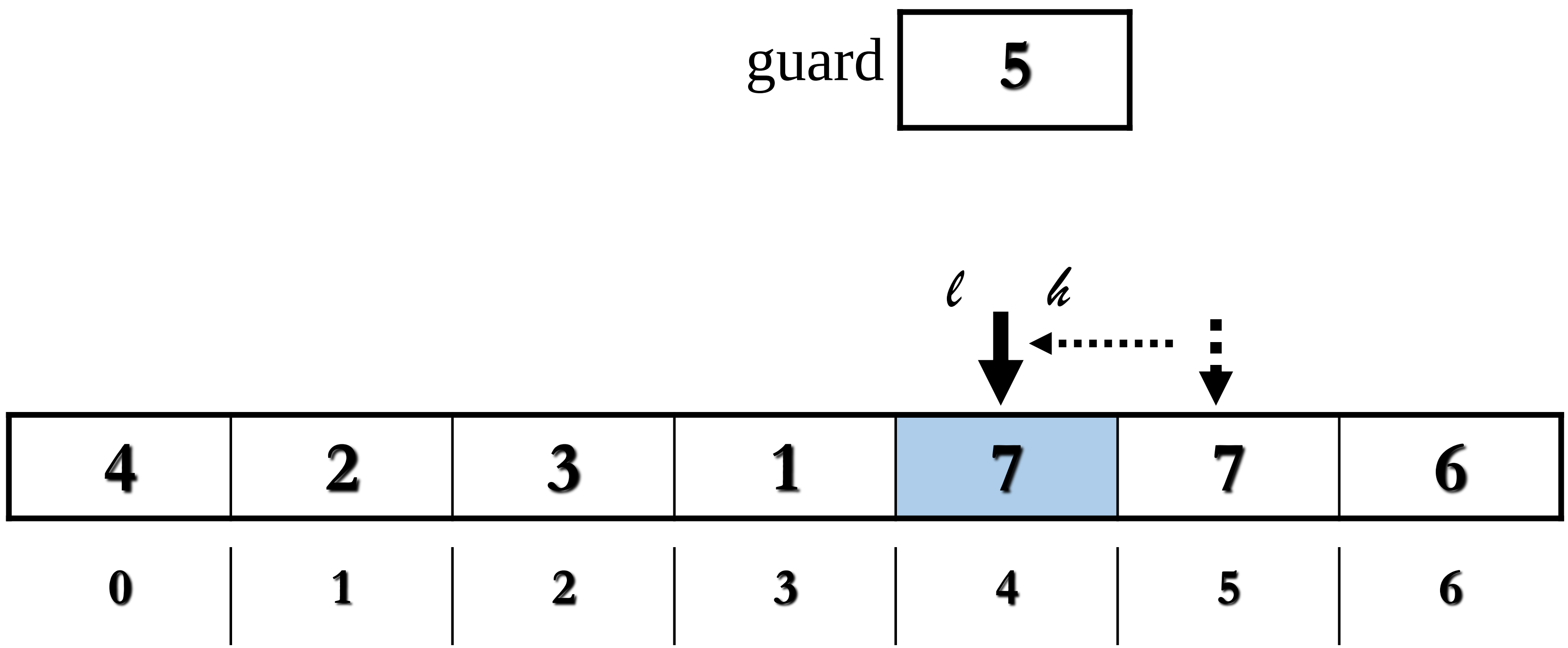



图 9

## 2、进入直到型循环

- 之后，$\ell == h$，退出直到型循环，做 a[$\ell$] = guard（即做 a[4] = 5），这是 5 的最终位置，5 将整个数据分成左右两个集合，见图10。

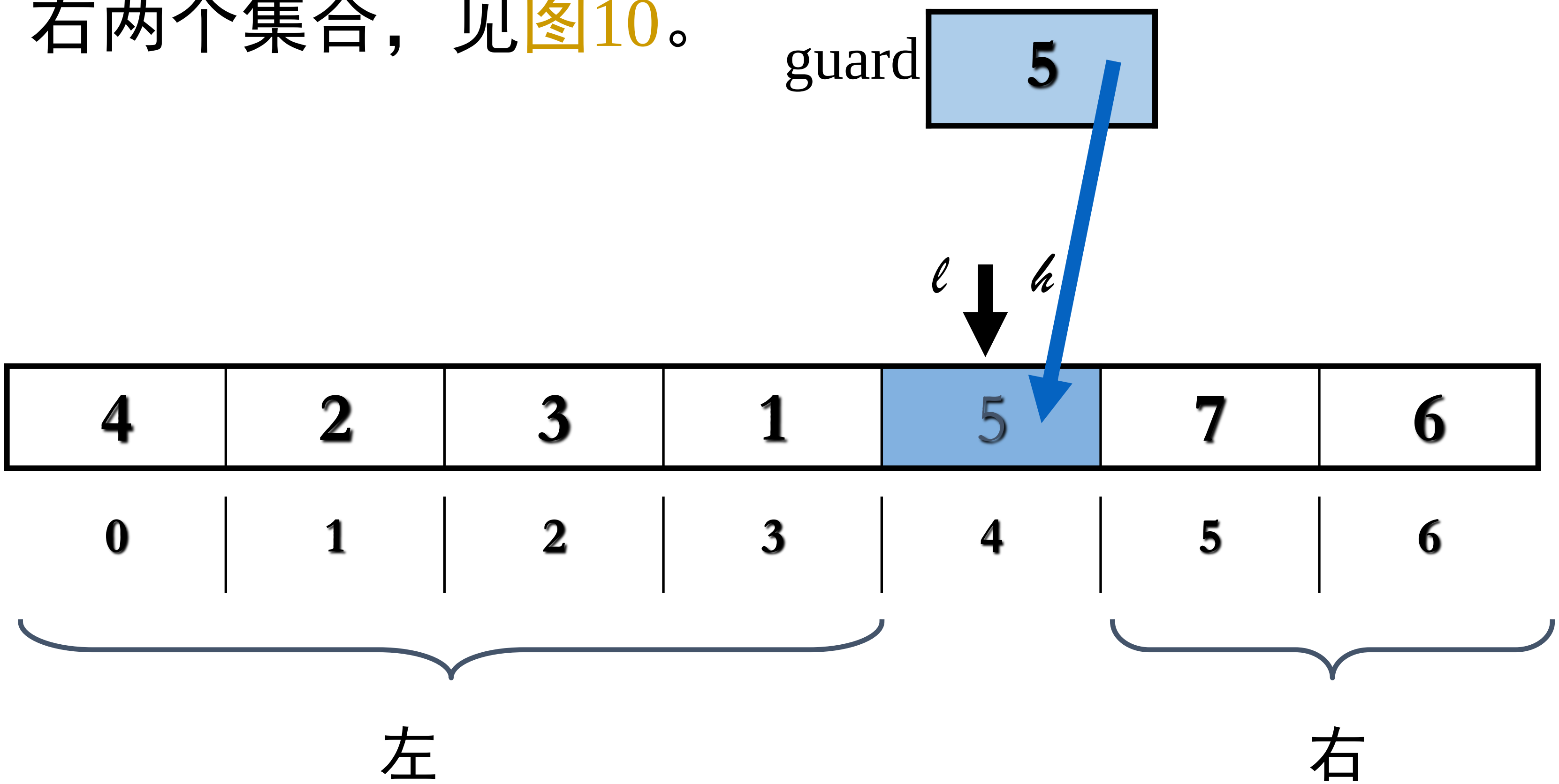
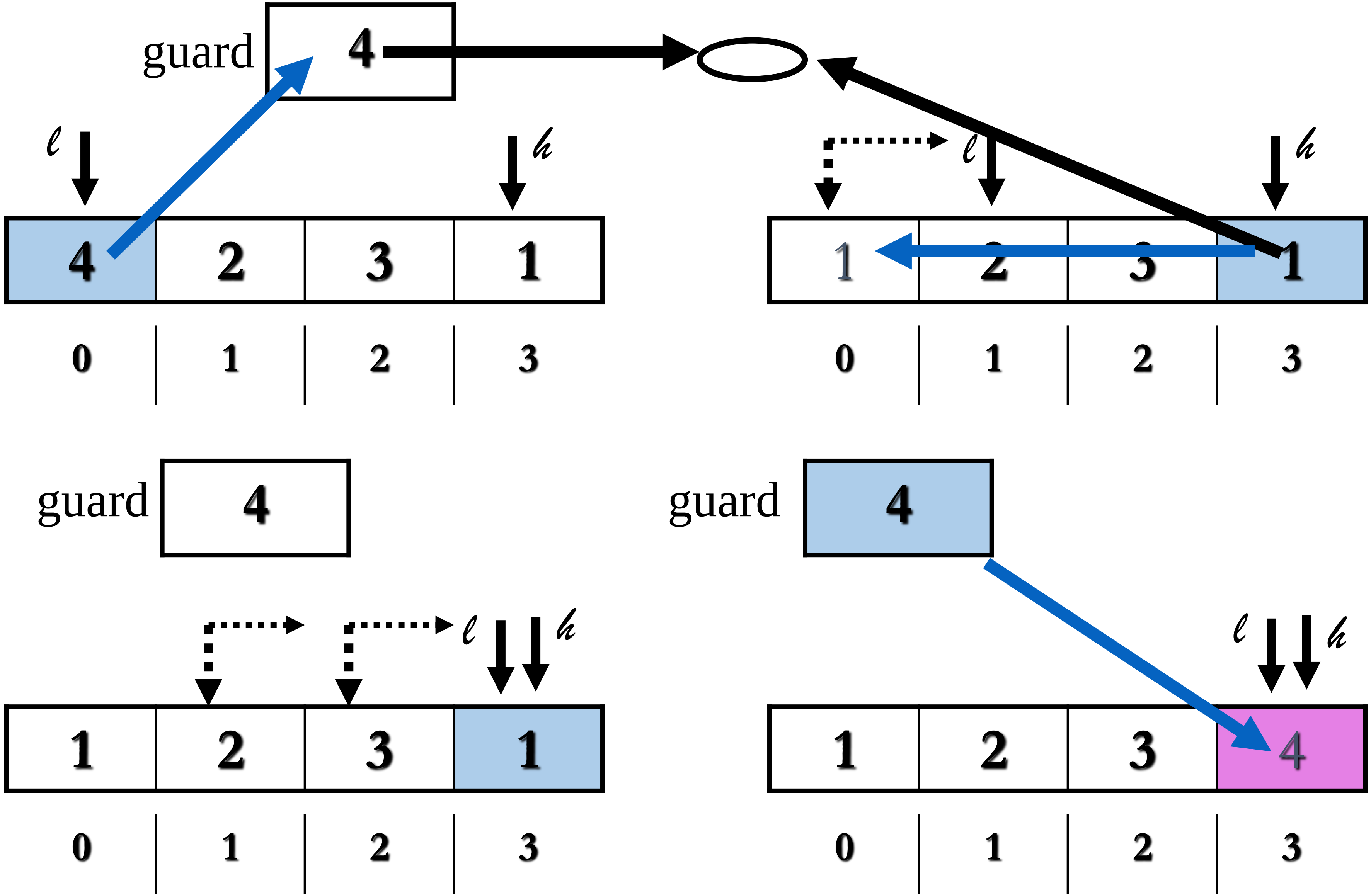
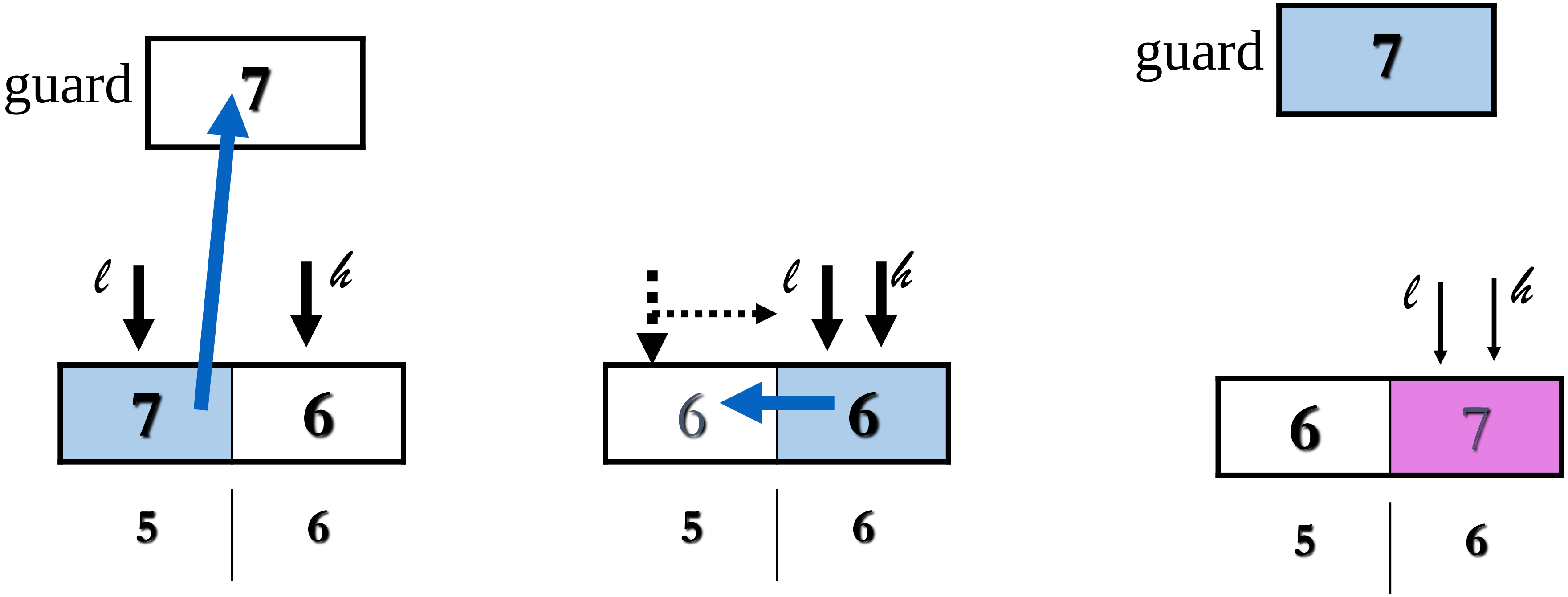



图 10

## 3、用上述思路去排左边的部分



# 4、用上述思路去排右边的部分



# 11.2.3 快速排序

```c
#include <stdio.h>
#define N 100

int partition(int[], int, int);
void quicksort(int[], int, int);

int main()
{
    int n, i;
    int a[N];

    scanf("%d", &n);
    for (i = 0; i < n; i++) scanf("%d", &a[i]);
    quicksort(a, 0, n - 1);
    for (i = 0; i < n; i++) printf("%d ", a[i]);
}

void quicksort(int a[], int l, int h)
{
    int m;
    if (l >= h)
        return;
    m = partition(a, l, h);
    quicksort(a, l, m - 1);
    quicksort(a, m + 1, h);
}
```

```c
int partition(int a[], int l, int h)
{
    int guard;
    guard = a[l];

    do
    {
        while (l < h && a[h] > guard)
            h--;
        if (l < h)
        {
            a[l] = a[h];
            l++;
        }
        while (l < h && a[l] <= guard)
            l++;
        if (l < h)
        {
            a[h] = a[l];
            h--;
        }
    } while (l != h);

    a[l] = guard;
    return l;
}
```

# 排序算法比较

| 排序方法 | 时间复杂度（平均） | 时间复杂度（最坏） | 时间复杂度（最好） | 空间复杂度 | 稳定性 |
|---|---|---|---|---|---|
| 冒泡排序 | $O(n^2)$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 稳定 |
| 选择排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 不稳定 |
| 插入排序 | $O(n^2)$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 稳定 |
| 希尔排序 | $O(n^{1.3})$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 不稳定 |
| 快速排序 | $O(nlog_2n)$ | $O(n^2)$ | $O(nlog_2n)$ | $O(log_2n)$ | 不稳定 |
| 归并排序 | $O(nlog_2n)$ | $O(nlog_2n)$ | $O(nlog_2n)$ | $O(n)$ | 稳定 |
| 堆排序 | $O(nlog_2n)$ | $O(nlog_2n)$ | $O(nlog_2n)$ | $O(1)$ | 不稳定 |
| 基数排序 | $O(n*k)$ | $O(n*k)$ | $O(n*k)$ | $O(n+k)$ | 稳定 |

- 均按从小到大排列
- k代表数值中的"数位"个数
- n代表数据规模

# 特殊的排序实例

- 数据完全逆序
- 数据完全顺序

# 数据完全逆序

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| | 9 | 7 | 6 | 5 | 3 | 2 |
| 第0趟 | 2 | 7 | 6 | 5 | 3 | 9 |
| 第1趟 | 2 | 3 | 6 | 5 | 7 | 9 |
| 第2趟 | 2 | 3 | 5 | 6 | 7 | 9 |
| 第3趟 | 2 | 3 | 5 | 6 | 7 | 9 |
| 第4趟 | 2 | 3 | 5 | 6 | 7 | 9 |

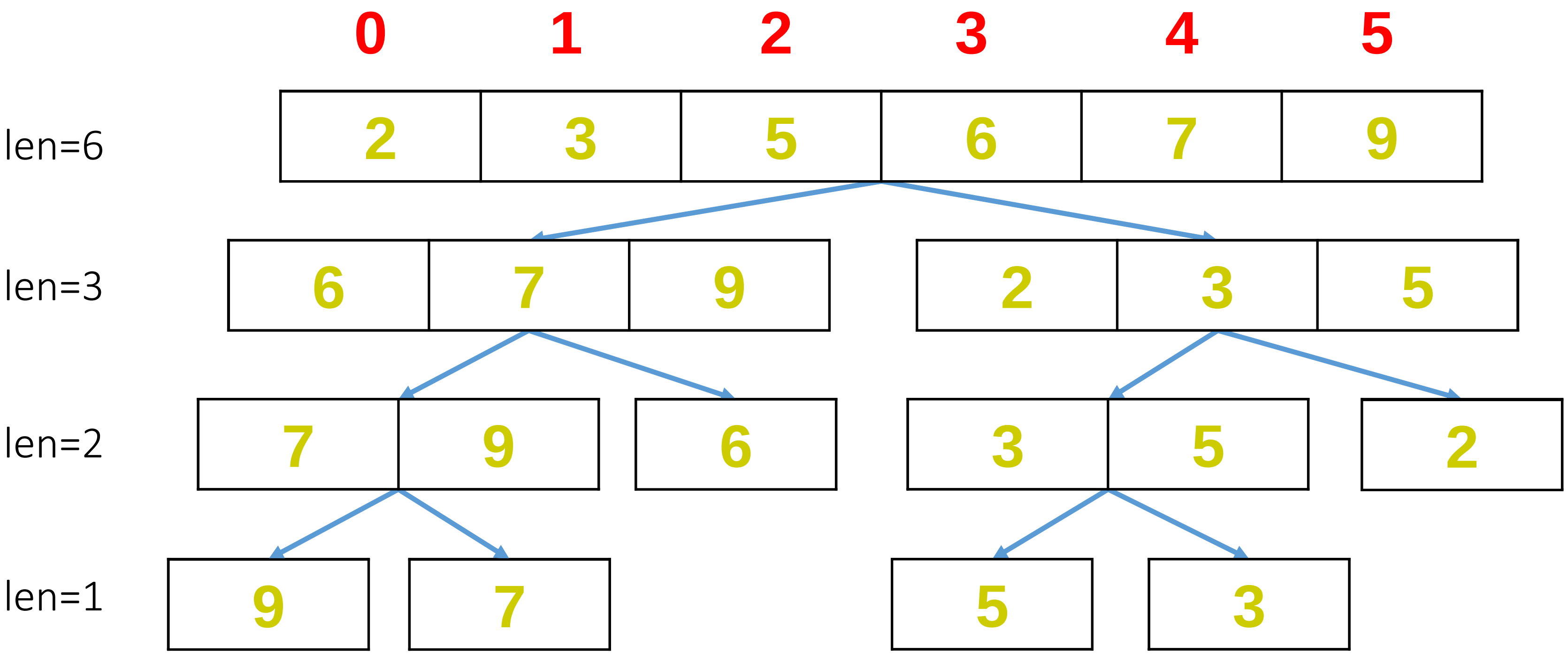
选择排序 $O(n^2)$

# 数据完全逆序

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| | 9 | 7 | 6 | 5 | 3 | 2 |
| 第0趟 | 7 | 6 | 5 | 3 | 2 | 9 |
| 第1趟 | 6 | 5 | 3 | 2 | 7 | 9 |
| 第2趟 | 5 | 3 | 2 | 6 | 7 | 9 |
| 第3趟 | 3 | 2 | 5 | 6 | 7 | 9 |
| 第4趟 | 2 | 3 | 5 | 6 | 7 | 9 |

冒泡排序 $O(n^2)$

# 数据完全逆序




归并排序　O(nlogn)

# 数据完全逆序

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| | 9 | 7 | 6 | 5 | 3 | 2 |
| 第1次分区 | 2 | 7 | 6 | 5 | 3 | 9 |
| 第2次分区 | 2 | 7 | 6 | 5 | 3 | 9 |
| 第3次分区 | 2 | 3 | 6 | 5 | 7 | 9 |
| 第4次分区 | 2 | 3 | 6 | 5 | 7 | 9 |
| 第5次分区 | 2 | 3 | 5 | 6 | 7 | 9 |

快速排序　$O(n^2)$

# 数据完全顺序

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| | 2 | 3 | 5 | 6 | 7 | 9 |
| 第0趟 | 2 | 3 | 5 | 6 | 7 | 9 |
| 第1趟 | 2 | 3 | 5 | 6 | 7 | 9 |
| 第2趟 | 2 | 3 | 5 | 6 | 7 | 9 |
| 第3趟 | 2 | 3 | 5 | 6 | 7 | 9 |
| 第4趟 | 2 | 3 | 5 | 6 | 7 | 9 |

选择排序　$O(n^2)$

# 数据完全顺序

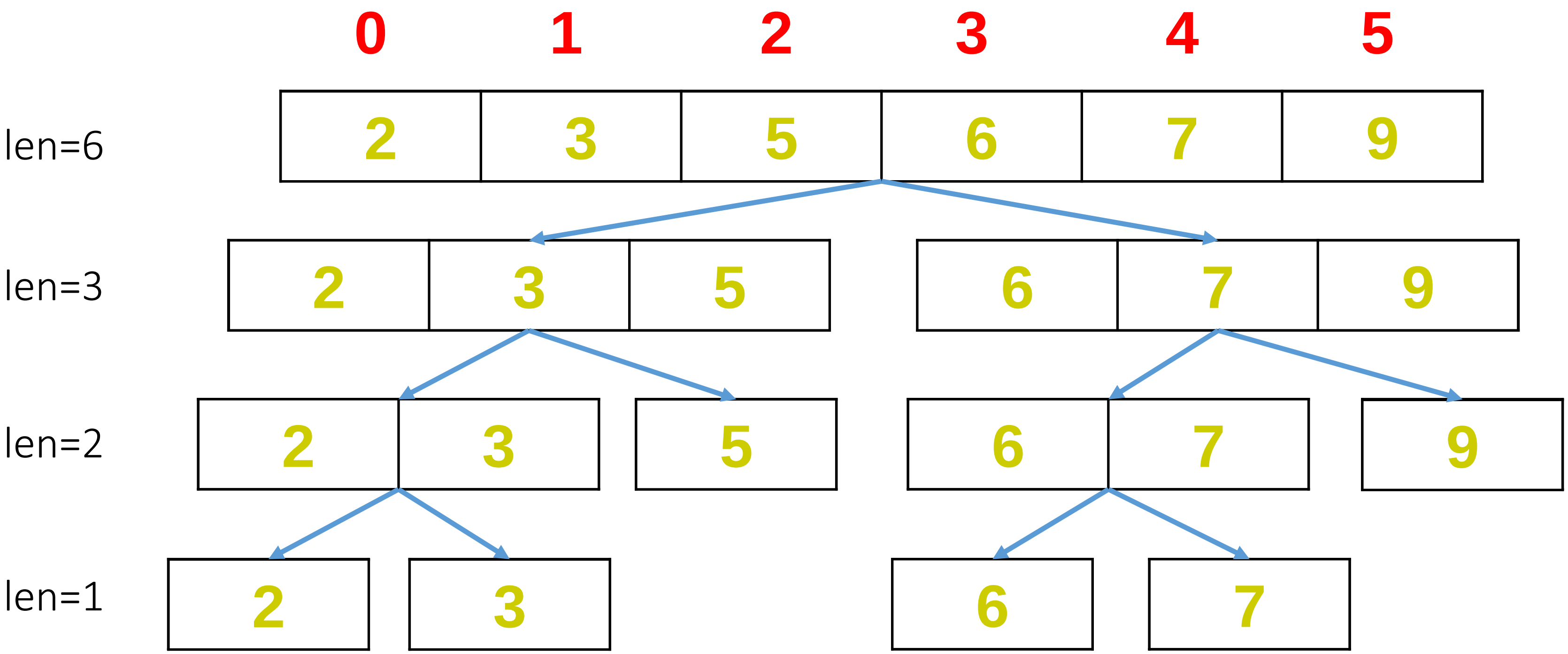
| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 3 | 5 | 6 | 7 | 9 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 第0趟 | 2 | 3 | 5 | 6 | 7 | 9 |

第0趟没有发生任何交换，带优化的冒泡排序算法会立即终止。

冒泡排序　O(n)

# 数据完全顺序




归并排序 O(nlogn)

# 数据完全顺序

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| | 2 | 3 | 5 | 6 | 7 | 9 |
| 第1次分区 | 2 | 3 | 5 | 6 | 7 | 9 |
| 第2次分区 | 2 | 3 | 5 | 6 | 7 | 9 |
| 第3次分区 | 2 | 3 | 5 | 6 | 7 | 9 |
| 第4次分区 | 2 | 3 | 5 | 6 | 7 | 9 |
| 第5次分区 | 2 | 3 | 5 | 6 | 7 | 9 |

快速排序 $O(n^2)$