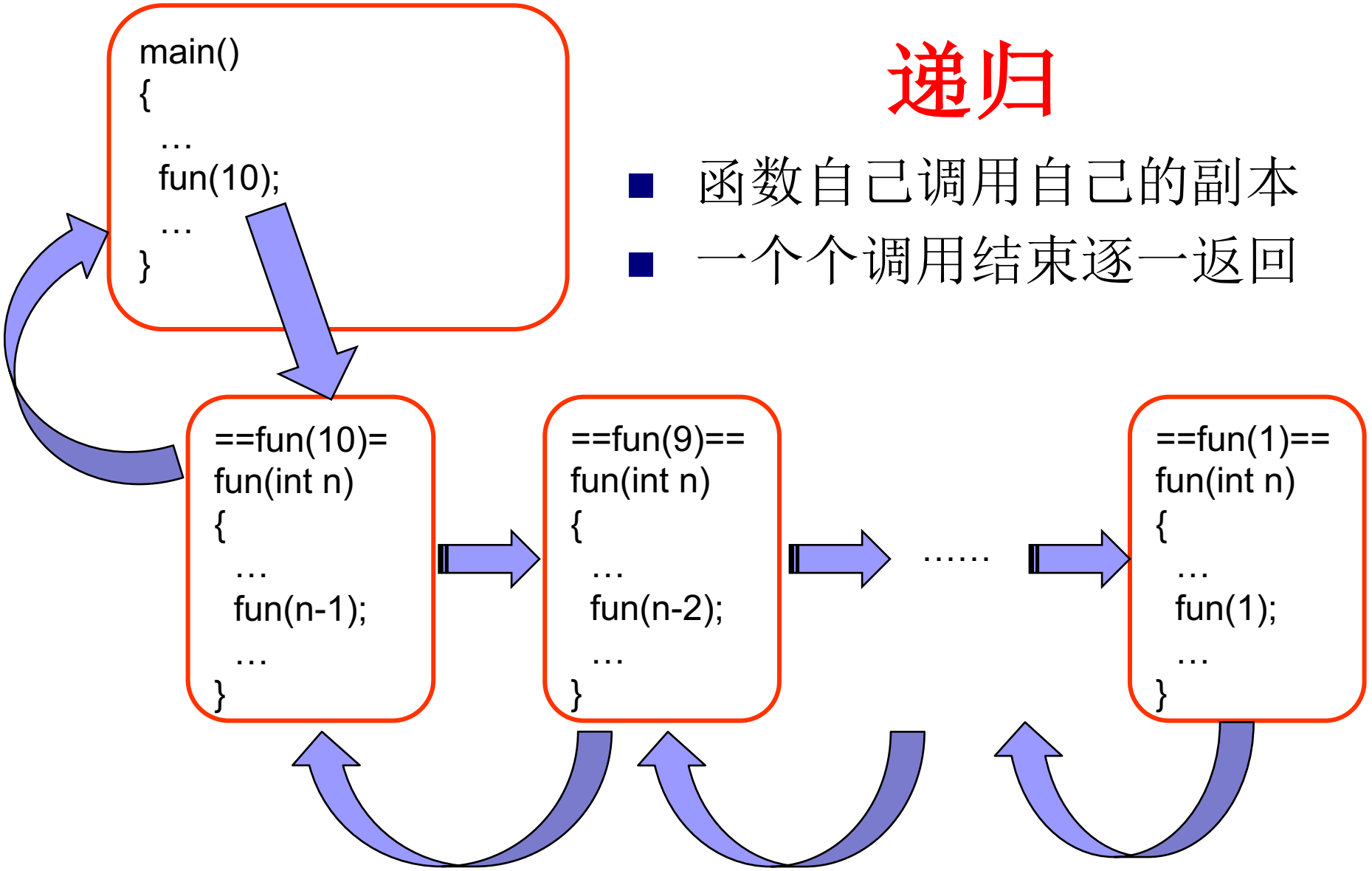
专题三
模块化程序设计

Part A

专题三 模块化程序设计

- 递归程序设计 (10.2)
  - 递归函数基本概念 (10.2.2)
  - 递归程序设计 (10.2.1, 10.2.3)
  - 拓展内容：用递归实现枚举所有可能性
- 编译预处理 (10.3)
- 大程序构成 (10.4)
- 大程序开发技巧

函数调用自己是什么调用？

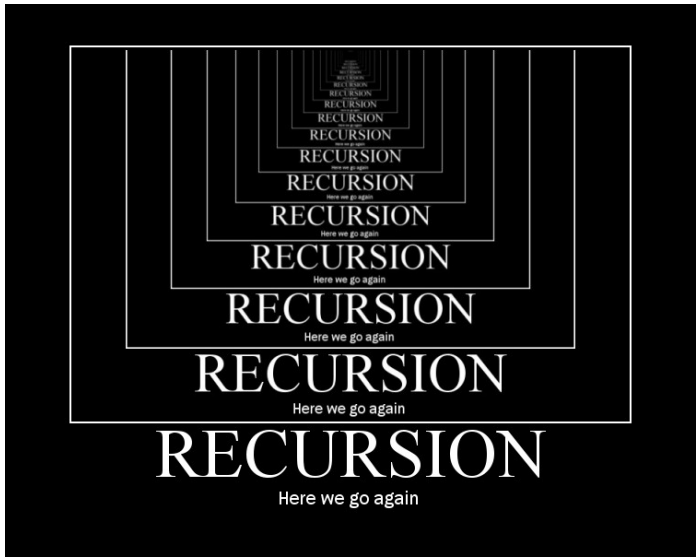


递归

- 函数自己调用自己的副本
- 一个个调用结束逐一返回

递归函数基本概念 (10.3.2)

- 递归是一种将问题简化为相同形式的较小问题来解决问题的技术
- 递归函数是调用自身的函数
- 用递归解决问题需要满足两个条件
  - 问题可以逐步简化成自身较简单的形式(递归式)
  - 递归最终能结束(递归出口)



递归数据表示
二级指针
二维数组
链表

递归函数基本概念

- 函数的直接递归调用 vs. 函数的间接递归调用

```
int f(int x)
{
    int y;
    ……
    y = f(x);
    ……
    return y;
}
```

```
int f(int x)
{
    int y;
    ……
    y = g(x);
    ……
    return y;
}
int g(int x)
{
    int z;
    ……
    z = f(x);
    ……
    return z;
}
```

- 递归与嵌套调用
  - 递归 → 嵌套调用
  - 嵌套调用 ↛ 递归

递归函数基本概念

- [例10-2] 用递归函数实现求n!
  - 递推法
    - 在学习循环时，计算n!采用的就是递推法
      n! = 1×2×3×…×n
    - 用循环语句实现

```
result = 1;
for (i = 1; i <= n; i++) {
    result = result * i;
}
```

  - 递归法
    - n! = n ×(n-1)!　　当n＞1　　递归式子
      = 1　　当n=1或n=0　　递归出口
    - 求n!可以在(n-1)!的基础上再乘上n (n → n-1 较小问题)
    - 如果把求n!写成函数fact(n)，则fact(n)的实现依赖于fact(n-1)

例10-2 源程序

```
#include <stdio.h>
double  fact (int n);
int main (void)
{   int n;
    scanf ("%d", &n);
    printf ("%f", fact (n));
    return 0;
}
double  fact (int n)              /*  函数定义  */
{  double result;
    if (n == 1 || n == 0)         /*  递归出口  */
        result = 1;
    else
        result = n * fact(n-1);   /*  递归定义  */
   return result;
}
```

例10-2 分析

求n! 递归定义

$n! = n \times (n-1)!$ (n > 1)

$n! = 1$ (n = 0,1)

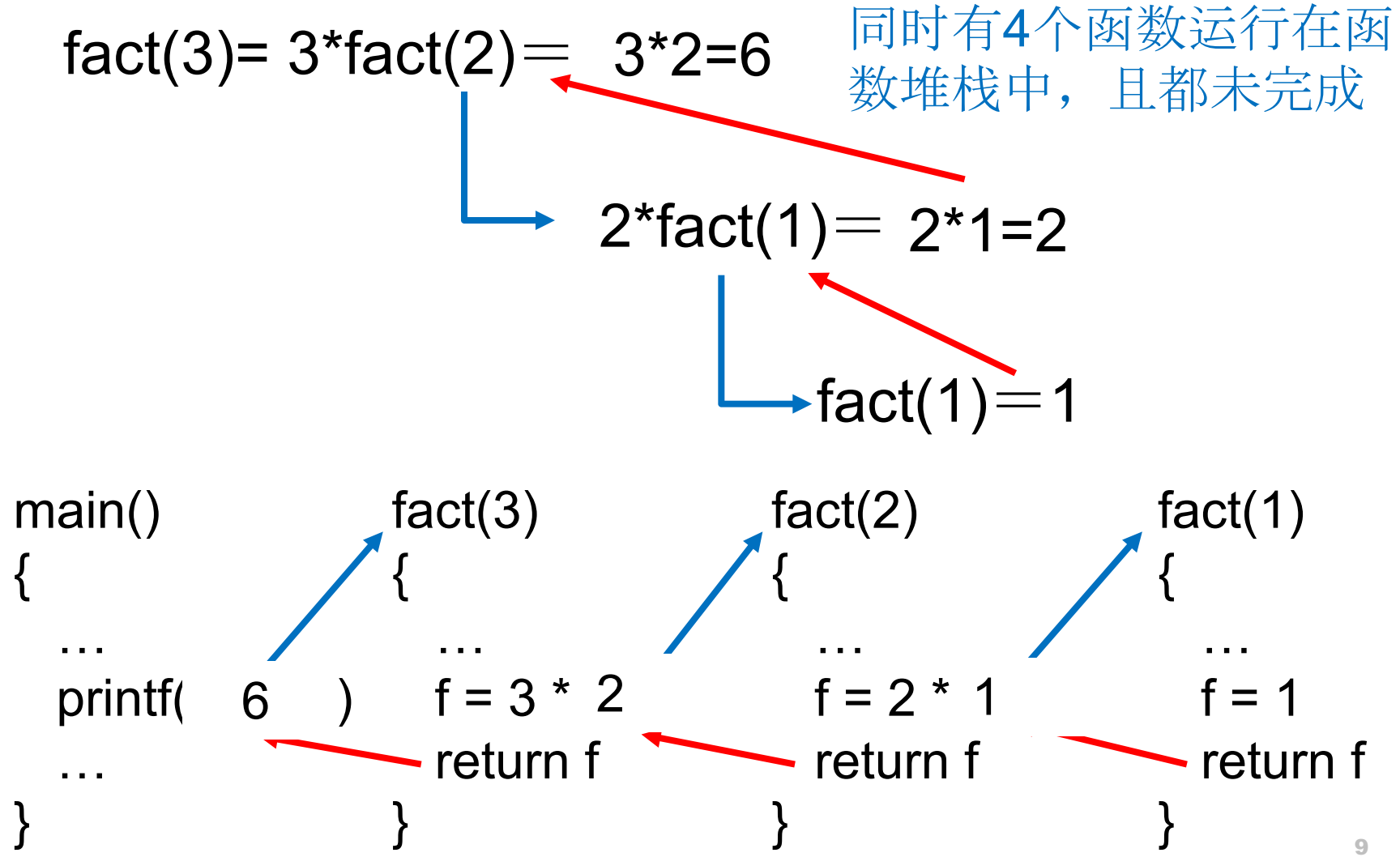
```
#include <stdio.h>
double  fact (int n);
int main (void)
{   int n;
    scanf ("%d", &n);
    printf ("%f", fact (n));
    return 0;
}
double  fact (int n)            /*  函数定义  */
{  double result;
    if (n == 1 || n == 0)       /*  递归出口  */
        result = 1;
    else
        result = n * fact(n-1);  /*  递归定义  */
   return result;
}
```

递归出口

递归式

~~fact(n) = n * fact(n-1);~~

递归函数fact(n)的求解过程

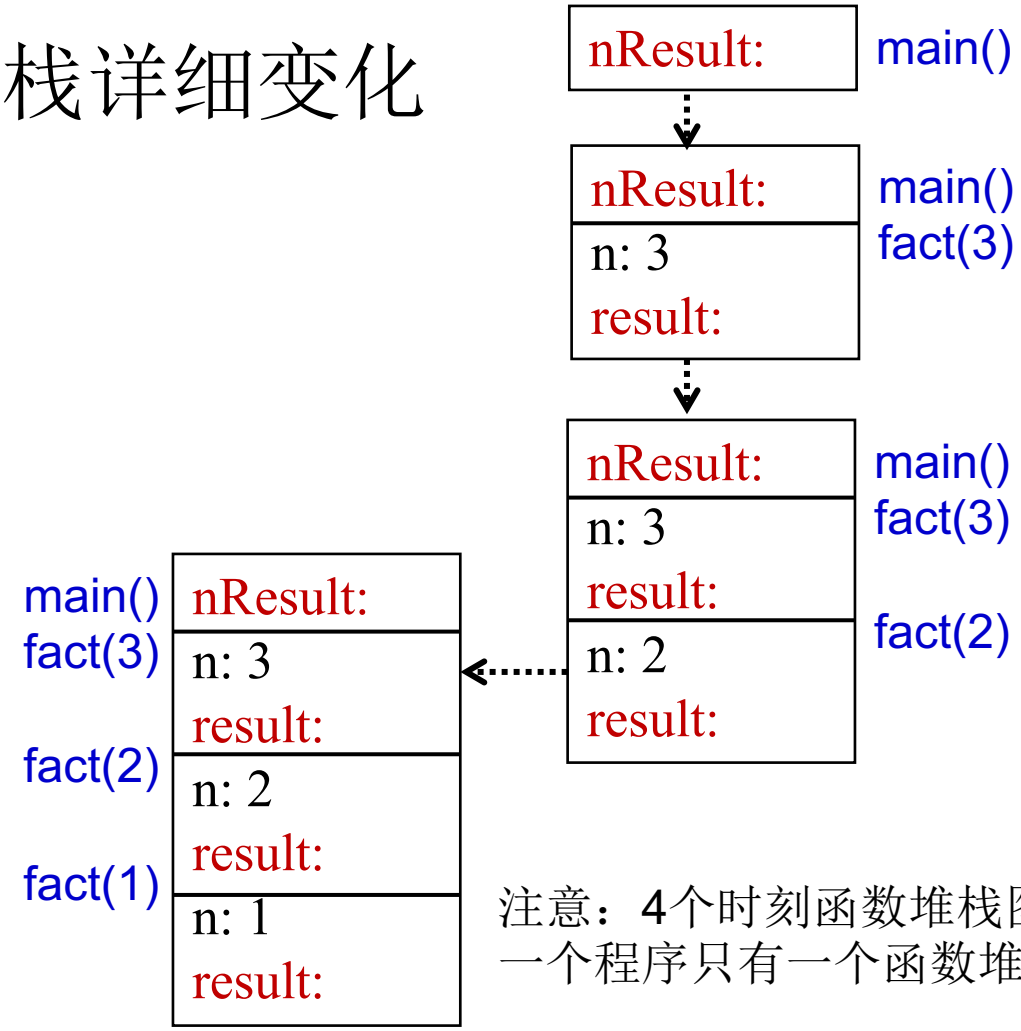


递归函数fact(n)的求解过程

- 调用过程内存堆栈详细变化

```
int main()
{
    int nResult = fact(3);
    printf("%d\n", nResult);
    return 0;
}

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



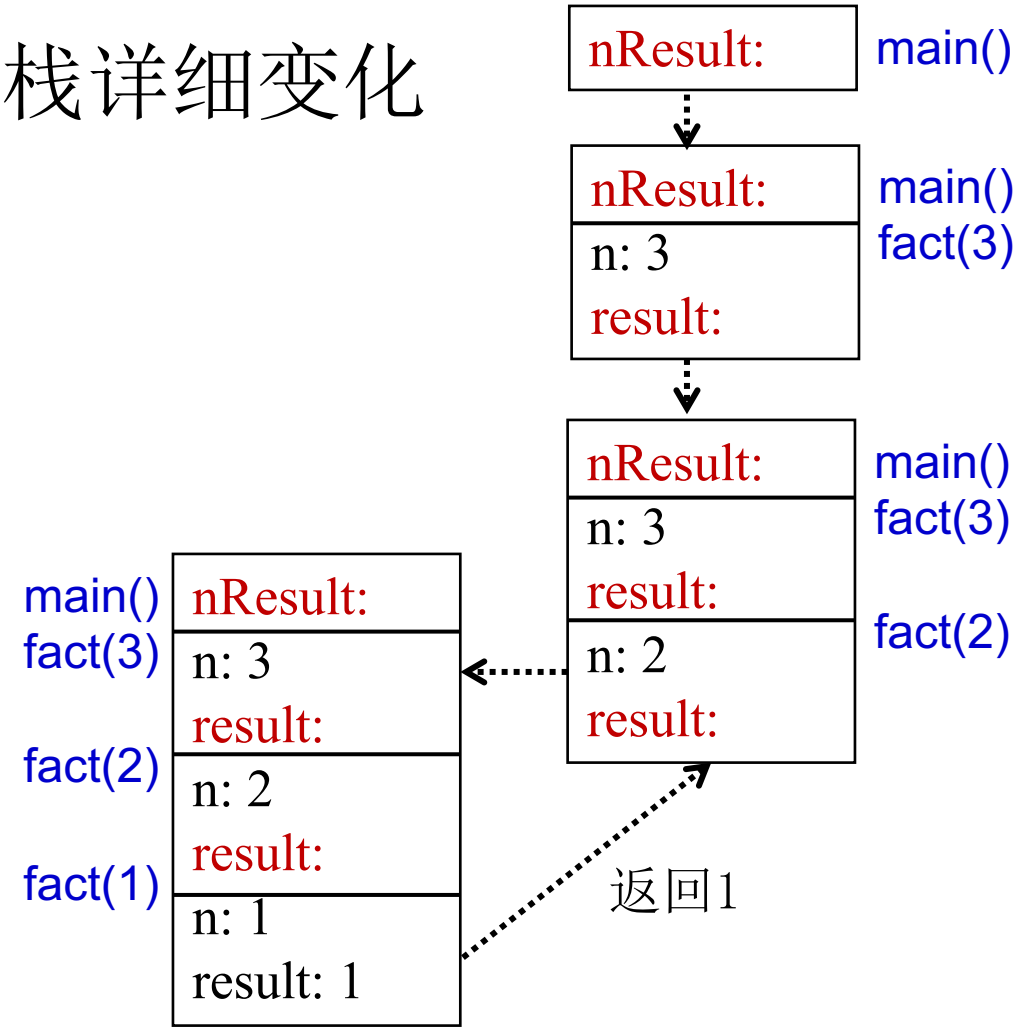
注意：4个时刻函数堆栈图，一个程序只有一个函数堆栈

递归函数fact(n)的求解过程

- 调用过程内存堆栈详细变化

```
int main()
{
    int nResult = fact(3);
    printf("%d\n", nResult);
    return 0;
}

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

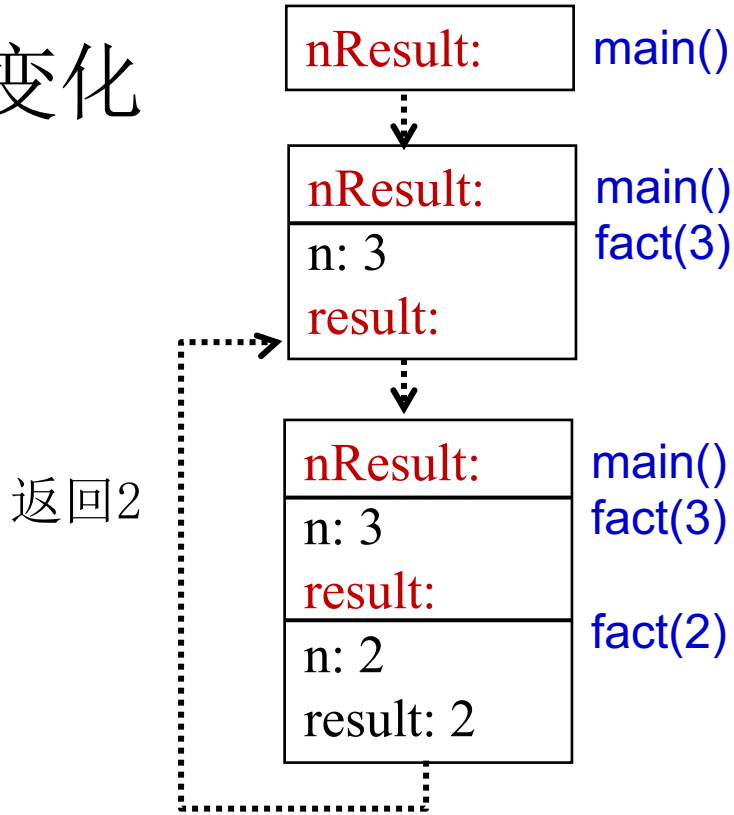


递归函数fact(n)的求解过程

- 调用过程内存堆栈详细变化

```
int main()
{
    int nResult = fact(3);
    printf("%d\n", nResult);
    return 0;
}

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

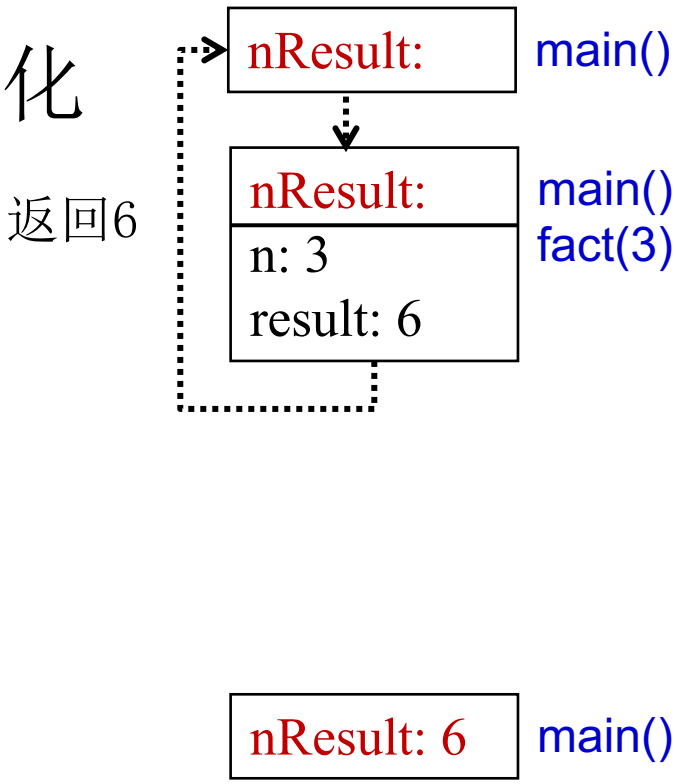


递归函数fact(n)的求解过程

- 调用过程内存堆栈详细变化

```
int main()
{
    int nResult = fact(3);
    printf("%d\n", nResult);
    return 0;
}

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



可视化演示递归过程

https://www.cs.usfca.edu/~galles/visualization/RecFact.html

递归函数基本的概念

- 递归函数的正确性 (Leap of Faith)　数学归纳法
  - 递归出口正确，递推式正确
- 不要忘记写递归出口　int factorial(int n) { return n * factorial(n-1); }
  - 函数会永远调用下去，直到操作系统为程序预留的栈空间耗尽程序崩溃为止，这称为无穷递归
- 何时使用递归？
  - 对于某一小范围内的问题，使用递归会带来简单、优雅的解；对于大多数问题，它所带来的解将会是极其复杂的，因此要有选择地使用递归
  - 大部分搜索问题都可以使用递归 + 回溯实现

递归函数基本的概念

- 调用递归函数：调用另一个有着相同名字和相同代码的函数
- 每次调用函数时分配参数和局部变量的存储空间，退出函数时释放
- 随着递归函数的层层深入，存储空间的一端逐渐增加，然后随着函数调用的层层返回，存储空间的这一端又逐渐缩短
- 递归存在着可用堆栈空间过度使用的危险
  - 递归出口错误，将导致堆栈溢出，即段错误
  - 在安全相关系统中强制规定：不能使用递归函数

递归程序设计 (10.3.3)

用递归实现的问题，满足两个条件：

- 问题可以逐步简化成自身较简单的形式 (递归式)

$$n! = n * (n-1)!$$

$$\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i$$

- 递归最终能结束 (递归出口)

递归应用
例10-2 阶乘
例10-3 最大公约数
例10-4 整数逆序输出
例10-5 汉诺塔问题
例10-6 分治法求解金块问题
链表操作的递归实现

两个条件缺一不可

解决递归问题的两个着眼点

递归程序设计

- [例10-3] 定义函数gcd(m, n)，用递归求m和n的最大公约数
  - 辗转相除法 (欧几里得算法)

    (1) r = m % n

    (2) 若r为0，则返回n的值；否则转第(3)步

    (3) m = n, n = r, 返回第(1)步
- 递归实现的两个关键点
  - 递归出口：n　　当 m%n = 0
  - 递归式子：递归调用gcd(n, m%n)　　当 m%n != 0

[例10-3] 源程序

```
int gcd(int m, int n)
{
    if (m % n == 0) {                    /* 递归出口 */
        return n;
    } else {
        return gcd(n, m % n);            /* 递归调用 */
    }
}
```

[例10-4] 递归实现将整数逆序输出

- 编写递归函数reverse(int n)实现将整数n逆序输出

  分析：
  - 将整数n逆序输出可以用循环实现，且循环次数与n的位数有关
  - 递归实现整数逆序输出也需要用位数作为控制点。归纳递归实现的两个关键点如下：
    - 递归出口：直接输出n，如果n<=9，即n为1位数
    - 递归式子：输出个位数n%10，再递归调用reverse(n/10) 输出前n-1位，如果n为多位数

[例10-4] 源程序

- 由于直接输出结果，因此函数返回类型为void

```
void reverse(int num)
{
    if (num <= 9) {
        printf ("%d", num);                    /* 递归出口 */
    } else {
        printf ("%d", num%10);
        reverse (num/10);                      /* 递归调用 */
    }
}
```

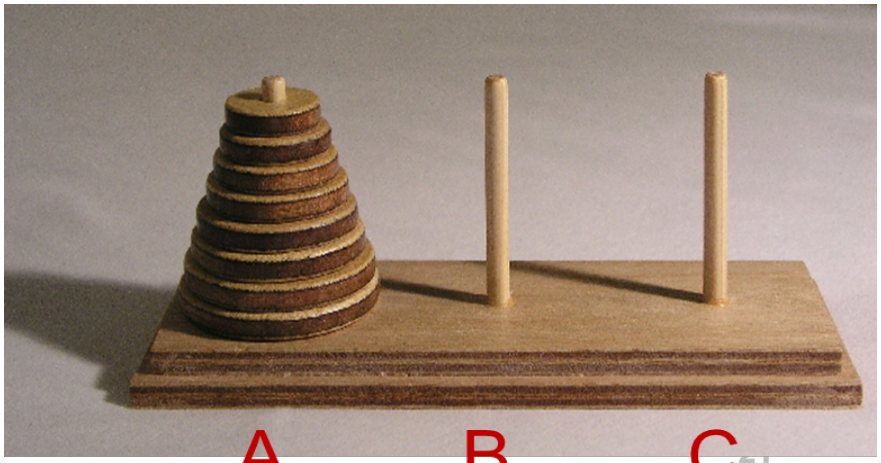
思考1：如何修改代码，递归实现将整数顺序输出？

思考2：设计递归函数，实现非负整数数字求和，如sumOfDigitsOf(137) = 11

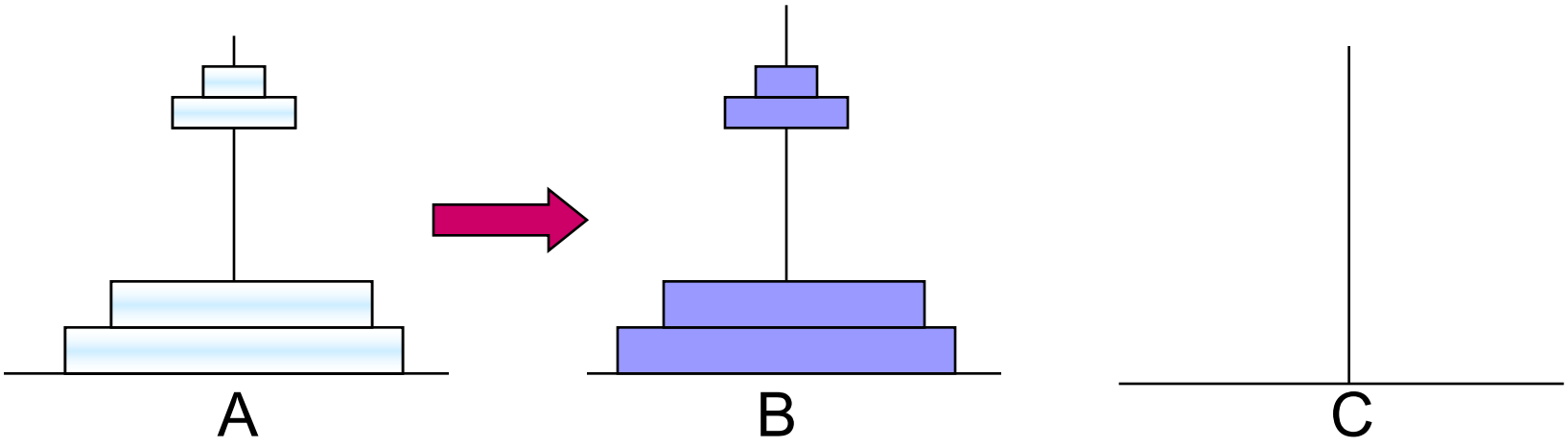
思考3：设计递归函数，实现字符串逆序输出，如reverseOf("TOP") = "POT"

[例10-5] 汉诺(Hanoi)塔问题

- 汉诺塔问题是源于印度一个古老传说的益智玩具。大梵天创造世界的时候做了三根金刚石柱子，在一根柱子上从下往上按照大小顺序摞着64片黄金圆盘。大梵天命令婆罗门把圆盘从下面开始按大小顺序重新摆放在另一根柱子上
- 规则：在小圆盘上不能放大圆盘，在三根柱子之间一次只能移动一个圆盘



[例10-5] 汉诺(Hanoi)塔问题



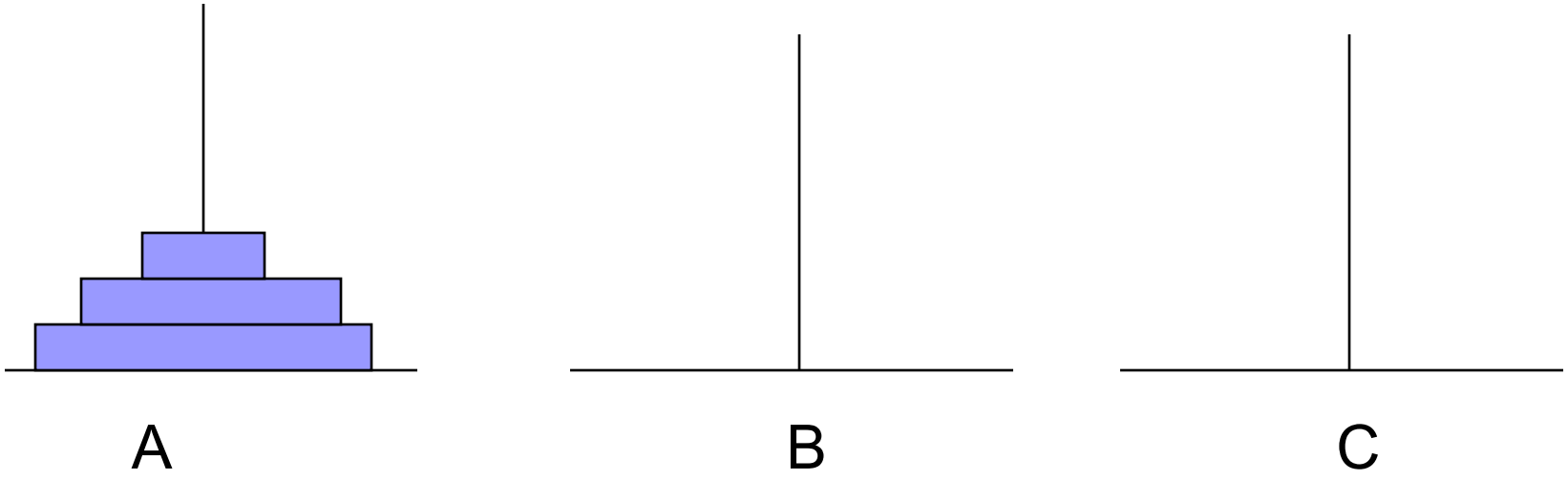
将 64 个盘从座A搬到座B

(1) 一次只能搬一个盘子

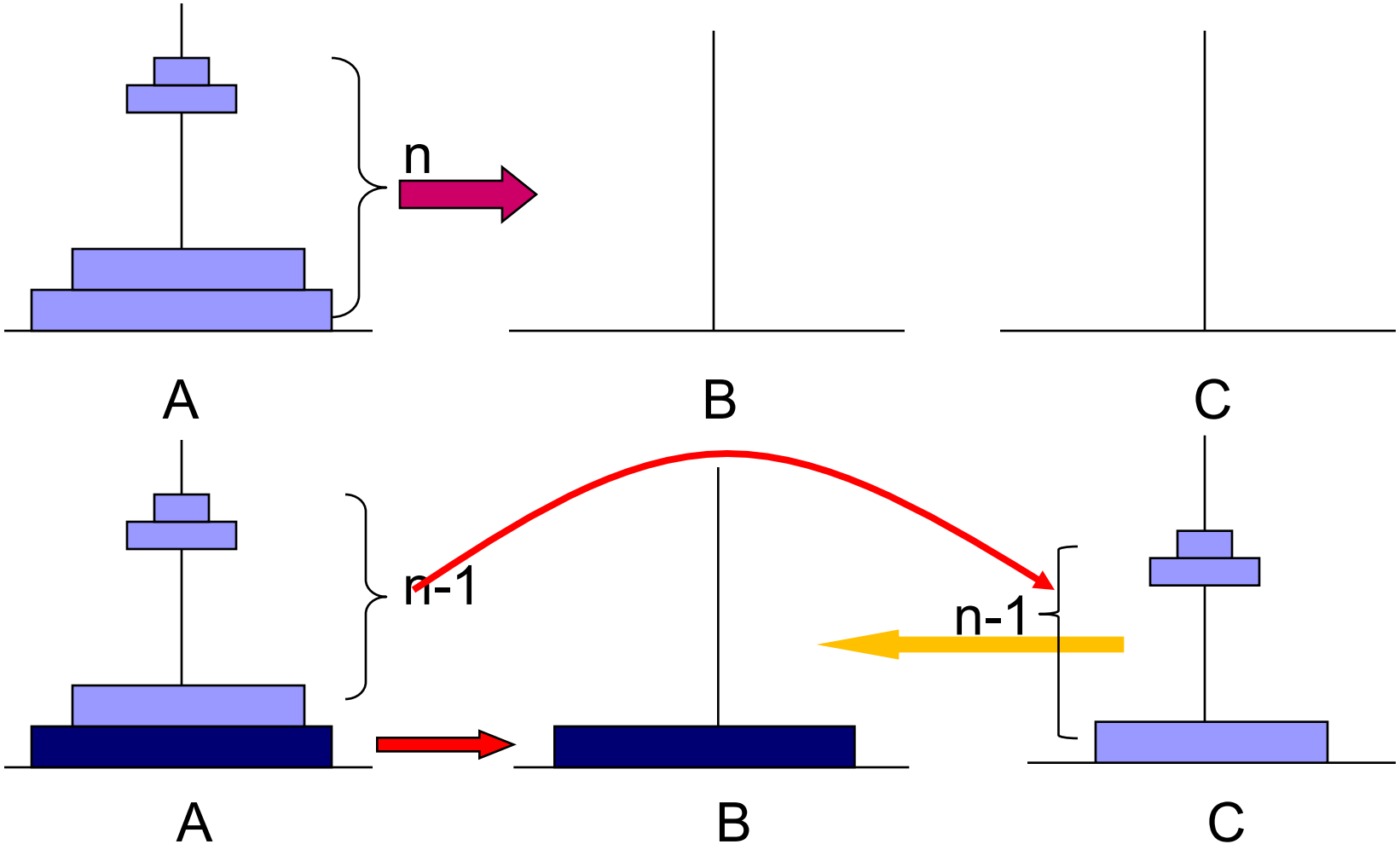
(2) 盘子只能插在A、B、C三个杆中

(3) 大盘不能压在小盘上

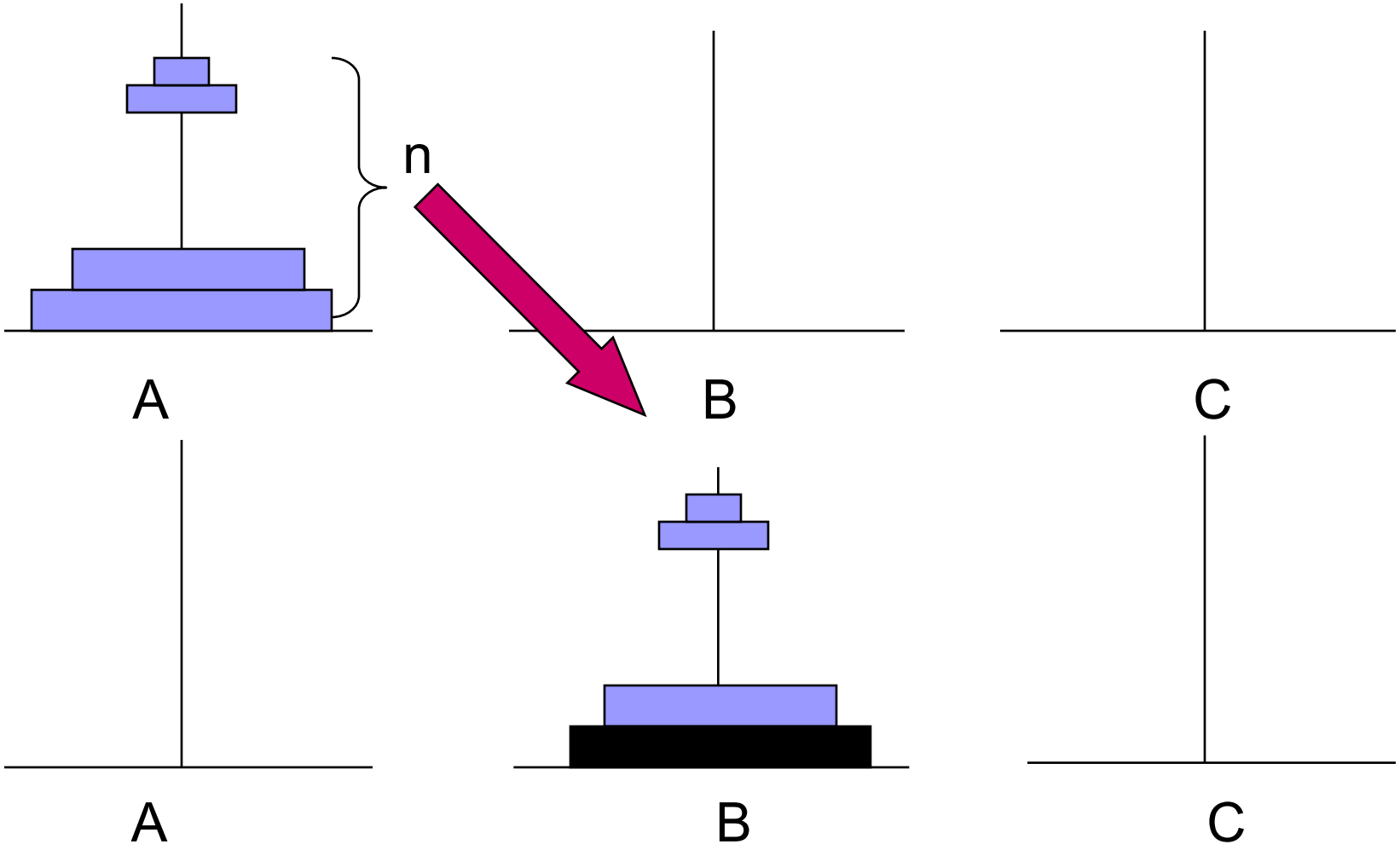
[例10-5] 汉诺(Hanoi)塔问题分析



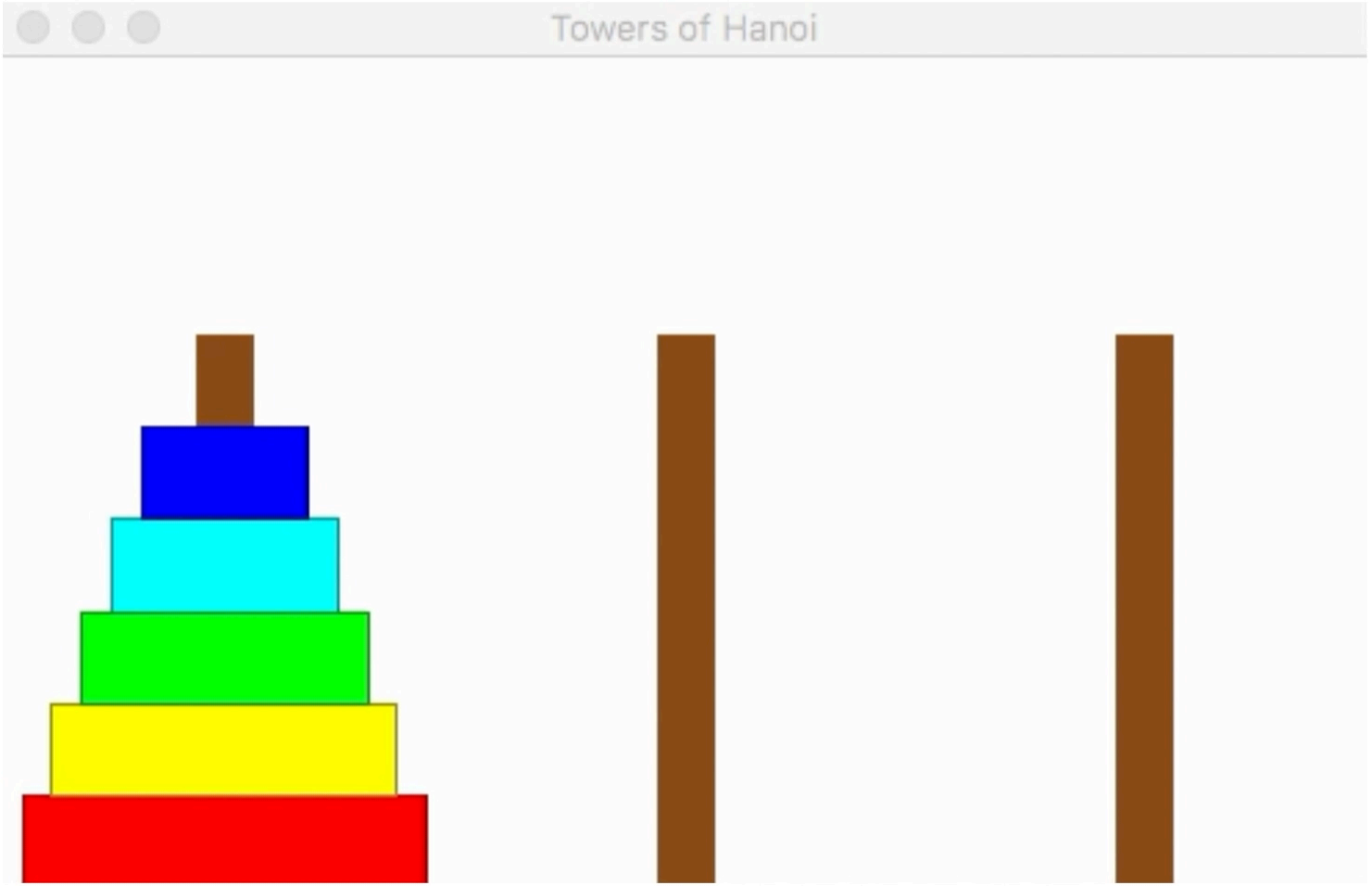
[例10-5] 汉诺(Hanoi)塔问题分析



[例10-5] 汉诺(Hanoi)塔问题分析

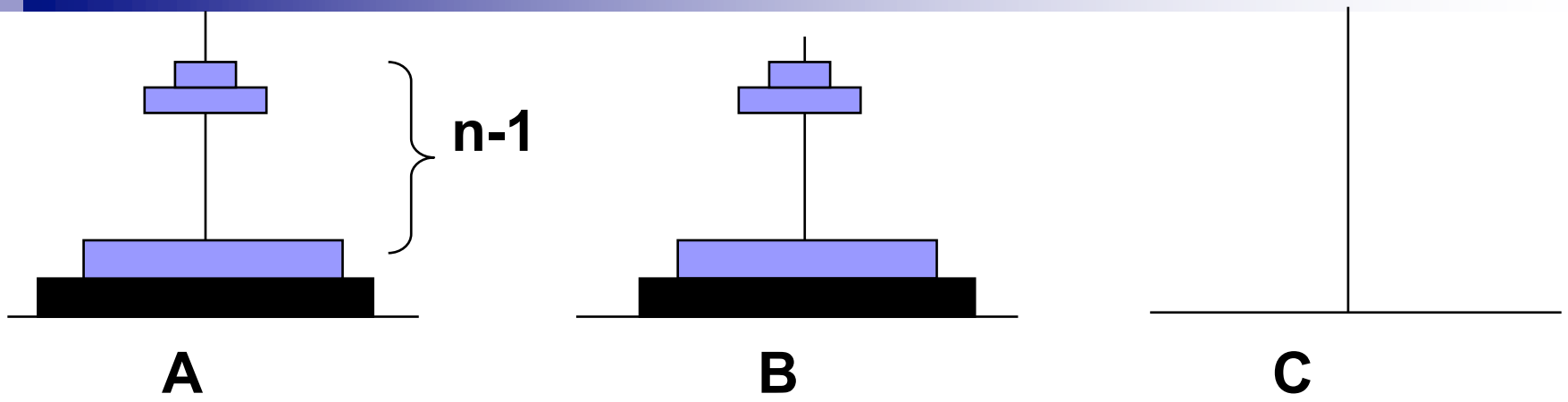


[例10-5] 汉诺(Hanoi)塔问题分析



[例10-5] 汉诺(Hanoi)塔问题解析

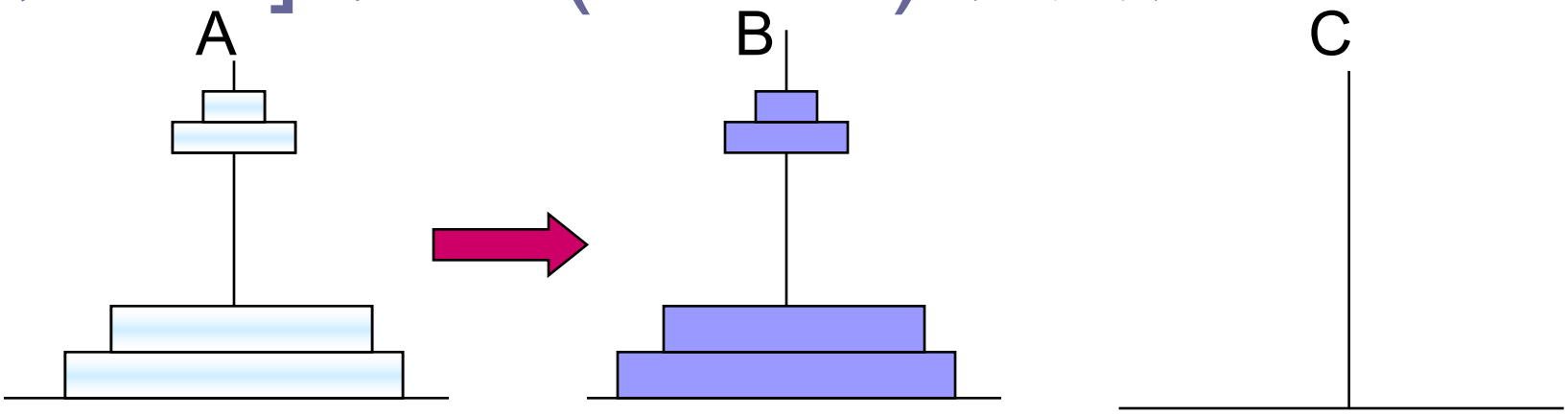
- 递归方法的两个要点
  - 递归出口：一个盘子的解决方法；
  - 递归式子：如何把搬动64个盘子的问题简化成搬动63个盘子的问题。
- 把汉诺塔的递归解法归纳成三个步骤
  - n-1个盘子从座A搬到座C
  - 第n号盘子从座A搬到座B
  - n-1个盘子从座C搬到座B



算法：

```
hanio(n个盘，A→B， C为过渡)
{
    if (n == 1)
        直接把盘子A→B
    else {
        hanio(n-1个盘，A→C， B为过渡)
        把第n号盘 A→B
        hanio(n-1个盘，C→B， A为过渡)
    }
}
```

[例10-5] 汉诺(Hanoi)塔问题



一路递归思路
二路递归思路
多路递归思路

```
hanio(n个盘，A→B，C为过渡)
{
    if (n == 1)
        直接把盘子A→B
    else {
        hanio(n-1个盘，A→C，B为过渡)
        把n号盘 A→B
        hanio(n-1个盘，C→B，A为过渡)
    }
}
```

例10-1 源程序

```
/* 搬动n个盘，从a到b，c为中间过渡 */
void hanio(int n, char a, char b, char c)
{  if (n == 1) {
       printf("%c-->%c\n", a, b);
   } else {
       hanio(n-1, a, c, b);
       printf("%c-->%c\n", a, b);
       hanio(n-1, c, b, a);
   }
}
int main(void)
{    int n;
     printf("input the number of disk: " );
     scanf("%d", &n);
     printf("the steps for %d disk are:\n", n);
     hanio(n, 'a', 'b', 'c') ;
     return 0;
}
```

```
input the number of disk: 3
the steps for 3 disk are:
a-->b
a-->c
b-->c
a-->b
c-->a
c-->b
a-->b
```

input the number of disk: 3
the steps for 3 disk are:

开始

动态展示

结束

第1步： a-->b

第2步： a-->c

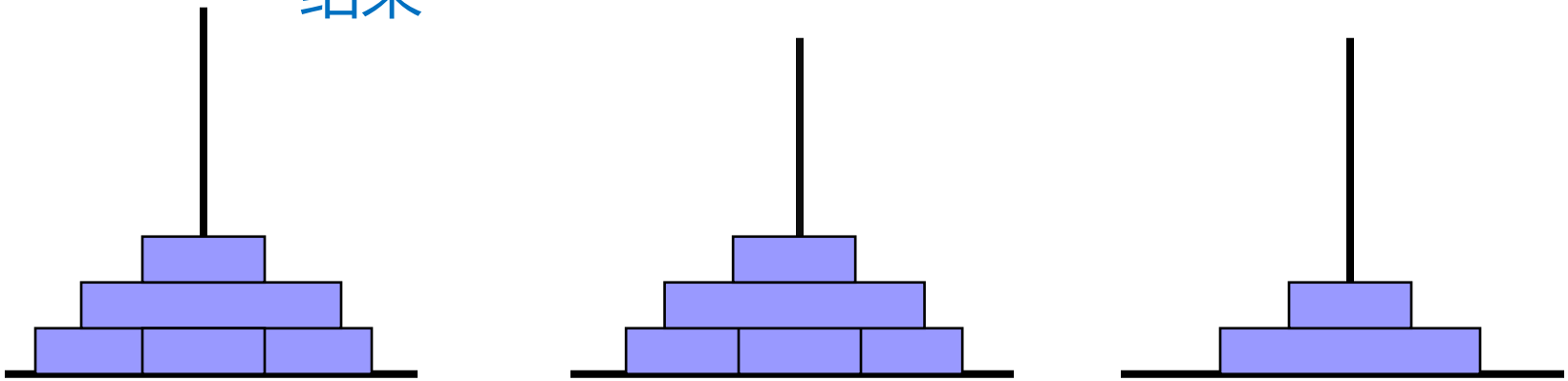
第3步： b-->c

第4步： a-->b

第5步： c-->a

第6步： c-->b

第7步： a-->b



[例10-6] 分治法求解金块问题

- 老板有一袋金块，两名最优秀的雇员每人可以得到其中的一块，排名第一的得到最重的金块，排名第二的则得到袋子中最轻的金块
- 输入n(共n块，2≤n≤100)及n个整数，用分治法求出最重金块和最轻金块
  - 定义递归函数max(int a[ ], int m, int n)，在a[m]~a[n]中找出最大值
    - 递归出口： a[m]，当m == n，即a中只有1个元素
    - 递归式：将数组a分割为两部分，分别递归求最大值
      - k = (m + n) / 2;
      - u = max(a, m, k);
      - v = max(a, k+1, n);

分治法，“分而治之”，把一个复杂的问题分成两个或多个相同或相似的子问题，再把子问题分成更小的子问题直到最后子问题可以简单地直接求解，原问题的解即子问题的解的合并

[例10-6] 源程序

算法思想(递推和递归):
穷举[例4-11 搬砖问题]
贪心[例4-12 找零钱]
分治[例7-7　二分查找]
动态规划

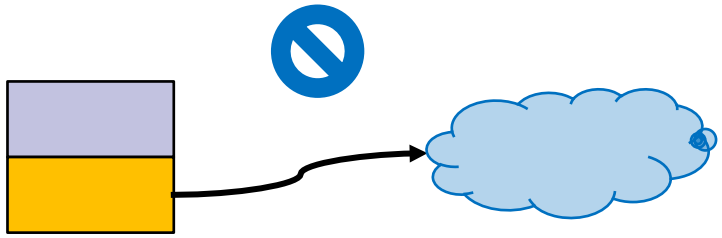
```
/* 分治法求a[m]~a[n]中最大值的递归函数 */
int max(int a[ ], int m, int n)
{
    int k, u, v;
    if (m == n) {
       return a[m];
    }
    k = (m + n) / 2;          /* 计算中间元素的下标k */
    u = max(a, m, k);         /* 在a[m]~a[k]中找出最大值 */
    v = max(a, k+1, n);       /* 在a[k+1]~a[n]中找出最大值*/

    return (u > v) ? u : v;   /* 返回u和v中较大的值 */
}
```

递归对复杂问题进行分解
- n分解为n和(n-1)
- (m, n)分解为(n, m%n)
- n分解为1~n/2和n/2+1~n
- ……

链表操作的递归实现

- 链表是一种存储序列数据的数据结构
  - 空链表 (递归出口)
  - 当前节点 + 剩余节点
    - n x (n-1)!



- 链表删除 (递推版本和递归版本)

```
void deleteList(Node *list) {
    Node *next;
    while (list) {
        next = list->next;
        free(list);
        list = next;
    }
}
```

```
void deleteList(Node *list) {
    if (list == NULL)
        return;
    deleteList(list->next);
    free(list);
}
```

链表操作的递归实现

- 链表长度 (递推版本和递归版本)

```
int lengthOf(Node *list) {
    int length = 0;
    while (list) {
        length++;
        list = list->next;
    }
    return length;
}
```

```
int lengthOf(Node *list) {
    if (list == NULL)
        return 0;
    else
        return 1 + lengthOf(list->next);
}
```

- 打印链表 (递推版本和递归版本)

```
void printList(Node *list) {
   while (list) {
        printf("%d ", list->data;
        list = list->next;
   }
}
```

```
void printList(Node *list) {
    if (list == NULL) return;
    else {
        printf("%d ", list->data);
        printList(list->next);
    }
}
```

课堂练习：Fibonacci数列的递归实现

fib(g) = 1　　g<=1

fib(g) = fib(g-1)+fib(g-2)　　g>=2

课堂练习：利用递归函数计算x的n次幂

```
int power(int x, int n)
{
    int i, result = 1;
    for (i = 0; i < n; ++i)
        result = x * result;
    return result;
}
```

递推法

典型的递归函数结构

```
returnValue recursiveFunction(parameter) {
    if (test for simple case) {
        Compute the solution without recursion
    } else {
        Break the problem into a subproblem of the same form,
            where "parameter" becomes "newParameter"
        Call recursiveFunction(newParameter)
        Get the result of the subproblem and update
    }
}
```

parameter → newParameter

- n!: n → (n-1)
- 最大公约数: (m, n) → (n, m%n)
- 最大值: 1~n →1~n/2, n/2+1~n
- x^n : n → n/2
- ……

update (y = result of the subproblem)

- n!：n * y
- 最大公约数: y
- 最大值：max(y1, y2)
- x^n : x * y or y * y
- ……

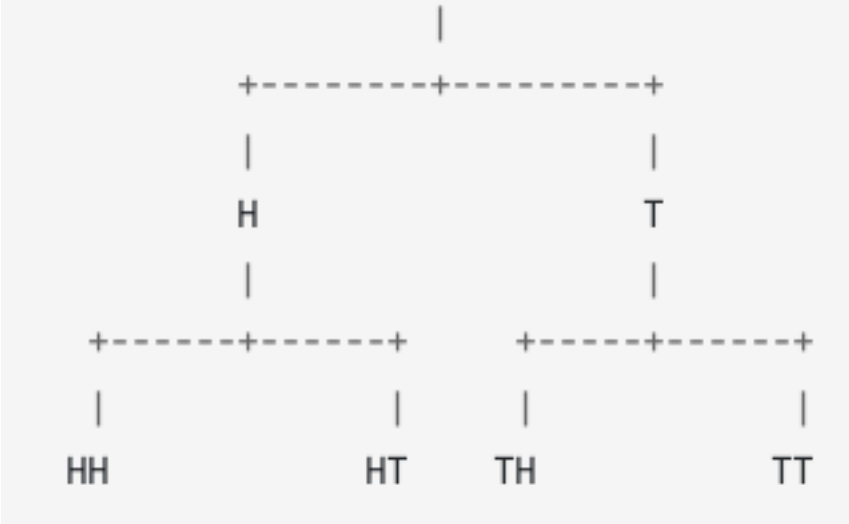
递归用于枚举所有可能性(拓展内容

- 序列生成 – 硬币抛N次的所有可能结果

N=2	N=3
H H	H H H
H T	H H T
T H	H T H
T T	H T T
	T H H
	T H T
	T T H
	T T T

是否存在自相似性(self-similarity)?

决策树(decision tree)



递归用于枚举所有可能性

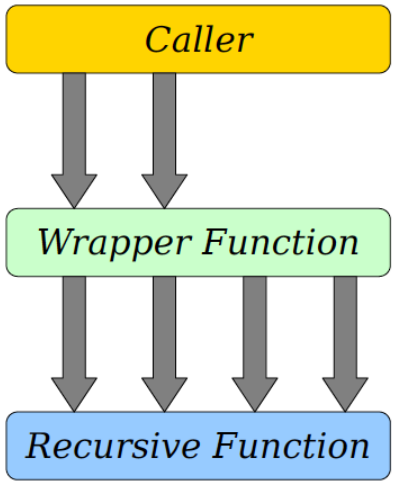
```
void generateSequences(int length, int next, char soFar[])
{
    if (length == 0) {
        soFar[next] = '\0';
        printf("%s\n", soFar);
    } else {
        soFar[next] = 'H';  // choose option 1
        generateSequences(length - 1, next + 1, soFar);

        soFar[next] = 'T';  // unchoose option 1, choose option 2
        generateSequences(length - 1, next + 1, soFar);
    }
}
void generate(int length)
{
    char soFar[MAXLENGTH];
    generateSequences(length, 0, soFar);
}
```

思考1：树是按层遍历？

思考2：可以用递推实现序列生成？

A wrapper function is a function that does some initial prep work, then fires off a recursive call with the right arguments



递归用于枚举所有可能性

- 序列生成 – 更多选择，更大搜索空间
  - 递归 + 递推

```
void generateSequences(int length, int next, char soFar[])
{
    if (length == 0) {
        soFar[next] = '\0';
        printf("%s\n", soFar);
    } else {
        char c;
        for (c = 'A'; c <= 'Z'; c++) {
            soFar[next] = c;    // choose current option
            generateSequences(length - 1, next + 1, soFar);
        }
    }
}
```

递归回溯

- 递归回溯(recursive backtracking)用于枚举所有可能性，搜索所有状态空间

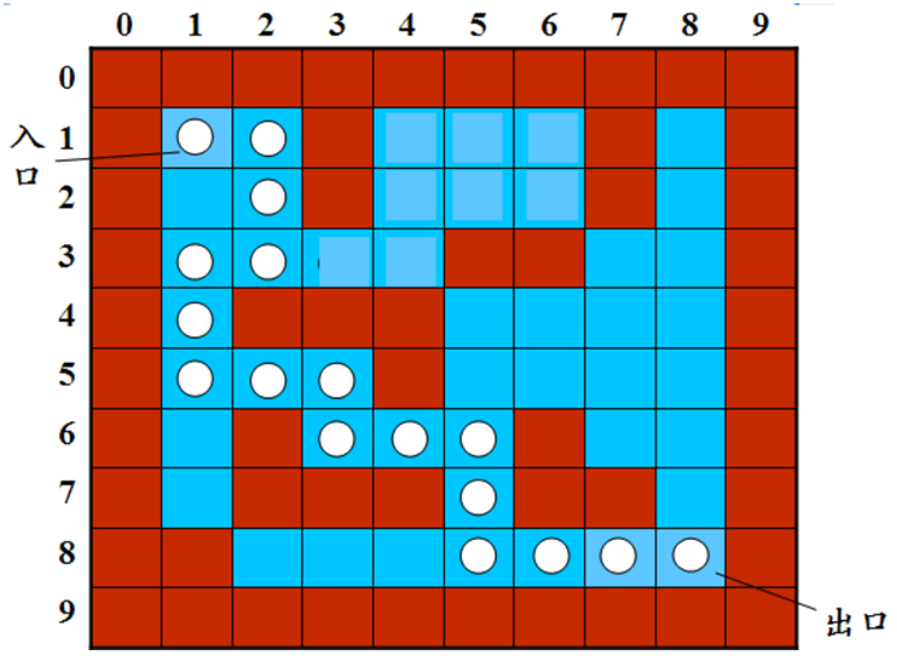
```
void explore(options, soFar) {
    if (no more decisions to make) {
        // base case
    } else {
        // recursive case, we have a decision to make
        for (each available option) {
            choose (update options/soFar)
            explore (recur on updated options/soFar) [递归]
            unchoose (undo changes to options/soFar) [回溯]
        }
    }
}
```

递归回溯

- 序列生成 – 给定选择

```
void explore(char *options, int length, int next, char soFar[])
{
    if (length == 0) {          // no more decisions to make，soFar已经生成
        soFar[next] = '\0';
        printf("%s\n", soFar);
    } else {
        int i;
        for (i = 0; i < strlen(options); ++i) {
            soFar[next] = options[i];  // 使用当前option更新soFar
            // 使用更新的soFar和相关参数递归搜索
            generateSequences(options, length - 1, next + 1, soFar);
            // unchoose (undo changes to options/soFar)，与choose合并
        }
    }
}
```

用递归回溯求解迷宫问题



求迷宫路径算法的基本思想

- 若当前位置“可通”，则纳入路径，继续前进；
- 若当前位置“不可通”，则后退，换方向（按东南西北的顺序）继续探索；
- 若四周“均无通路”，则将当前位置从路径中删除出去

用递归回溯求解迷宫问题

```
void explore(int maze[10][10], int x, int y, int length, int* soFar)
{
    if (isEixt(maze, x, y)) {  // no more decisions to make
        if (length < *soFar)
            *soFar = length;
    } else {
        int i, dx[4] = {1, 0, -1, 0}, dy[4] = {0, 1, 0, -1};  // options
        for (i = 0; i < 4; ++i) {
            int nx = x + dx[i], ny = y + dy[i];
            if (maze[nx][ny] == 0) {  // 可走的空位
                maze[nx][ny] = 1;  // choose (update options/soFar)
                // explore (recur on updated options/soFar)
                explore(maze, nx, ny, length + 1, soFar);
                maze[nx][ny] = 0; // unchoose (undo changes to options/soFar)
            }
        }
    }
}
```

递归回溯应用举例

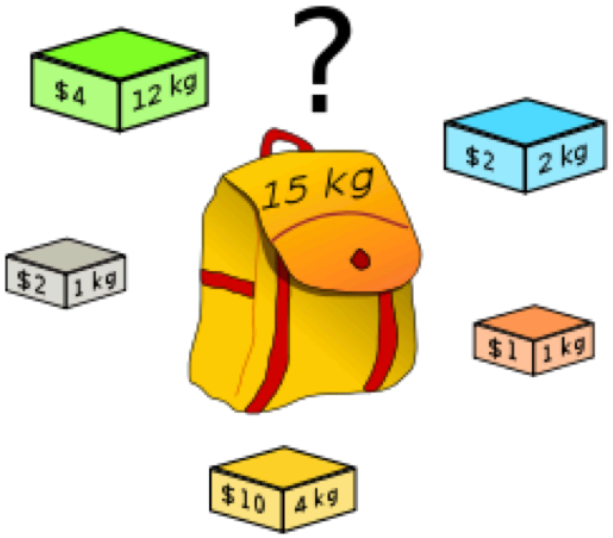
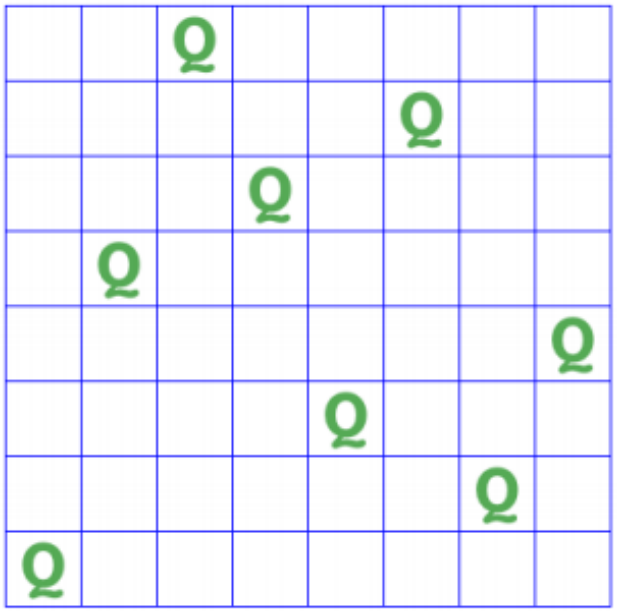


image courtesy of wikipedia.org

背包问题



八皇后问题

				3		2		
	2			4		3	5	
		7	1					8
8					1		6	2
	6						7	
2	7		6					3
7					8	9		
	5	6		9			4	
		8		1				

数独

自动纠正，单词推荐

输入"tounf"，推荐"young", "round", "found"

g → tyfhcvb

决策树搜索模板

```
bool search(currentState) {
    if (no more moves possible from currentState) {
        return isSolution(currentState);
    } else {
        for (option : moves from currentState) {
            nextState = takeOption(currentState, option);
            if (search(nextState)) {
                return true;
            }
        }
        return false;
    }
}
```

递归程序设计总结

- 递归是一种将问题简化为相同形式的较小问题来解决问题的技术
- 用递归解决问题需要满足两个条件
  - 问题可以逐步简化成自身较简单的形式(递归式)
  - 递归最终能结束(递归出口)
- 递归程序设计 (递推、递归、分治)
  - 阶乘、最大公约数、整数逆序输出、汉诺塔问题、分治法求解金块问题、链表操作的递归实现
- 拓展内容：用递归实现枚举所有可能性
  - 递归回溯模板 (PTA练习)